Università degli Studi di Pisa

Dipartimento di Informatica
Dottorato di Ricerca in Informatica

Ph.D. Thesis

Discovery of Unconventional Patterns for Sequence Analysis: Theory and Algorithms

(SSD) INF 01

Giovanni Battaglia

Supervisor
Prof. Roberto Grossi

December 3, 2011

Abstract

The *pattern discovery* task (or equivalently *motif inference*) is the knowledge discovery process that, given a dataset and some constrains either on the combinatorial pattern structure or on the occurrence lists, returns all the patterns satisfying the given constraints.

In this thesis, we consider the problem of discovering patterns in sequential data, such as texts, biological sequences, access logs, etc. When it comes to defining what is a pattern, several classes have been proposed in literature. For example, *rigid patterns* like $p = \mathtt{c}\circ\mathtt{tc}$ where the don't care symbol $\circ$ matches any single character of the input alphabet Σ, or *gapped patterns* like $p = \mathtt{ctt} - 2, 3 - \mathtt{tc}$ where the gap represents either a sequence of 2 or 3 don't cares (we refer to [154] for a thorough discussion of the above classes of patterns).

The adjective "unconventional" in the title of this thesis is referred to the unusual combinatorial structure of the patterns we are going to investigate. In fact, while the classic literature of this field focus on string patterns (maybe with wildcards), our line of research explores three different kind of patterns: *mask patterns*, where each pattern represents a set of string patterns with wildcards, *permutation patterns* where each pattern is a multiset of characters, and the order of the contained symbols doesn't matter, and *transposons* which, roughly speaking, represent the non-conserved regions of a global alignment.

To everyone that has believed and still believes in me.

Acknowledgments

I would like to thank everyone that has believed in me for making this work possible. In particular, I am very grateful to my advisor Professor Roberto Grossi for his guidance, and to my co-authors Nadia Pisanti, Roberto Marangoni and Giulia Menconi for their invaluable contribution. I am thankful to my parents for their unconditional support, and to every teacher, classmate, friend that I met in my life, who have invested one hour of their time in me. Last but not least, I would like to thank Chiara. This work would not be possible without her continuous encouragements and her belief in me.

Contents

Introduction

The biology community is collecting a large amount of raw data, such as the genome sequences of organisms, microarray data, interaction data such as gene-protein interactions, protein-protein interactions, etc. This amount is rapidly increasing and the process of understanding the data is lagging behind the process of acquiring it. An inevitable first step towards making sense of the data is to study their regularities focusing on the non-random structures appearing surprisingly often in the input sequences: *patterns*.

It is not easy to define precisely what is a pattern, since several classes of patterns have been proposed in literature, and each one of them is characterized by its combinatorial structure (a string, a string with wildcards, a set/multiset, etc) and a notion of occurrence that specifies when the pattern occurs at a specific position of the input dataset (see [154] for an overview of some classes of patterns).

The *pattern discovery* task (or equivalently *motif inference*) is the knowledge discovery process that, given a dataset and some constrains either on the combinatorial pattern structure or on the occurrence lists, returns all the patterns satisfying the given constraints.

In the following chapters we will focus on the frequency constrain, defining the minimum number of times a pattern must occur but, for example, we could also constraint the returned patterns to have at most d wildcards, or limit the maximum distance in a pattern between two consecutive identical characters to be smaller than a given threshold l, etc.

At this point, it should be clear the difference between *pattern matching* and *pattern discovery*. While in the former, a pattern p (or a set of patterns) is given in input together with a dataset and the task is to return all the occurrences of p in the input dataset, in the latter no pattern is provided in input but the reverse task is to be performed: a pattern discovery algorithm must return all the patterns satisfying the input constraints, together with their occurrence lists.

The following chapters are devoted to three different kinds of patterns: *mask patterns*, where each pattern represents a set of string patterns with wildcards, *permutation patterns* where each pattern is a multiset of characters, and *transposons* which, roughly speaking, represent the non-conserved regions of a global alignment. However, before describing in detail these pattern discovery problems, Chapter 1 gives some preliminary definitions, introducing the basic biological concepts about genome structure and organization that will be used in the following chapters.

Chapter 2 is devoted to mask patterns (parts of our contributions have been published in [14] and [15]).

In this chapter, we follow a new approach based on modeling motifs by using simple binary patterns, called *masks*, that implicitly represent *families of patterns* in the input text T (instead of individual patterns). For example, mask 101001 represents both a∘t∘∘c and t∘g∘∘a: each 1 represents a solid symbol while each 0 represents the don't care symbol ∘, matching all the symbols of the alphabet. A mask has *quorum* if at least one of its represented patterns occurs q or more times in the given sequence T.

As it should be clear from the above informal definition, we can describe interesting repetitions in a sequence, using a description (mask) that is more succinct than before. Therefore, we aim at giving rise to a smaller set of output motifs. Intuitively, consider some patterns that occur at least q times each and that also share the same structure, meant as a certain concatenation of solid and don't care symbols. Since they originate from the same mask, we take this mask as a motif. Moreover, any two patterns sharing the same structure but having a different number of occurrences in T (still at least q in number), which were previously considered as different motifs, are now giving rise to the same motif by our definition of mask. Since each mask can be seen as a binary string, we have potentially 2^L masks of size L, instead of $(|\Sigma| + 1)^L$ classical motifs.

The topic of Chapter 3 is a different kind of pattern, that is defined by merely its symbol content, ignoring the order in which the symbols appear. For example, if we are given in input the two strings $T_1 = \ldots\mathtt{a}\underline{\mathtt{bcde}}\mathtt{fg}\ldots$ and $T_2 = \ldots\mathtt{m}\underline{\mathtt{ebdc}}\mathtt{ma}\ldots$, it is of interest to note that the symbols in set $S = \{\mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}\}$ are consecutive in both strings, although they do not occur in the same order.

The above set of symbols is often called a *gene cluster* (if symbols are genes) or π-pattern in general, since the first pattern can be numbered 1 to 4 and every other occurrence is a permutation of $\{1, 2, 3, 4\}$.

Several genome models formalizing the notion of gene cluster have been proposed in literature in the last two decades. In Chapter 3, we formally recall the notion of π-pattern, discussing how to detect the π-patterns satisfying a given quorum threshold q, selecting the π-patterns that show a highly-conserved structure across their occurrences. (Parts of this chapter have been published in [17] and [16].)

The last chapter of this thesis is devoted to the *transposon detection problem.* (Parts of this chapter have been published in [26, 78, 79].)

Transposons are sequences in the genome that are mobile, namely they are able to transport themselves to other locations of the genome, without using any extra-chromosomal vector (such as viruses), autonomously moving from one site of the genome to another.

In the past, transposons have been considered to be "genomic parasites" but more recently, significant beneficial attributes for facilitating evolution have been recognized [67]. In fact, transposons act to increase the evolvability of their host genome and provide a means of generating genomic modifications. In some sense, they can be thought as generators of variations upon which natural selection can act, and represent one of the main engines of genome evolution [31].

Traditional approaches for transposon detection are mainly based on consensus-like pattern search, that scan the investigated genome against an already identified library of known transposons. These approaches mainly suffer two limitations. First, since the transposons to be searched must be given in input together with the genome to be analyzed, we must know a priori all the classes of transposons that occur in the given genome. Moreover, these approaches can only detect transposons that are highly similar to the input ones, but they fail in identifying previously unknown classes of transposons. This is not an issue in species where the structure of transposable elements is well characterized (for example in yeasts), but it is unapplicable where transposons show a much more variable structure, as for example in the human genome [84].

The second issue of the pattern search approach concern the noisy nature of the analyzed genomes. In fact, modern sequencing technologies reduced the cost of the sequencing process, but the released genomic sequences usually have a low-coverage, and they are rich of *unresolved* bases.

To overcome the above limitations, in Chapter 4 we describe an alternative approach that can be applied when multiple copies of the input genome are available. The core idea is simple. Our alternative approach relies neither on homology nor structural features of transposons, but on their behavior. Recall that transposons are mobile elements. Hence, if we align the given genome with its copies, we can detect transposition events by detecting large insertions or deletions in the global alignment of the input sequences. In other words, our approach recasts a pattern search problem where the transposons to be searched are part of the input, as a pattern discovery problem. In Chapter 4, we assess the precision and the recall of our pattern discovery approach by a case study where we analyzed the dataset of 38 strains of the *S. cerevisiæ* yeast that has been recently released in [130].

Chapter 1

Structure of the Genome

Before discussing the pattern discovery algorithm presented in the following chapters, in the current chapter we recall some basic biological definitions as the concept of gene cluster, repeat, and transposon, that are essential to better understand the biological relevance of the problems discussed in the following chapters. A comprehensive discussion of the genome structure at a molecular level is beyond the scope of the current chapter, and we omit all the biochemical details that are not strictly necessary for our purposes. We address the interested reader to [128] for a rigorous and detailed description of all the contents of the following sections.

This chapter is organized as follows. After describing the structure of the genome in Section 1.1, we focus on the basic unit of heredity: genes. More precisely, while in Section 1.2 we recall the notion of gene, briefly describing the gene expression process, in Section 1.3 we focus on the distribution of genes across the genome. In fact, genes are not randomly distributed across genomes. Genes that are functionally related tend to be clustered in group of physically adjacent genes, in order to facilitate the control of their expression [30]. Until recent years such group of genes were thought to exist solely in prokaryotes, since much more complicated mechanisms for gene regulation were known in eukaryotes. However, the discovery of the first group of genes in eukaryotes in the early 1990s radically changed the scenario [30], creating a new exciting field of investigation.

We conclude this chapter focusing on the intergenic regions of the genomes, which are the parts of the genome that fall between genes. Although the role of the intergenic regions is not fully understood, it is clear that they are not useless “junk-DNA” regions, since mutations occurring to these regions are responsible of severe genetic diseases such as cancer and Huntington’s disease [175]. On the other hand, while the structure of the coding regions of the genome containing genes is well understood, this is not the case of intergenic regions, whose structure is much more complicated and rich in repeated sequences. In recent years, given the repetitive nature of these regions, the identification of such repeated sequences has attracted a lot of interest in the algorithmic community, specifically in the pattern discovery field. In Section 1.4, we briefly describe the main biochemical properties of the

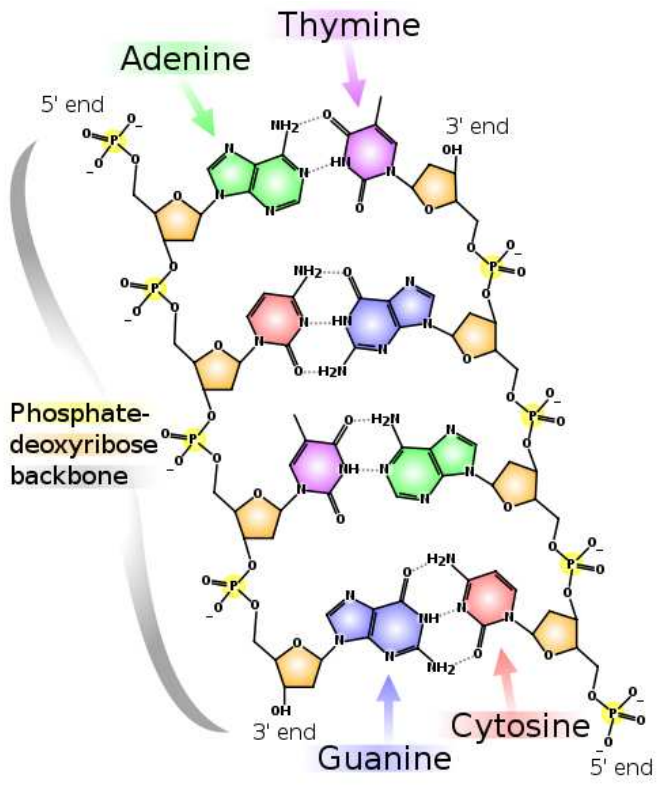


Figure 1.1: Double-stranded structure of the DNA molecule. (Redrawn from `en.wikipedia.org/wiki/Dna`.)

intergenic regions, presenting the most important families of repeats that have been discovered.

1.1 Introduction to the Genome Structure

Living organisms are categorized in two main categories: cellular and non-cellular. While the former includes all living organisms that are made up of cells, the non-cellular life forms, namely *viruses*, are merely genetic material surrounded by a protein membrane, that are most often considered replicators rather than forms of life, since they can replicate by creating multiple copies of themselves, but only inside a host cell.

Cellular organisms are further classified in *prokaryotes*, that lack a cell nucleus and any other membrane-bound *organelles*, and *eukaryotes* that are structurally more complex than prokaryotes because they have a cell nucleus containing the genetic materials, and several other organelles that are responsible for the cell metabolism. For example, *S. cerevisiæ*, which will be the subject of our investigation in Chapter 4, is an eukaryotic organism (more precisely a fungus), while *E.*

Coli is a prokaryotic organism (more precisely a bacteria). Both in prokaryotes and in eukaryotes the biological information needed for the organism existence and reproduction is encoded by the *genome*. The genomes of all the cellular organisms are made of DNA while the genomes of many viruses are made up of RNA. DNA consists of two long chains of nucleotides twisted into a double helix and joined by hydrogen bonds between the complementary bases. As shown in Figure 1.1, each nucleotide is composed by a five-carbon sugar (represented as a pentagon), a phosphate group (`P`), and one of four nitrogenous bases: Adenine (`A`), Thymine (`T`), Cytosine (`C`), and Guanine (`G`). Each nucleotide on one strand forms a bond with just one type of base on the other strand. *Purines* form hydrogen bonds to *pyrimidines*, with `A` bonding only to `T`, and `C` bonding only to `G`. This arrangement of two nucleotides binding together across the double helix is called a *base pair*, while the linear order in which the nucleotides are arranged is called the DNA sequence.

In the double-stranded DNA molecule, each strand has a direction and the direction of the nucleotides in one strand is opposite to their direction in the other strand. As we can see in Figure 1.1 the asymmetric ends of DNA strands are called the 5' (five prime) and 3' (three prime) ends, with the 5' end having a terminal phosphate group and the 3' end a terminal hydroxyl group (OH). It follows that, the relative positions of structures along a strand of nucleic acid, including genes and protein binding sites, are usually noted as being either *upstream* (towards the 5' end) or *downstream* (towards the 3' end). The DNA strand orientation is by convention 5'→3', hence the 5'→3' DNA strand is designated, for a given gene, as *sense* (or *positive*) strand, while the 3'→5' strand is referred as the *antisense* (or *minus*) strand. Because of this convention, the DNA sequence in the 5'→3' strand in Figure 1.1 is $x = \texttt{ACTG}$, while the negative strand encodes the string $-x = \texttt{CAGT}$ representing the *reverse complement* of x.

Differently from DNA, RNA is a single-stranded polymeric molecule. RNA nucleotides are made up of *ribose* sugar, and have the same nitrogenous bases of DNA except that the Thymine (`T`) is replaced with Uracil (`U`). As with DNA, also RNA nucleotides are paired: `A` pairs with `U`, and `G` pairs with `C`.

The size of the genome varies significantly in eukaryotes organisms. In general, the size of the genome increases with the complexity of the organism, in fact the simplest eukaryotes, such as fungi, have the smallest genomes, while the higher ones, such as vertebrates and plants, have the largest ones. However, there are some notable exceptions. For example, while the human genome (3 billion bases) is much larger than the yeast genome (12 million bases), it is much smaller than that of the *Amoeba dubia* (200 billion bases), that is a unicellular organism as simple as yeast. In recent years, this surprising phenomenon has been explained by proving that these large genomes contain a large amount of *repetitions*, that have undergone a massive proliferation in the genome of certain species. As we will discuss later, the identification of such repeated genomic sequences, is one of the main motivation behind pattern discovery.

In eukaryotic organisms the genomic materials are distributed in three organelles:

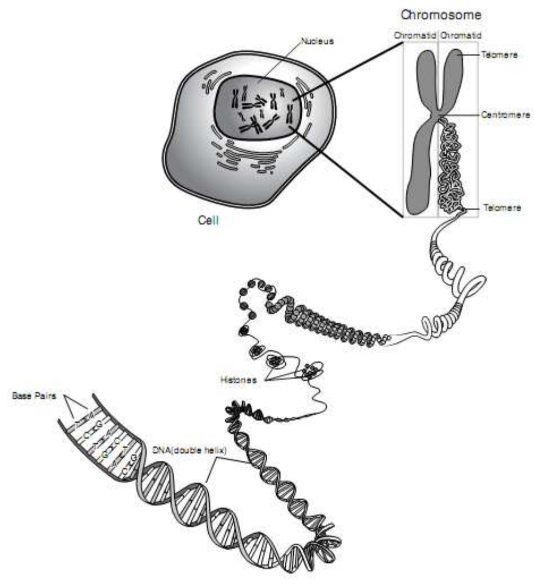


Figure 1.2: Schematization of the structure of a chromosome during the cell division process, when the two chromatin are attached by the centromere. Telomeres at the ends of the chromosome protect the chromosome from deterioration. (Redrawn from `www.accessexcellence.org`.)

nucleus, mitochondrion, and chloroplast (that are preset in photosynthetic organisms only). While the nuclear genome is the "general-purpose" genome that codes for proteins which are used by all types of cells of the organism, the mitochondria and chloroplast genomes essentially code for proteins that are used in mitochondria to produce energy, and in chloroplasts in the photosynthesis process. The nuclear genome is much different from the mitochondria and chloroplast genomes. In general, the nuclear genome is much longer (the human nuclear genome is 3 billion bases long, while the human mitochondrial genome has 16k bases), it is split into atomic units called *chromosomes* (the non-nuclear genomes are organized as several copies of a single, circular molecule), and also the "semantic" of its parts is different from that of the non-nuclear genomes (the same DNA triplet can encode a different amino acid in nuclear and non-nuclear genome). Although the non-nuclear genomes play a fundamental role in eukaryotic organisms, and its mutation can yield to severe genetic diseases, in the rest of this thesis the word genome will only refer to nuclear genome.

The nuclear genome is packaged into atomic units of coiled DNA called *chromosomes*. Chromosomes vary widely between different organisms, both in shape (although there are many exceptions to this rule, in general eukaryotic cells have large linear chromosomes, while prokaryotic cells have smaller circular chromosomes), and in their size, that can range from thousands to billions of bases. In eukaryotes, nuclear chromosomes are packaged by proteins into a condensed structure called *chromatin*, that allows the very long DNA molecules to fit into the cell nucleus. The structure of chromosomes and chromatin varies through the cell cycle. Chromosomes may exist as either duplicated or unduplicated. Unduplicated chromosomes are single linear strands, whereas duplicated chromosomes (copied during synthesis phase) contain two copies joined by a *centromere*, as shown in Figure 1.2. The number of chromosomes differs significantly across different organisms, and it is neither related to the genome size, nor to the organism complexity. For example, the human nuclear genome contains 46 chromosomes: two sex chromosomes (X and Y), and 22 pairs of homologous chromosomes (one paternal and one maternal copy). They have not equal sizes, and if the smallest one is chromosome 21 (45 million bases), the largest one is chromosome 1 (279 million bases). On the other hand, *S. cerevisiæ* genome, which will be analyzed in Chapter 4, has 16 chromosomes whose size range over 200 thousand to 1.5 million bases.

Figure 1.2 shows the structure of an eukaryotic chromosome during the cell division process. It is composed of two parts: *centromere* and *telomeres*. The centromere is the region near the middle of the chromosome, that is used during cell division as the attachment point for the spindle fibers. The telomeres are the regions of repetitive DNA sequence at the end of a chromosome, that protect the end of the chromosome from deterioration or from fusion with other chromosomes. The remaining part of the chromosome encodes the genetic information.

1.2 Genes: the Units of Heredity

Inside each chromosome the genetic information is organized in *genes* that are segments of the chromosomes that code for a type of protein or for an RNA chain that has a function in the organism. In fact, genes are usually distinguished in *coding* and *non-coding* genes. Coding genes are transcribed to RNA and eventually translated to proteins. Non-coding genes are also transcribed to RNA but they are not translated to proteins. In fact, their products are other RNA molecules that are used in the protein synthesis process.

Genes in eukaryotes are not a continuous sequence of nucleotides. As shown in Figure 1.3, the coding segments containing the genetic information, *exons*, are interrupted by segments, called *introns*, that do not code for a protein, and whose function is still an argument of debate in the biological community. Figure 1.3 shows the role played by exons and introns in the *gene expression* process. Gene expression is the most fundamental level at which the genotype of a living organism gives rise to the phenotype. The genetic code stored in DNA is “interpreted” by gene expression, and the properties of the expression give rise to the phenotype (the observable characteristics) of the organism. More specifically, gene expression is the process by which the information contained in a gene is used in the synthesis of a

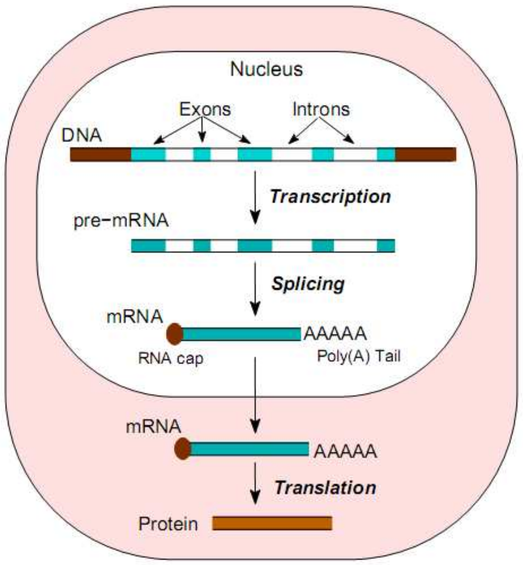


Figure 1.3: Schematization of the three main steps involved in the gene expression process: transcription, splicing, and translation. (Redrawn from `www.accessexcellence.org`.)

functional gene product, which is a protein in the case of coding genes, or an RNA molecule in the case of non-coding genes. As shown in Figure 1.3, it consists of three steps: *transcription*, *splicing*, and *translation*. During the *transcription* process, the segment of DNA encoding for the gene of interest is transcribed to a RNA molecule known as *precursor messenger* RNA (pre-mRNA). The pre-mRNA sequence encodes the reverse complement of the 3'→5' DNA strand, hence it is identical to the 5'→3' strand of DNA, but Thymine (`T`) is substituted by Uracil (`U`). After the transcription, during the *splicing* step, the introns are removed from the pre-mRNA sequence and a subset of exons is assembled to produce a messenger RNA molecule (mRNA) that will be the blueprint of the produced protein. Different subset of exons can be selected from the same pre-mRNA, producing different proteins from the same gene. This alternative splicing process is called *alternative splicing*, and it explains why the number of human protein is more than twice the estimated number of human genes. After a "maturation" period inside the nucleus where the mRNA can be further modified by enzymes by removing some parts of the sequence and by adding new sequences, the mature mRNA is exported to the cytoplasm, where the *translation* step takes place. In the translation phase, the mRNA sequence of nucleotide is used to produce the sequence of amino acids coding for the target protein by mapping each triplet of nucleotides (*codon*) in the mRNA to a specific amino acid. In the case of non-coding genes the translation phase is skipped, since the mature mRNA is the final gene product that is released from the cell.

The analysis of the human genome, together with that of the other sequenced eukaryotes, answered to several open questions about genome and its structure. However, there are several points that are still argument of debate in the scientific community. Two fundamental aspects that are not fully understood are the gene distribution inside a genome and the function of intergenic regions. They are the topics of Section 1.3 and 1.4, respectively.

1.3 Gene Distribution in the Genome

Genes are not uniformly distributed along the genome, but there are some regions that are richer of genes than others. Moreover, it frequently happens that functionally related genes are clustered in groups of physically adjacent genes (although there are some notably exceptions to this rule). For example, this is the case of *gene families*, which are groups of structurally identical genes, that usually have very similar biochemical functions because they have been originated by duplication of a single ancestral gene. One such family is the family of the genes for human haemoglobin subunits. The 10 genes are clustered in two different groups on different chromosomes, called the α-globin and β-globin loci, that are located in chromosome 16 and 11, respectively. Rearrangements of genes inside one of these clusters are possible, but in this case the gene products are expressed at improper stages of development.

Another example of genes that are physically adjacent in the genome are *operons*. An operon is a functional unit of genomic material containing a cluster of genes that are controlled by a single regulatory promoter. Operons are an atomic transcription unit. Namely, the physical adjacency of the cluster of genes in the operon allows the transcription process to be regulated by one regulator promoter gene only, which is physically located in the genomic sequence preceding the gene cluster. Such shared regulatory mechanism easily allows either to transcribe all the genes of the cluster or none of them by “switching on/off” one regulatory promoter only. Until recent years, operons were thought to exist solely in prokaryotes since much more complicated mechanisms for gene regulation were known in eukaryotes. However, the discovery of the first operons in eukaryotes in the early 1990s radically changed this scenario [30]. Also in the case of operons, the genes inside the cluster can be rearranged, affecting the operon functionality. In Chapter 3, we define the problem of detecting the above gene clusters as a pattern discovery problem, discussing a novel approach to select the most conserved of them.

1.4 Intergenic Regions and Repeats

One of the surprising facts revealed by the human genome project is that just 1.5% of the human genome is made of exons (belonging to both coding and non-coding genes), about 32.5% of the genome contains exons, while the remaining two thirds of intergenic regions have an unknown function and are rich in repeated DNA sequences. This is one of the main structural differences between human genomes (but also eukaryotic genomes), and the genomes of prokaryotes, which are much more compact, and have very short intergenic regions.

Intergenic regions are the parts of the genome that fall between genes. Although they do not code for proteins or other RNA products, some of their parts are involved in the gene expression, while the function of some other parts is not fully understood. Schematically, we can categorize the segments that occur in the intergenic regions as: *pseudogene*, *gene fragments*, *regulatory elements*, and *repeated sequences* (*repeats* for short).

A pseudogene is a dysfunctional relative of a known gene that has lost its protein-coding ability and is no longer expressed in the cell. In the same way that Darwin thought of two species as possibly having a shared common ancestry followed by millions of years of evolutionary divergence, a pseudogene and its associated functional gene also share a common ancestor and have diverged as separate genetic entities over millions of years. Roughly speaking, pseudogenes are considered as the last stop for genomic material that is going to be removed from the genome, hence they are often referred to as junk DNA. Several factors can alter the gene functionality. As we briefly discussed in the previous sections, a gene undergoes several steps in going from a genetic DNA sequence to a fully-functional protein (transcription, pre-mRNA processing, translation, etc). All of these steps are performed by enzymes and proteins that recognize and bind to some specific DNA/mRNA subsequence. For example, during the translation phase, the ribosome starts the translation of the mature mRNA sequence from the *start codon* `AUG`, ending at the *stop codon* `UAG` (or at one of its variants). If the start or the stop codon are missing, because they are not encoded in the gene, the translation process fails. If any of these steps fails, then the sequence may be considered non-functional.

Another example of dysfunctional gene are the so-called gene fragments, that are short segments of existing genes that have been copied from an existing gene to an intergenic region.

A regulatory sequence is a segment of DNA where regulatory proteins such as transcription factors bind preferentially. Regulatory sequences are appropriately positioned in the genome, usually a short distance before the gene being regulated. An example is the `TATA`-box which contains the `TATAAA` DNA sequence or a variant. It is usually located 25 base pairs before the gene transcription site that signals the start of the coding DNA sequence. During the transcription phase it is bound by the `TATA` binding protein, which unwinds the DNA molecule, facilitating the transcription process.

The last type of segments occurring in the intergenic regions is that of repeats, which are repetitive DNA sequences occurring multiple times inside the genome. About 50% of the 3 billion bases of the human genome is composed by repeats [45]. Repeats are usually categorized with respect to their position, size, and content in two main categories: *tandem repeats*, and *dispersed repeats.*

In the case of tandem repeats, the occurrences of the repeated sequences are adjacent, or separated by very few bases. When between 10 to 60 nucleotides are repeated, the repeat is referred to as a *minisatellite*, smaller repeats are known as *microsatellites* or *short tandem repeats.* The repeats can vary in number depending on the specific tandem repeat, and host individual. In fact, some tandem repeats can occur in multiple "forms" presenting a different number of repeats (*Variable Number Tandem Repeats*). Since intergenic regions are well-conserved across generations, some tandem repeats are used as genetic marker in genealogical DNA tests. Although the role of tandem repeat in intergenic regions has not been fully understood, it has been proved that variation in the number of repetitions is associated with genetic diseases as Huntington's disease and muscular dystrophy [175].

Differently than tandem repeats where the repeated genomic sequences are adjacent in the DNA sequence, in the case of dispersed repetitions the occurrences of the repeats are not located in continuous region of the DNA sequence, but spread the whole chromosome (*chromosome-specific repeats*), or even the whole genome (*interchromosomal repeats*). Genome-wide interspersed repeats are believed to be derived from *transposable elements* (*transposons* for short), who are able to move from one DNA location to another. Transposons are usually classified based upon the mechanism of transposition in DNA-*transposons*, and *retrotransposons.* While DNA-transposons have an autonomous cut-and-paste transposition mechanisms, that does not involve any RNA intermediate molecule, retrotransposons, instead, copy themselves in two stages. First from DNA to RNA by transcription, then from RNA back to DNA by reverse transcription (a process used by some RNA-viruses to transform their RNA genomes into DNA which is then integrated into the host genome and replicated along with it). The DNA copy is then inserted into the genome in a new position. According to their structural properties, retrotransposons are further classified in SINEs (short interspersed nuclear elements) that are about 300 bases long, LINEs (long interspersed nuclear elements) which reaches a size of 6000 bases, and `LTR`-retrotransposon which have the structure described in Figure 4.1 with two `LTR` sequences of about 300 bases, flanking the central sequence.

Nowadays, it is currently believed that repeats (and in particular those originated by transposons), have played a fundamental role in eukaryotes evolution, generating a large number of genetic and phenotypical variations upon which natural selection worked. However, because of the accumulation of mutations, the identification of the repeats is not a simple task. Chapter 2 discusses a new class of approximated motifs (*masks*) that aim at modeling short fixed length repetitions, showing a high conservation of the contents at certain positions, while the content of the other positions do not matter at all.

The problem of transposon detection is the topic of Chapter 4. In this chapter we propose a novel approach to detect transposons, which can benefit from the availability of population genomic datasets containing the genomes of different individuals of the same species. We will assess the precision and the recall of our pattern discovery approach by a case study where we analyzed the dataset of 38 strains of *S. cerevisiæ* recently released in [130].

Chapter 2

Mask Motif Discovery

2.1 Introduction

In this chapter we introduce a new class of *motifs with don't cares*, motivated by sequence analysis in biological data and data mining on sequences. Motifs are repeated patterns, where a pattern is an intermixed sequence of alphabet symbols (*solid* symbols) and special symbols $\circ$ (*don't care* symbols). The don't care symbol, found in a position of the pattern, specifies that the position may contain any alphabet symbol. For example, pattern `a`$\circ$`t`$\circ\circ$`c` repeats twice in the input text sequence $T =$ `aaaattaccccatagt` at positions 2 and 3 (starting from 0), and matches the two corresponding strings `aattac` and `attacc` of T.

Informally, motifs represent frequent patterns, which occur at least q times, for a user defined integer $q \geq 2$ called the *quorum*. Given an input text sequence T of length n, a quorum q, and a motif length L, we consider the problem of *motif discovery*: find the motifs of quorum q and length L in the text T. Each motif may have associated the list of the starting positions of its occurrences in the given sequence T. Unfortunately, due to the don't cares, the number of motifs can be exponentially large for increasing values of L. Potentially, there can be as many as $\Theta\big((|\Sigma|+1)^L\big)$ motifs, where Σ is the alphabet of the distinct symbols in the text T. Even though this number can be smaller for some particular instances, the known algorithms discovering these motifs still require, in the worst case, exponential time and space for increasing values of L.

A lot of research has investigated these issues in order to mitigate the combinatorial explosion of motifs [66, 139, 156, 159, 180]. We follow a new approach based on modeling motifs by using simple binary patterns, called *masks*, that implicitly represent *families of patterns* in T (instead of individual patterns). For example, mask `101001` represents both `a`$\circ$`t`$\circ\circ$`c` and `t`$\circ$`g`$\circ\circ$`a`: each `1` represents a solid symbol, while each `0` represents a don't care symbol. A mask is a *motif* if at least one of its represented patterns occurs q or more times in the given sequence T.

As it should be clear from the above informal definition, we aim at describing interesting repetitions in a sequence, using a succinct description (mask) that gives rise to a smaller set of output motifs. Intuitively, consider some patterns that occur at least q times each and that also share the *same structure*, meant as a certain concatenation of solid and don't care symbols. Since they originate from the same mask, we take this mask as a motif. Moreover, any two patterns sharing the same structure but having a different number of occurrences in T (still at least q in number), which were previously considered as different motifs, are now giving rise to the same motif by our definition of mask. Since each mask can be seen as a binary string, we have potentially 2^L masks to examine instead of $(|\Sigma|+1)^L$ frequent patterns with don't cares.

We study the problem of detecting *maximal masks*, namely, the most specific ones (maximal number of `1`s) such that at least one of its represented patterns occurs q times or more (see Section 2.3 for a formal definition). For example, given the text $T =$ `aaaattaccccatagt`, fixing $L = 4$ and $q = 2$, we obtain the maximal masks `1110`, `0111`, and `1101`. Notice that `1110` and `0111` are equivalent since they originate the same patterns (three consecutive solid symbols) ignoring border effects, so we can treat them as the same mask. Therefore, the patterns that are represented by the maximal masks are `aaa`, `ccc`, and `aa`$\circ$`t`, and so the parameter L can equivalently be read as an upper bound on their length.

Specifically, we intend to solve the following *motif discovery* problem. We are given an input text sequence T over alphabet Σ, an integer length $L \geq 1$, and a quorum $q \geq 2$. We want to infer the set $\mathscr{M}$ of all *motifs* μ such that

1. μ is composed of L bits;

2. at least one of the patterns implicitly represented by μ occurs q or more times in T;

3. μ is *maximal*, namely, flipping any of its `0`s into a `1` violates condition 2 above.

It is worth noting that motif discovery has many applications in the investigation of properties of biological sequences. In such applications, it is a must to allow distinct occurrences of a motif to show some differences. In other words, we actually infer *approximated* motifs. This approximation can be realized in several ways, according to the kind of application one has in mind. Motifs of limited length with don't cares can typically model biological object such as transcription factors binding sites, which are characterized by a short length, and a high conservation of their structure. Also, they present a high conservation of the contents in certain positions while for others it does not matter at all. The don't care symbols of our masks indeed aim at *masking* the latter, while the solid character should *unmask* the former.

Moreover, our masks could also be employed as building blocks for longer and flexible motifs, of different kinds, allowing also indels. In recent years, there has been a growing interest in *seeds* for several applications (preprocessing filtration prior to a multiple alignment, approximate search task, data base search, BLAST like homology search, profile search, probe design) in bioinformatics [68, 100, 119, 120, 118, 177]. Among them, many have focused the attention on *gapped seeds*, or *spaced seeds* [36, 40, 43, 57, 110, 176]. It turns out that gapped seeds can be found using the masks, and thus using the algorithms introduced in this chapter.

Finding motifs with don't cares could help to detect structural similarities, with a suitable input sequence. Possible applications involve the detection of structural motifs in alpha helices, of conserved positions in channeling trans-membrane proteins, and repeated structures that are involved in the DNA folding process. In fact, when investigating the folding of a DNA sequence, it can be interesting to rewrite the sequence itself into the alphabet $\{\mathtt{w}, \mathtt{s}\}$ replacing each `A` and `T` with `w` (weak), and each `C` and `G` with `s` (strong). The motivation is that in the *base pairing* that assists in stabilizing the DNA structures, Adenine (`A`) binds to Thymine (`T`) via two hydrogen bonds, while Cytosine (`C`) forms three hydrogen bonds with Guanine (`G`). Hence, the latter bond is stronger than the former, and this has an influence on the actual structure of the molecule. Here, a motif on such sequence could represent a repeated structure, regardless of the actual DNA bases that form it.

In the following sections we show conceptually how to associate a pruned trie of height L with each mask μ. Since the text positions of the occurrences of the patterns implicitly represented by μ cannot overlap (while the patterns themselves can), we store the corresponding partition of the text positions into the trie, where the positions corresponding to the occurrences of the same pattern share a common leaf.

Our algorithm refers to the above pruned tries for the masks but it does not actually need to store them explicitly. Indeed, it extends the Karp-Miller-Rosenberg doubling scheme [105] and applies it to the masks, of length an increasing sequence of powers of 2 up to L. Our algorithm avoids to actually create the tries and just performs scanning and sorting of some suitable lists of consecutive pairs and triplets of integers.

However, the above method still generates the set $\mathscr{Q}$ of all the (maximal and not) masks having quorum q for the given sequence T, where $\mathscr{Q} \supseteq \mathscr{M}$. A post-processing that filters from $\mathscr{Q}$ the masks that are not maximal, may increase the time complexity: precisely, it may take $\Theta(|\mathscr{Q}|^2 L)$ time in the worst case (e.g. [80]), yielding an additional cost of $\Omega(2^{2L}\,L)$ time.

We therefore introduce the crucial notion of *safe masks*, which includes the maximal masks as a special case. We show how to explore the lattice of 2^L masks of length L by examining only safe masks, so that maximal masks can be efficiently detected. In this way, we avoid the above postprocessing and obtain our final bound of $\mathcal{O}(2^L n)$ time and space in the worst case, for discovering all the masks belonging to $\mathscr{M}$, which is our main result.

In order to compare the time complexity of our proposed algorithm, consider the following scenario. After a preprocessing phase of the text T in polynomial time $\mathcal{O}(n^c)$, for a constant $c \geq 1$, consider the following checking phase: for each of the $(|\Sigma| + 1)^L$ candidate patterns, verify if the given pattern has quorum and is maximal, taking just constant time (which is the best we can hope for, once a candidate pattern is given). An algorithm based on this ideal strategy would cost $\mathcal{O}(n^c + (|\Sigma| + 1)^L)$ time. When the latter is compared to the $\mathcal{O}(2^L n)$ time cost of our algorithm, we observe that $2^L n \leq n^c$ when $L \leq (c-1)\log_2 n$ and that $2^L n \leq (|\Sigma| + 1)^L$ when $L \geq \log_2 n/(\log_2(|\Sigma| + 1) - 1)$. Hence, our cost $\mathcal{O}(2^L)$ is better than the ideal bound $\mathcal{O}(n^c + (|\Sigma| + 1)^L)$ except for few degenerate cases (namely, when $c < 1 + (\log_2(|\Sigma| + 1) - 1)^{-1}$). In general, we can establish an upper bound $2^L n = \mathcal{O}\big(n^{\Theta(1+1/\log_2|\Sigma|)} + \min\{2^L n, (|\Sigma| + 1)^L\}\big)$. In other terms, our algorithm performs better than virtually *constant-time enumerating and checking* all the potential $(|\Sigma| + 1)^L$ candidate patterns in T. In the above discussion for the complexity, we assume that the word size of w bits in the standard RAM is sufficiently large, so that $L = \mathcal{O}(w)$. When L is much larger, the time complexity of our algorithm must be multiplied by a factor of $\mathcal{O}(L/w)$.

This chapter is organized as follows. After reviewing the state of the art in Section 2.2, in Section 2.3 we give a formal definition of our motifs based on masks. We then present efficient algorithms to compute these motifs, reviewing also basic notions as that of maximality in the light of this new class of motifs. More precisely, in Section 2.4, we show how to discover the motifs of length L in $\mathcal{O}(2^L n)$ time by extending the Karp-Miller-Rosenberg approach to the masks. In Section 2.5, we represent the space of all possible 2^L masks of length L as a lattice and introduce the notion of safe masks. We also describe how to traverse implicitly this lattice for discovering maximal masks in it, querying the oracle only for safe masks. Finally, we draw our conclusions in Section 2.6.

2.2 Related Problems and State of the Art

We are not aware of any previous work introducing our class of motifs. Hence, we relate our results in motif discovery to those of mining frequent itemsets, where more sophisticated techniques have been found over the years [4, 5, 81]. The notion of masks comes naturally into play when performing data mining for frequent itemsets, where the "apriori" algorithm is intensively employed [91]. Here, a set of L items is given, and each transaction (basket) corresponds to a subset of these items, which can be represented as a binary sequence in which the ith symbol is `1` if and only if the ith item is chosen for the basket. A set of baskets can be therefore represented as a set of masks in our terminology. For the lattice of all possible 2^L masks, all possible itemsets should be examined. Note that, instead, our definition of masks has the goal of condensing patterns that have the same sequence of solid and don't care symbols. Moreover, our traversal of the lattice is different from the apriori algorithm, since we start from the top and generate candidates in a different way, namely, using safe masks (see Section 2.5 for details).

As far as we know, the "dualize and advance" algorithm [86, 87] is the best theoretical approach that can be obtained in terms of running time. It sets up an interesting connection between mining itemsets in the lattice of 2^L masks and finding hypergraph traversals [21]. In our terminology, suppose to have incrementally found some of the maximal masks, say $\mu_1, \mu_2, \ldots, \mu_k$. We build the corresponding hypergraph as follows: there are L nodes numbered from 1 to L, and there is one hyperedge per mask, where the jth bit in the mask is $\texttt{0}$ if and only if the node j is incident to the corresponding hyperedge ($1 \leq j \leq L$). In general, the ith hyperedge connects the nodes that correspond to the $\texttt{0}$s in the ith mask μ_i ($1 \leq i \leq k$). In order to find additional maximal masks (and hence add hyperedges), it suffices to find all the hypergraph traversals as starting points for upward paths in the lattice, where each traversal is a minimal hitting set for the current set of k hyperedges [21].

The problem of finding hypergraph traversals is intimately related to the dualization of monotone Boolean functions [62]. The known algorithms required $\mathcal{O}(2^L)$ time in the worst case [21, 108] until the seminal result in [72, 113] showing a subexponential bound proportional to $t(k) = k^{\mathcal{O}(\log k)}$ time, when the number of hyperedges k is $o(2^L)$. This algorithm is plugged into the scheme of the "dualize and advance" algorithm, giving a bound of $\mathcal{O}\left(n \times t(|\mathscr{M}| + |Bd^-(\mathscr{M})|)\right)$ as shown in [86], where we include the cost $\mathcal{O}(n)$ of verifying the quorum, and $Bd^-(\mathscr{M})$ is the set of masks not satisfying the quorum constraint, such that all the predecessors of the masks satisfy the quorum, and all the successor do not. While $|\mathscr{M}|$ can be subexponential, there are cases in which $|\mathscr{M}| + |Bd^-(\mathscr{M})| = \Theta(2^L)$ [86], and so the final bound can be $\Omega(2^{L^2} n)$.

Surprisingly, this and other approaches based on hypergraph traversals, which are the state of the art theoretically, are slower than our solution in the worst case.

2.3 A New Class of Motifs

In this section, we introduce our new class of motifs with don't cares. Starting from some basic notions, we describe some features of this class that will then be exploited by the algorithms in the rest of this chapter. Let T be an input text of size n drawn over the alphabet Σ. The sequence T can be seen as an array $T[0 \ldots n-1]$ of symbols, where symbol $T[i] \in \Sigma$ is stored into position i, for $0 \leq i \leq n-1$. A substring $T[i]T[i+1] \cdots T[j]$ is represented as $T[i \ldots j]$.

2.3.1 Masks and patterns

Given a positive integer L, we call *mask* any binary sequence in $\{\texttt{0}, \texttt{1}\}^L$; hence, L is the length of the mask. For a given mask $\mu = \mu[0 \ldots L-1]$, we define $S_\mu = \{i \mid \mu[i] = \texttt{1}, \text{for } 0 \leq i \leq L-1\}$ as the set of its *solid positions*, and $D_\mu = \{i \mid \mu[i] = \texttt{0}, \text{for } 0 \leq i \leq L-1\}$ as the set of its *don't care positions.* For example, mask $\mu = \texttt{1010}$ has $S_\mu = \{0, 2\}$ and $D_\mu = \{1, 3\}$.

A *pattern* is a regular expression $m \in (\Sigma \cup \{\circ\})^L$, where $\circ$ is the *don't care symbol* that matches any symbol in Σ. We say that a pattern m is an *instance* of a mask μ when, for each position $0 \leq k \leq L-1$, it is $m[k] = \circ$ if and only if $\mu[k] = \mathtt{0}$. For example, given $\Sigma = \{\mathtt{a}, \mathtt{c}\}$, the mask $\mu = \mathtt{1010}$ has four instances, namely, $m_1 = \mathtt{a}\circ\mathtt{a}\circ$, $m_2 = \mathtt{c}\circ\mathtt{a}\circ$, $m_3 = \mathtt{a}\circ\mathtt{c}\circ$, and $m_4 = \mathtt{c}\circ\mathtt{c}\circ$.

By exploiting the fact that a mask μ implicitly represents the patterns that are its instances, we define a new relation among the text substrings according to μ, as follows.

Definition 1 ($\equiv_\mu$ relation). *Given a mask μ of length L, a text T of length n, and two text positions $0 \leq i, j \leq n-L+1$, we say that $i \equiv_\mu j$ if and only if $T[i+k] = T[j+k]$ for each solid position $k \in S_\mu$ in the mask.*

Definition 1 relates any two text substrings of length L, appearing at positions i and j, when these substrings match in each solid position specified by the mask μ. For example, given $T[i \ldots i{+}L{-}1] = \mathtt{actact}$ and $T[j \ldots j{+}L{-}1] = \mathtt{agttct}$, consider the two masks $\mu = \mathtt{101011}$ and $\mu' = \mathtt{110111}$. It holds $i \equiv_\mu j$ because $\mathtt{a}\circ\mathtt{t}\circ\mathtt{ct}$ occurs both at positions i and j of T, while $i \not\equiv_{\mu'} j$ because the two substrings $\mathtt{actact}$ and $\mathtt{agttct}$ mismatch at solid positions $1, 3 \in S_{\mu'}$.

It is easy to see that the relation $\equiv_\mu$ is an equivalence relation. Therefore, for a given mask μ, the relation $\equiv_\mu$ induces a *partition* of the first $n-L+1$ positions in T. Namely, for each equivalence class C in the partition, we have that $i, j \in C$ if and only if $i \equiv_\mu j$. Hence, each text position i (for $0 \leq i \leq n-L+1$) belongs to exactly one equivalence class. We denote the partition resulting from μ by π_μ. We use $|\pi_\mu|$ to indicate the number of equivalence classes in it, and $|C|$ to indicate the number of elements in a class $C \in \pi_\mu$.

Given an equivalence class C of a partition π_μ, we can associate a pattern m_C of length L with C. Specifically, symbol $m_C[k] = \circ$ if k is a don't care position of the mask μ ($k \in D_\mu$) while $m_C[k] = T[i+k]$ if k is a solid position ($k \in S_\mu$ and for any arbitrary $i \in C$).

In order to better illustrate the properties of the partition π_μ and the corresponding patterns m_C where $C \in \pi_\mu$, we can use a trie (digital search tree [115]) built on the set of strings $\{m_C \mid C \in \pi_\mu\}$. In this trie, the special symbol $\circ$ can be treated as an ordinary symbol, since all the patterns share the same mask μ. We can arrange the strings in this way since the arcs found on the same level of the trie are either all labeled with a solid symbol or all labeled with a don't care symbol. Each root-to-leaf path spells out a distinct pattern m_C, and the corresponding leaf stores the text positions in class C of the partition π_μ.

Example 1. Consider the alphabet $\Sigma_{DNA} = \{\mathtt{a}, \mathtt{t}, \mathtt{c}, \mathtt{g}\}$ and the text $T = \mathtt{aaaattaccccatagt}$ of length $n = 16$. For a mask $\mu = \mathtt{1100}$ of length $L = 4$, the partition induced by μ is $\pi_\mu = \{C_0 C_1 \cdots C_6\}$, where $C_0 = \{4\}$, $C_1 = \{5, 12\}$, ..., $C_6 = \{0, 1, 2\}$ are the classes labeling the leaves of the trie shown in Figure 2.1. The instances of μ are the patterns $m_{C_0} = \mathtt{tt}\circ\circ$, $m_{C_1} = \mathtt{ta}\circ\circ$, ..., $m_{C_6} = \mathtt{aa}\circ\circ$.

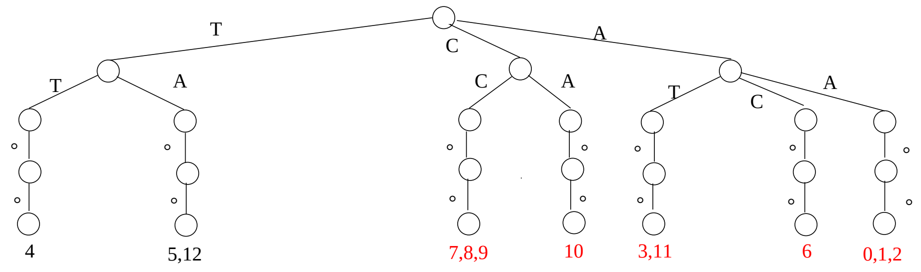


Figure 2.1: Trie for $\equiv_\mu$ where $\mu = \texttt{1100}$.

2.3.2 Partial order of masks and maximality

We now draw our attention to the masks μ that have the maximum number of solid symbols while inducing a partition π_μ which contains at least one class C such that $|C| \geq q$ for the given *quorum* q. For this, we need to introduce a partial order on the masks.

Definition 2 ($\preceq$ relation)**.** *Given two masks μ and μ' of length L, we say that μ is* less specific *than μ' (denoted by $\mu \preceq \mu'$) if and only if $\mu[i] \leq \mu'[i]$ for $0 \leq i \leq L-1$.*

When Definition 2 holds, we also say that μ' is *more specific* than μ, and that μ is a predecessor of μ', and μ' is a successor of μ. For example, $\texttt{0001} \preceq \texttt{1101}$, while $\texttt{0001} \not\preceq \texttt{0010}$. The mask μ is an *immediate predecessor* of μ' if $\mu \preceq \mu'$ and they differ in exactly one symbol (e.g. $\texttt{1001}$ and $\texttt{1101}$). We can define the immediate successor analogously.

Relation $\preceq$ is a partial order among the masks because it is reflexive, antisymmetric ($\texttt{1001} \preceq \texttt{1101}$, but $\texttt{1101} \not\preceq \texttt{1001}$) and transitive. Hence, it gives rise to the partially ordered set $\mathscr{L} = \langle \{\texttt{1},\texttt{0}\}^L, \preceq \rangle$, which is a finite lattice of 2^L masks. The top mask of $\mathscr{L}$ is $\texttt{1}\ldots\texttt{1}$ and the bottom mask is $\texttt{0}\ldots\texttt{0}$. The lattice $\mathscr{L}$ is isomorphic to the power set lattice $\mathscr{P} = \langle \mathcal{P}(\{0,\ldots,L-1\}), \subseteq \rangle$ because each mask represents the characteristic vector of a set in the powerset $\mathcal{P}(\{0,\ldots,L-1\})$, and $\mu \preceq \mu'$ if and only if $S_\mu \subseteq S_{\mu'}$.

Analogously, we can define the $\preceq$ relation between patterns. Given two patterns m and m' of length L, we say that m is *less specific* than m' (written $m \preceq m'$) if and only if either $m[i] = m'[i]$ or $m[i] = \circ$ for $0 \leq i \leq L-1$. For example, $\circ\circ\circ\texttt{a} \preceq \texttt{at}\circ\texttt{a}$ while $\circ\circ\circ\texttt{a} \not\preceq \texttt{at}\circ\texttt{c}$ and $\circ\texttt{c}\circ\texttt{a} \not\preceq \texttt{a}\circ\circ\texttt{a}$. Note that $\preceq$ is a partial order also for the patterns. However, it gives rise to a lattice of $(|\Sigma|+1)^L$ patterns, and so in the rest of this chapter we prefer to adopt the binary lattice $\mathscr{L}$ of 2^L masks.

At this point, we may wonder what is the connection between the partitions induced by the masks (Section 2.3.1) and the lattice $\mathscr{L}$ formed by the masks.

Lemma 1. *For any two masks μ and μ' such that $\mu \preceq \mu'$, $\pi_{\mu'}$ is a refinement of π_μ. Namely:*

(i) For each class $C \in \pi_\mu$, there exists classes $C_0, C_1, \ldots, C_{s-1} \in \pi_{\mu'}$ which form a partition of C.

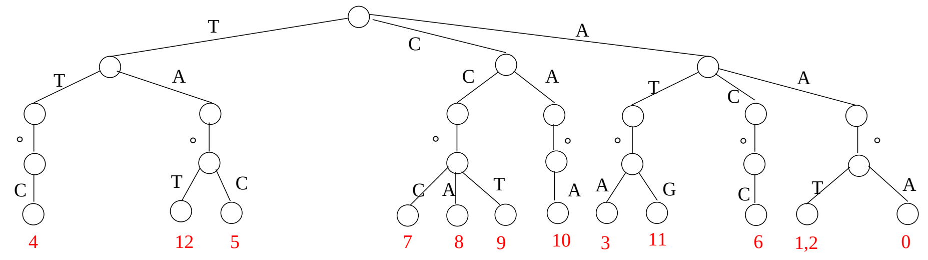


Figure 2.2: Trie for $\equiv_{\mu'}$ where $\mu' =$ `1101`.

(ii) For each class $C' \in \pi_{\mu'}$, there exists a class $C \in \pi_{\mu}$ such that $C' \subseteq C$.

Example 2 (continued)**.** Consider mask $\mu' =$ `1101`, for which $\mu \preceq \mu'$, where $\mu =$ `1100`. Consider the equivalence classes of π_μ (Figure 2.1) and $\pi_{\mu'}$ (Figure 2.2). We have that $C_0 = \{4\} = C'_0$, $C_1 = \{5, 12\} = \{12\} \cup \{5\} = C'_1 \cup C'_2$, $C_2 = \{7, 8, 9\} = \{7\} \cup \{8\} \cup \{9\} = C'_3 \cup C'_4 \cup C'_5$, $C_3 = \{10\} = C'_6$, $C_4 = \{3, 11\} = \{3\} \cup \{11\} = C'_7 \cup C'_8$, $C_5 = \{6\} = C'_9$, and $C_7 = \{0, 1, 2\} = \{1, 2\} \cup \{0\} = C'_{10} \cup C'_{11}$. Therefore, both properties (i) and (ii) hold.

2.3.3 Maximal Masks Problem (MMP)

We are given a text T of length n and a length $L \geq 1$ for the masks, along with a quorum $q \geq 2$. We say that a mask μ has *quorum* if there is an equivalence class C in the partition π_μ such that $|C| \geq q$. The relation $\preceq$ and the quorum q are the key ingredients to find interesting masks.

Let $\mathscr{Q}(L, T, q)$ be the set of all masks of length L that have quorum q. A mask μ is *maximal* if it has quorum and no other mask μ' of the same length has quorum and is more specific than μ (i.e. $\mu \preceq \mu'$). We denote by $\mathscr{M}(L, T, q) \subseteq \mathscr{Q}(L, T, q)$ the set of all maximal masks ($\mathscr{M}$ and $\mathscr{Q}$ for short, respectively), and address the following problem in this chapter.

Problem 1 (MMP= Maximal Masks Problem)**.** *Input: a mask length L, a text T, and a quorum q. Output: the set $\mathscr{M} \equiv \mathscr{M}(L, T, q)$ of all the maximal masks.*

Example 3 (continued)**.** Using $T =$ `aaaattaccccatagt`, for $L = 4$ and $q = 2$, the set of masks with quorum is $\mathscr{Q}(4, T, 2) = \{$`0000`, `0001`, `0010`, `0011`, `0100`, `0101`, `0110`, `0111`, `1000`, `1001`, `1010`, `1100`, `1101`, `1110`$\}$, while the maximal masks are only those of the set $\mathscr{M}(4, T, 2) = \{$`0111`, `1101`, `1110`$\}$. The patterns that are instances of the maximal masks are ∘`aaa`, `aa`∘`t`, and ∘`ccc`. Notice that masks `0111` and `1110`, which are both in $\mathscr{M}$, actually represent the same patterns, except possibly border effects, because one is the shift of the other obtained by only changing the number of ∘s at the sides. We will show in Section 2.4 how to remove this sort of redundancy, which will actually obtain a reduction of the search space, as a positive side effect.

We can see that checking whether a mask μ has quorum q corresponds to evaluating a Boolean predicate $P_T(\mu)$ which returns true if and only if there exists a pattern m that is an instance of μ and that has at least q matches in T. Note that $P_T(\mu)$ is anti-monotone.[1] Hence, MMP can be equivalently restated as checking an anti-monotone predicate $P_T(\mu)$ on a binary lattice of 2^L masks, to discover all the maximal masks where the predicate holds. In the following sections, we will describe how to solve this equivalent problem for sequences, and we will make use of this view of MMP.

2.4 The KMR Approach for Masks with Quorum

We now describe our first algorithm to solve MMP by building the sets $\mathscr{Q}$ and $\mathscr{M}$. Conceptually, we want to build the partition π_μ for each mask $\mu \in \{\mathtt{0}, \mathtt{1}\}^L$, check whether μ has quorum (i.e. there exists a class $C \in \pi_\mu$ such that $|C| \geq q$), and verify that no mask μ' with $\mu \preceq \mu'$ has quorum. Note that a straightforward way of checking whether a mask has quorum requires to scan the text T and build a trie like that shown in Figure 2.2, in $\Theta(Ln)$ time. This would give a total cost of $\Omega(L2^Ln)$ time since there are $\Omega(2^L)$ masks to check.

We reduce the above cost to $\mathcal{O}(2^Ln)$ time using a different and simple approach that avoids explicitly building the trie for each mask. Our idea is to maintain the partitions induced by the masks as follows. Assuming without loss of generality that L is a power of 2, we first compute the partitions induced by the masks of length 1; inductively, given the partitions induced by the masks of length 2^i, we show how to compute the partitions induced by the masks of length 2^{i+1} in a way that does not explicitly need the tries, even though we will implicitly refer to them during the description of our algorithm.

We implement our idea using the approach proposed by Karp, Miller and Rosenberg [105], hereafter called KMR. The KMR approach addresses the problem of identifying exact repeated substructures of fixed size in a given combinatorial structure. It applies to finding repeated substrings in strings, repeated subtrees in trees, and repeated segments in arrays [48]. For strings, KMR uses a relation E_k according to which two substrings of length k beginning at positions i and j of the text T are *k-equivalent*, written $i\, E_k\, j$, if and only if they are identical in every position. Given this, KMR provides a characterization of $E_{k+k'}$ in terms of E_k and $E_{k'}$, so that it constructs inductively the sets $\{E_2, E_4, E_8, \ldots, E_L\}$ by setting $k = k'$. That is, it doubles the length of the substrings by means of a concatenation of two substrings from the previous iteration. KMR starts out from the set E_1 (obtained by a simple scan of the input sequence T), and ends at the required length L. Each iteration takes time $\mathcal{O}(n)$, where n is the length of the text T. Since the number of iterations

[1]We recall that, given a partial order $\preceq$, a predicate p is *anti-monotone* if, for any x and y such that $x \preceq y$, we have that $p(y) = \mathit{true}$ implies $p(x) = \mathit{true}$. Conversely, p is *monotone* if $p(x) = \mathit{true}$ implies $p(y) = \mathit{true}$.

is $\mathcal{O}(\log L)$, the overall complexity of KMR is $\mathcal{O}(n \log L)$ time.

In our case, MMP differs from the problem solved by KMR since we use masks as one further level of abstraction. We do not apply KMR directly to the text substrings; instead, we double the length of the masks and, as a side effect, we double the length of the induced equivalent substrings in the text (i.e. they match on all the solid positions of the given mask). In this way, the original KMR can be seen as a special case of our approach when the mask is made by all solid symbols $\texttt{11}\cdots\texttt{1}$.

We observe that, for any given mask, we can employ KMR when the relation E_k is replaced by the $\equiv_\mu$ relation introduced in Definition 1. We also replace the concatenation performed by KMR at each iteration by the concatenation operation among masks. Given two masks μ and μ', we indicate their concatenation by $\mu\mu'$. The following result relates the $\equiv_\mu$ equivalence relation to the mask concatenation operation, showing how the KMR paradigm can be generalized to our case.

Lemma 2. *Given a string T of length n, two masks μ, μ', two positions i, j in T then $i \equiv_{\mu\mu'} j$ if and only if $i \equiv_\mu j$ and $(i + |\mu|) \equiv_{\mu'} (j + |\mu|)$.*

Proof. We start by showing how $i \equiv_{\mu\mu'} j$ implies $i \equiv_\mu j$ and $(i + |\mu|) \equiv_{\mu'} (j + |\mu|)$. If $i \equiv_{\mu\mu'} j$, then by definition we have that

$$T[i + k] = T[j + k] \text{ for all } k \in S_{\mu\mu'}. \tag{2.1}$$

Notice that $S_{\mu\mu'} = S_\mu \cup (S_{\mu'} + |\mu|)$. Hence, (2.1) implies that (i) $T[i+k] = T[j+k]$ for all $k \in S_\mu$, and (ii) $T[i + k] = T[j + k]$ for all $k \in (S'_\mu + |\mu|)$. Observe that (i) exactly matches the definition of $i \equiv_\mu j$, which is then proved. On the other hand, (ii) implies that $T[i+k] = T[j+k]$ holds for $k = |\mu|+k'$ for all $k' \in S_{\mu'}$, that is to say $T[i + |\mu| + k'] = T[j + |\mu| + k']$ for all $k' \in S_{\mu'}$, and hence that $(i + |\mu|) \equiv_{\mu'} (j + |\mu|)$. In order to show that if $i \equiv_\mu j$ and $(i + |\mu|) \equiv_{\mu'} (j + |\mu|)$, then $i \equiv_{\mu\mu'} j$, it is enough to observe that all steps above can be inverted, and hence the result is proved. □

2.4.1 Partition construction and generation of masks

Using Lemma 2 requires an efficient procedure that computes all the related equivalence classes in a partition. In this section we describe an algorithm that solves this problem.

Given a quorum $q \geq 2$ and two partitions π_μ and $\pi_{\mu'}$, where μ is not necessarily different from μ', the algorithm returns a new partition $\pi_{\mu\mu'}$ built according to Lemma 2, possibly filtered in order to satisfy the quorum constraint. The key point is illustrated in Figure 2.3, where we are given two partitions π_μ and $\pi_{\mu'}$ and we want to obtain the new partition $\pi_{\mu\mu'}$ shown in Figure 2.2. (Note that the text is the same as before $T = \texttt{aaaattaccccatagt}$, that $q = 2$, and that the tries are shown for the sake of presentation since we do not actually employ them in our implementation.) In order to concatenate two masks μ and μ' of length ℓ, the main

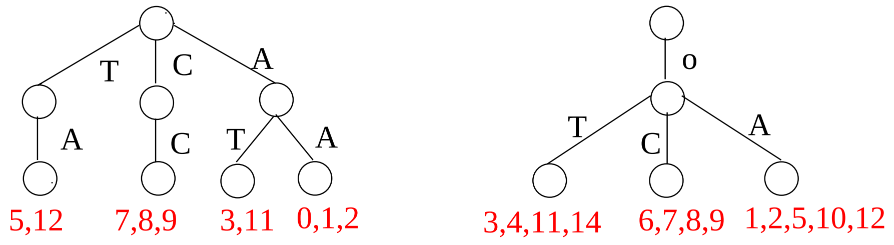


Figure 2.3: Partitions for masks $\mu = \texttt{11}$ (left) and $\mu' = \texttt{01}$ (right).

steps can be summarized as follows (we refer to Figure 2.2 and Figure 2.3 as an example).

1. We are given masks μ and μ' and their induced partitions π_μ and $\pi_{\mu'}$. We consider only the equivalence classes C such that $|C| \geq q$, and number these classes so that each class has its own *class name* inside its partition. (In our example, $\pi_\mu = [\{5,12\}_0, \{7,8,9\}_1, \{3,11\}_2, \{0,1,2\}_3]$ and $\pi_{\mu'} = [\{3,4,11,14\}_0, \{6,7,8,9\}_1, \{1,2,5,10,12\}_2]$, where we ignore classes with less than q elements and report the numbering of each relevant class as its subscript.)

2. We create a (multiset) list LP of pairs as follows. First, for each class $C \in \pi_\mu$, we add the pairs $\langle i, n_C \rangle$ to LP for all positions $i \in C$, where n_C is the number assigned to C in step 1. Second, for each class $C' \in \pi_{\mu'}$, we add the pairs $\langle i' - |\mu|, n_{C'} \rangle$ to LP for all positions $i' \in C'$ such that $i' \geq |\mu'|$, where $n_{C'}$ is the number assigned to C' in step 1. (In our example, we obtain $LP = [\langle 5,0 \rangle$, $\langle 12,0 \rangle$, $\langle 7,1 \rangle$, $\langle 8,1 \rangle$, $\langle 9,1 \rangle$, $\langle 3,2 \rangle$, $\langle 11,2 \rangle$, $\langle 0,3 \rangle$, $\langle 1,3 \rangle$, $\langle 2,3 \rangle$, $\langle 1,0 \rangle$, $\langle 2,0 \rangle$, $\langle 9,0 \rangle$, $\langle 12,0 \rangle$, $\langle 4,1 \rangle$, $\langle 5,1 \rangle$, $\langle 6,1 \rangle$, $\langle 7,1 \rangle$, $\langle 3,2 \rangle$, $\langle 8,2 \rangle$, $\langle 10,2 \rangle]$ since $|\mu'| = 2$.)

3. We sort the list LP in a *stable* way according to the first component of each pair in it. We drop from the list the pairs $\langle i, j \rangle$ such that no other pair has i as its first component in the list. (We obtain $LP = [\langle 1,3 \rangle$, $\langle 1,0 \rangle$, $\langle 2,3 \rangle$, $\langle 2,0 \rangle$, $\langle 3,2 \rangle$, $\langle 3,2 \rangle$, $\langle 5,0 \rangle$, $\langle 5,1 \rangle$, $\langle 7,1 \rangle$, $\langle 7,1 \rangle$, $\langle 8,1 \rangle$, $\langle 8,2 \rangle$, $\langle 9,1 \rangle$, $\langle 9,0 \rangle$, $\langle 12,0 \rangle$, $\langle 12,0 \rangle]$.)

4. The actual concatenation between μ and μ' takes place. Indeed, there is an occurrence of a pattern m (instance of mask $\mu\mu'$) in position i if and only if $\langle i, j \rangle$ and $\langle i, j' \rangle$ are consecutive pairs in LP for some $0 \leq j, j' \leq n - 1$. Hence, we generate a triplet $\langle j, j', i \rangle$ from these two pairs (note that $\langle i, j \rangle$ precedes $\langle i, j' \rangle$ in LP), thus forming a list LT of triplets. (We obtain $LT = [\langle 3,0,1 \rangle$, $\langle 3,0,2 \rangle$, $\langle 2,2,3 \rangle$, $\langle 0,1,5 \rangle$, $\langle 1,1,7 \rangle$, $\langle 1,2,8 \rangle$, $\langle 1,0,9 \rangle$ $\langle 0,0,12 \rangle]$.)

5. We lexicographically sort the list LT according to the first two components of each triplet in it. We drop from the list the triplets $\langle j, j', i \rangle$ such that there are less than q triplets in the list having j and j' as their first component in the list, since they do not reach the quorum. (We obtain $LT = [\langle 3,0,1 \rangle$, $\langle 3,0,2 \rangle]$.)

6. We start from an empty partition $\pi_{\mu\mu'}$. For each maximal run of consecutive triplets $\langle i, j, k_1\rangle$, $\langle i, j, k_2\rangle$, $\ldots\langle i, j, k_r\rangle$ in LT ($r \geq q$), we add the class $\{k_1, k_2, \ldots, k_r\}$ to $\pi_{\mu\mu'}$. We return $\pi_{\mu\mu'}$ after completing the scan of LT. (In our example, $\pi_{\mu\mu'} = [\{1, 2\}]$ since only one class contains at least q elements in Figure 2.2.)

For the sake of simplicity, we described steps 3 and 5 as sorting steps, but it suffices a stable grouping of the input, meaning that pairs (triplets) having the same first (two) component(s) should be consecutive in the resulting list.

Lemma 3. *Given partitions π_μ and $\pi_{\mu'}$ and a quorum threshold $q \geq 2$, steps 1–6 correctly compute the set $\{C \in \pi_{\mu\mu'} \mid |C| \geq q\}$ in $\mathcal{O}(n)$ time and space.*

Proof. We begin by proving the method to be correct. Given two partitions π_μ and $\pi_{\mu'}$, in step 1, we label each class C of theirs, by a distinct class name n_C, while in step 2 we rewrite each occurrence of a μ's instance (i.e. a text position i in a class $C \in \pi_\mu$) as a pair $\langle i, n_C\rangle$ and each occurrence j' of an instance of μ' as a pair $\langle j, n_{C'}\rangle$ where $j = j' - |\mu|$. In this way, given two pairs $\langle i, n_C\rangle$ and $\langle j, n_{C'}\rangle$, if i is equal to j then $i \equiv_\mu j$ (since i belongs to an equivalence class in π_μ) and $(i + |\mu|) \equiv_{\mu'} (j + |\mu|)$ (since j' belongs to an equivalence class in $\pi_{\mu'}$). In order to detect pairs having the same first component, in step 3 we sort them in a stable way, discarding all pairs that do not share the first value with any other. At step 4 for each couple of consecutive pairs $\langle i, n_C\rangle$, $\langle i, n_{C'}\rangle$ the triplets $\langle n_C, n_{C'}, i\rangle$ representing the text position i of a new equivalence class in $\pi_{\mu\mu'}$ is created. At this point, all the new classes with strictly less than q text positions are discarded and the partition $\pi_{\mu\mu'}$ containing the remaining classes is finally returned. Detecting equivalence classes having more than q text positions is easy by the sorting of step 5: triplets representing text positions in the same class $C'' \in \pi_{\mu\mu'}$ are now grouped as they agree on their first two components n_C and $n_{C'}$.

We prove that steps 1–6 take $\mathcal{O}(n)$ time and space. Since there can be at most $\mathcal{O}(n)$ positions, the list generated at step 1 has size in $\mathcal{O}(n)$. Its sorting in step 2 can be done in $\mathcal{O}(n)$, for example, using radix sort since the integers are small. After sorting, the detection of pairs to be dropped at step 2, as well as that of triplets to be generated at step 3 can also be done in linear time, because pairs that start with the same position are now consecutive. Each newly generated list is either a permutation of the previous one, or even a subset of it, and thus the size remains in $\mathcal{O}(n)$. Therefore, the sorting of step 5 can be done in linear time using radix sort because the number of distinct classes cannot be larger than n. This sorting allows us to detect in linear time the triplets to be dropped at the same step. Similarly, it permits the final detection of maximal runs to be done in linear time as well. Therefore, the overall time and space complexity is $\mathcal{O}(n)$. □

Notice that the elimination, at each iteration, of masks that do not satisfy the quorum actually results in a practical important reduction of the search space. Notice also that if we build the classes of a new mask obtained by the overlapping of

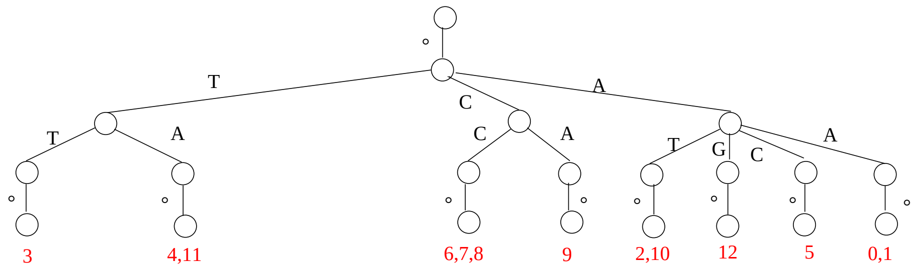


Figure 2.4: Trie for $\pi_{\mu'}$ where $\mu' = \texttt{0110}$.

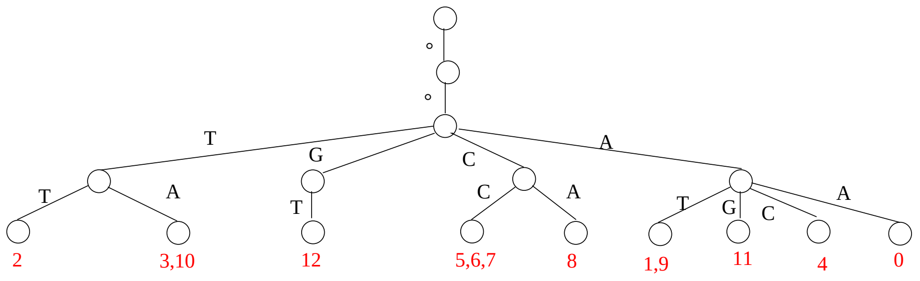


Figure 2.5: Trie for $\pi_{\mu''}$ where $\mu'' = \texttt{0011}$.

two shorter masks (rather than their concatenation), then the very same procedure can be applied (having the same complexity) with the only difference that Step 2 should build pairs $\langle i' - \delta, n_{C'} \rangle$ instead of $\langle i' - |\mu|, n_{C'} \rangle$, where δ is the size of the overlap.

2.4.2 Equivalent masks

We now list some interesting properties that allow us to generate half of all possible 2^L masks. Although these properties do not improve over the worst-case complexity, they are useful optimizations for the practical behavior of our algorithms.

We first need to introduce some notation. Intuitively, consider the partitions shown in Figures 2.1, 2.4, and 2.5, respectively, for masks $\mu = \texttt{1100}$, $\mu' = \texttt{0110}$, and $\mu'' = \texttt{0011}$. The classes in these partitions are reported in Table 2.1, so that their mutual dependence is highlighted. Ignoring border effects in the first and the last $L-1$ positions of the text, we can say that μ, μ', and μ'' conceptually represent the same set of patterns: those that have only two solid and adjacent symbols. Each row of Table 2.1 puts the classes of different partitions into a one-to-one correspondence, in which a class can be obtained from another by adding an integer d to the positions, such that $|d|$ is exactly the amount of shifted symbols in their masks needed to make them equal. For example, class $\{5, 6, 7\}$ can be obtained from $\{7, 8, 9\}$ by adding the integer -2 to the positions of the latter, since μ'' can be transformed into μ shifting its symbols by 2 positions to the left.

Given two masks μ and μ' of the same length, we say that they are *equivalent* if

Figure 2.1	*Figure* 2.4	*Figure* 2.5
{4}	{3}	{2}
{5,12}	{4,11}	{3,10}
-	-	{12}
{7,8,9}	{6,7,8}	{5,6,7}
{10}	{9}	{8}
{3,11}	{2,10}	{1,9}
-	{12}	{11}
{6}	{5}	{4}
{0,1,2}	{0,1}	{0}

Table 2.1: The partitions shown in Figures 2.1, 2.4, and 2.5.

they have the same number of `1`s and μ can be obtained from μ' by a shift of the symbols, as long as only `0`s exceed the mask border and the empty positions are filled with `0`s (also the inverse holds). We define the following notations for sets of positions. Given two sets C and C' of text positions in $[0 \ldots n-1]$, we say that $C \equiv_d C'$ for an integer d if two conditions hold:

1. for each $i' \in C'$ such that $0 \le i' + d \le n-1$, we have that $i' + d \in C$;
2. for each $i \in C$ such that $0 \le i - d \le n-1$, we have that $i - d \in C'$.

It is easy to see that, removing border effects, there is a one-to-one correspondence between the classes of the partitions induced by two equivalent masks that actually coincides with the relation $\equiv_d$, where d is the size of the shift. In other words, the following result clearly holds.

Lemma 4. *For any two equivalent masks μ and μ', there exists an integer d inducing a one-to-one correspondence between the classes of the partitions π_μ and $\pi_{\mu'}$ as follows: for each class $C \in \pi_\mu$, we have a unique class $C' \in \pi_{\mu'}$ such that $C \equiv_d C'$ (and vice versa).*

It is worth noting that, ignoring the first and the last $L-1$ text positions, π_μ and $\pi_{\mu'}$ have the same number of equivalence classes, the same number of elements, and any position shifted by an integer $d > 0$. Alternatively, we can extend the text with $L-1$ endmarker symbols to its left and its right. In this way, Lemma 4 makes partitions $\pi_{\mu'}$ redundant with respect to π_μ.

Consequently, for each group of equivalent masks, we choose as representative the one having `1` as its first symbol. (Such a mask always exists except for the mask `00`$\cdots$`0`, which forms a trivial singleton group whose partition contains just one class made up of all the text positions.) We then eliminate the other masks in the class from our computation (steps 1–6), since we can always recover their partitions by adding a suitable integer d. For example, with $L = 4$, we now build explicitly only the representative masks when applying Lemma 3, which illustrates the fact that the number of representatives is *at most half* the number of equivalent masks: only those that start with a `1`. The masks of length $\ell = 1, 2, 4$ examined are those in the sets F_ℓ below:

$F_1 = \{\texttt{1}\}$,

$F_2 = \{\texttt{11}, \texttt{10}\}$,

$F_4 = \{\texttt{1111}, \texttt{1110}, \texttt{1101}, \texttt{1100}, \texttt{1011}, \texttt{1010}, \texttt{1001}, \texttt{1000}\}$.

2.4.3 Algorithm KMR for masks

We now have all the ingredients for describing our algorithm that applies KMR to solve MMP. Given a string T, a length L and a quorum q, we first compute the partition for $F_1 = \{\texttt{1}\}$ (since that for the mask `0` is trivial). Next, we compute F_2, F_4 and so on, as described in Sections 2.4.1–2.4.2. In particular, we just compute the representative for each equivalent class of masks, when running steps 1–6: for each pair of (not necessarily distinct) masks $\mu, \mu' \in F_\ell$, we build $F_{2\ell}$ by processing the concatenation of μ with all possible shifts of μ' (whose partitions we can quickly recover from that of μ'). Summing up, we actually perform the following two steps:

1. Scan T to construct the partition induced by relations $\equiv_{\texttt{1}}$ and build F_1.

2. Use Lemmas 2–4 to construct, successively, $F_2, F_4, F_8, \ldots, F_{2^r}$, and F_L, where r is the largest value such that $2^r < L$.

Theorem 5. *Using the* KMR *approach, we can build the set $\mathscr{Q}(L, T, q)$ of masks for MMP, along with their induced partitions, in $\mathcal{O}(2^L n)$ time and space.*

Proof. Given a length L, we have at most $\sum_{\ell=1}^{r} 2^\ell + 2^L < 2^{L+1}$ different masks μ to consider, and so as many equivalence relations $\equiv_\mu$, during the execution of the procedure referred to by Lemma 3. Since this latter guarantees that such execution takes $\mathcal{O}(n)$ per mask, we have that KMR requires a total of $\mathcal{O}(2^L n)$ time and space. □

In order to fully solve MMP, we need to select the maximal masks that will form the set $\mathscr{M}(L, T, q) \subseteq \mathscr{Q}(L, T, q)$. This maximality check on the set $\mathscr{Q}(L, T, q)$ computed in Theorem 5 can be done *a posteriori* using the LESS algorithm proposed in [80], with a cost that is linear on the average and quadratic in the worst-case with the size of the input. In our case, we need to apply it to $\mathscr{Q}$ in $\mathcal{O}(L|\mathscr{Q}|^2)$ worst-case time. Given that we have up to 2^L masks of length L in $\mathscr{Q}$, this method has $\mathcal{O}(L2^L)$ average time complexity, and $\mathcal{O}(L2^{2L})$ worst-case time complexity.

Corollary 6. *Using the* KMR *approach, we can solve MMP in $\mathcal{O}(L2^L n)$ average time and space, and $\mathcal{O}(L2^{2L} n)$ worst-case time and space.*

One drawback of the algorithm behind the result stated in Corollary 6 is that if a maximal mask μ is discovered and has a certain number k of `1`s in it, we have to

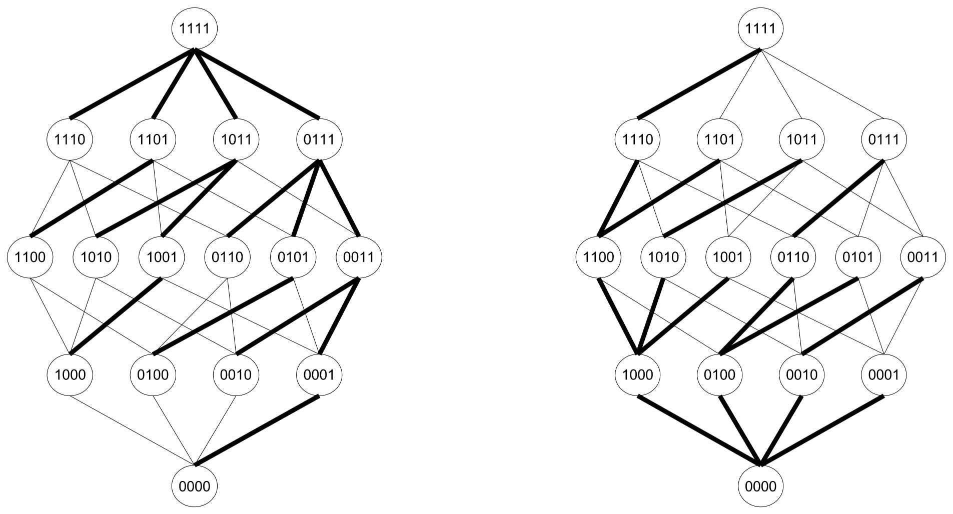


Figure 2.6: Two binomial (spanning) trees B_L, rooted at the top and at the bottom of lattice $\mathscr{L}$.

generate nevertheless all the $\Theta(2^k)$ masks μ' such that $\mu' \preceq \mu$. In Section 2.5, we define a hybrid approach using KMR to evaluate the $P_T(\mu)$ predicate and a branch-and-bound strategy that exploits the anti-monotonicity of the above predicate, in order to obtain a better $\mathcal{O}(2^L n)$ algorithm for MMP (see Theorem 8).

2.5 Adaptive KMR for Maximal Masks

In this section, we describe how to improve the worst-case complexity $\mathcal{O}(L2^{2L}n)$ of Corollary 6. The reason for this bound is that, when using the KMR approach described in Section 2.4, we first generate all the masks of length L having quorum and, then, perform a postprocessing to select those being maximal too. Our task can be better viewed in terms of the lattice $\mathscr{L} = \langle \{\mathtt{1}, \mathtt{0}\}^L, \preceq \rangle$ of 2^L masks, introduced in Section 2.3.2 and illustrated in Figure 2.6. Given a mask μ having quorum, we would like to avoid to compute the partitions for mask μ', such that $\mu' \preceq \mu$. Recall that μ' is called predecessor of μ, and the latter is called successor of μ'. In general, given a mask, its successors are those masks that are reachable going upward in the lattice $\mathscr{L}$ and its predecessors are those reachable going downward.

2.5.1 Lattice traversal

Our idea can be summarized as follows. Suppose that we compute the set $\mathscr{Q}(L/2, T, q)$ of all masks of length $L/2$ that have quorum q, using Theorem 5, in $\mathcal{O}(2^{L/2}n)$ time and space. We store these masks using perfect hashing [50], so each can be retrieved in $\mathcal{O}(1)$ worst-case time. Instead of considering all $\mathcal{O}(2^{L/2}) \times \mathcal{O}(2^{L/2})$ possible concatenations of two masks in $\mathscr{Q}(L/2, T, q)$, we perform concatenation on demand, thus obtaining an *adaptive* KMR approach. The masks of length $L/2$, which we decide to concatenate in $\mathcal{O}(n)$ time using Lemma 3, are chosen according to a suitable traversal of the lattice $\mathscr{L}$. We can employ two different pruning strategies, analogously to the *apriori* algorithm [91], where μ is the current mask of length L in a traversal of $\mathscr{L}$:

1. if μ has the quorum, we do not check its predecessors in $\mathscr{L}$ since they have quorum but cannot be maximal (since they are less specific);
2. if μ has not the quorum, we do not check its successors because they cannot have quorum either (since having quorum is an anti–monotone property).

In the former case, we simulate the traversal of $\mathscr{L}$ by enumerating the masks of length L starting from $1 \dots 1$ and proceeding to less specific masks, which are downward in $\mathscr{L}$, while in the latter case we start from $0 \dots 0$ and go upward looking for more specific masks. In both cases, whenever we need to build the partition π_μ for the current mask μ, we split it as $\mu = \mu_1\mu_2$ such that $|\mu_1| = |\mu_2| = L/2$ (if L is not a multiple of 2, we can proceed with an overlap of μ_1 and μ_2). If both $\mu_1, \mu_2 \in \mathscr{Q}(L/2, T, q)$, we apply Lemma 3 to them to check whether μ has quorum and compute its partition π_μ. Otherwise, we declare that μ does not have quorum in T. We denote this checking operation by the predicate $P_T(\mu)$.

Since we apply Lemma 3 to at most 2^L masks of length L, the complexity of the algorithm is $\mathcal{O}(2^L n)$. It remains to see how to enumerate the masks avoiding those that are non-maximal.

Since the mask lattice $\mathscr{L}$ is isomorphic to the powerset $\mathcal{P}(\{0, \dots, L-1\})$ (see Section 2.3.2), our implicit traversal of $\mathscr{L}$ can be obtained by enumerating all the subsets of $\{0, \dots, L-1\}$ visiting the corresponding *binomial tree* [185], which is also a spanning tree for $\mathscr{L}$. We recall that a binomial tree B_k is an ordered tree representing the set $\mathcal{P}(\{0, \dots, k-1\})$, and can be defined recursively as follows: B_0 consists of a single node and, for $k > 0$, B_k consists of two binomial trees B_{k-1}, where the root of the former is added as the rightmost child to the root of the latter. Figure 2.6 shows two possible binomial trees B_L, both spanning $\mathscr{L}$. The first tree, shown on the left, is rooted at the top of the lattice and can be employed to implement the first pruning strategy, avoiding to visit the predecessors of the mask μ (downward in the lattice). The second tree, shown on the right, is rooted at the bottom of the lattice and can be employed to implement the second pruning strategy, avoiding to visit the successors of μ. Since we want to identify the maximal masks, we opt for the first tree, starting from the top of the lattice $\mathscr{L}$.

2.5.2 Implementation

Algorithm 1 shows the main steps when starting on top. Line 2 checks if mask $1 \dots 1$ has quorum and, if this is so, that mask is the only one returned since any other masks would be less specific. Otherwise, create a queue D containing mask

Algorithm 1 Top-down binomial-tree traversal of the lattice $\mathscr{L}$

Input: *The predicate $P_T(\mu)$ for checking if μ has quorum q in text T.*

Out: *The set $\mathscr{M}(L, T, q)$ of all maximal masks with quorum q in T.*

```
1:  $\mathscr{M} := \varnothing$
2:  if $P_T(\mathtt{1}\ldots\mathtt{1})$ then
3:      return $\{\mathtt{1}\ldots\mathtt{1}\}$
4:  end if
5:  Queue $D := \{\mathtt{1}\ldots\mathtt{1}\}$
6:  while not empty($D$) do
7:      $\mu := \text{head}(D)$
8:      $\mu' := \text{next\_immediate\_predecessor}(\mu)$
9:      if $\mu' = \text{null}$ then
10:         dequeue($D$)
11:     else
12:         if $P_T(\mu')$ then
13:             if $\mathscr{M}$ does not contains $\mu''$ s.t. $\mu' \preceq \mu''$ then
14:                 $\mathscr{M} := \mathscr{M} \cup \{\mu'\}$
15:             end if
16:         else
17:             enqueue($\mu'$, $D$)
18:         end if
19:     end if
20: end while
21: return $\mathscr{M}$
```

$\mathtt{1}\ldots\mathtt{1}$. As long as D is not empty, the main loop on line 6 selects a mask μ and generates one of its immediate predecessors μ' that are not yet visited. (In order to obtain μ', function next_immediate_predecessor systematically switches a `1` into `0` in μ at a time, so $\mu' \preceq \mu$ and they differ in one bit.) If μ' does not exist, then it means that all of μ predecessors have been already visited and μ is dequeued; otherwise, line 12 checks if μ' has quorum. If this is so, μ' is added to $\mathscr{M}$ if and only if a more specific mask is not already in it (lines 13–14). Otherwise, μ' is not enqueued, because the anti-monotonicity of $P_T(\mu)$ guarantees that all of its predecessors have also quorum but they are less specific. Instead, if μ' has not quorum, it is enqueued because one of its predecessors could have quorum.

Implementing D as a queue actually leads to a breadth-first visit of the binomial tree, because we visit a node of level $i + 1$ when all the node of level i have been removed from the queue. Conversely, implementing D as a stack leads to a depth first visit of the tree, but we have to pay more attention when adding a new mask to $\mathscr{M}$ (i.e. all of the predecessors of the mask must be removed).

Although in the worst case scenario almost all the masks of $\mathscr{L}$ must be checked, the pruning strategies can affect heavily the performance of the algorithms. Proceeding top-down with Algorithm 1, if all the interesting masks are close to the top, they are quickly found visiting only a small fraction of the 2^L masks of the lattice. For example, if only $\texttt{1}\ldots\texttt{1}$ has quorum, $P_T(\mu)$ is evaluated once. (Similar considerations hold for the bottom-up traversal of $\mathscr{L}$.)

Unfortunately, Algorithm 1 cannot yet obtain $\mathcal{O}(2^L n)$ time, since checking the condition in line 13 can take time $\mathcal{O}(L\,|\mathscr{M}|) = \mathcal{O}(L2^L)$ per mask, thus giving a total cost of $\mathcal{O}(2^L(n + L2^L))$. We therefore discuss how to refine the breadth-first traversal of Algorithm 1 to get $\mathcal{O}(2^L n)$ time.

2.5.3 Safe masks

Consider the lattice $\mathscr{L}$ and call level 0 the top mask $\texttt{1}\ldots\texttt{1}$, level 1 its predecessors, and so on, up to level L, which is the bottom mask $\texttt{0}\ldots\texttt{0}$. Also, consider the maximal masks in $\mathscr{L}$ (for the given predicate $P_T(\mu)$) and their predecessors.

Definition 3. *We call a mask μ on level i* safe, *where $0 \le i \le L$, if μ is not predecessor of any maximal mask on levels $0, 1, \ldots, i-1$.*

Note that a safe mask μ itself can be maximal (i.e. $P_T(\mu)$ holds), which is consistent with our definition of safeness.

Our goal is to modify Algorithm 1, so that it runs $P_T(\mu)$ only for safe masks μ on each level $i = 0, 1, \ldots, L$. The rationale is that, having traversed the first i levels of the lattice $\mathscr{L}$ and having found the maximal masks on these levels, we cannot eliminate a priori any safe mask μ on level i without first testing $P_T(\mu)$ on it. We show how to find safe masks on each level. Initially, for $i = 0$, the mask $\texttt{1}\ldots\texttt{1}$ is trivially safe.

During the top-down (breadth-first) traversal of $\mathscr{L}$, let us call S_i the set of safe masks on level i. We enforce the invariant that S_i is indeed the set of masks that are in queue D on level i. The other masks on level i are not of interest to us, since they surely have a maximal successor. We show how to produce S_{i+1}, so that the traversal can insert the masks from S_{i+1} into queue D for the next level $i + 1$. We need the crucial lemma below.

Lemma 7. *Let M_i be the set of maximal masks on level i, where $0 \le i \le L$. Then, the following properties hold for each mask μ:*

(i) $\mu \in M_i$ if and only if $\mu \in S_i$ and $P_T(\mu)$ holds (hence, $M_i \subseteq S_i$).

(ii) $\mu \in S_{i+1}$ if and only if all the immediate successors *of μ are in $S_i - M_i$.*

Proof. We consider the two properties in order. (i) Since μ is maximal, it has quorum while none of its successors can have the quorum, hence it is safe. The converse also holds because if μ has quorum and none of its successors is maximal (hence has quorum), then it is maximal by definition.

(ii) A mask μ on level i either is maximal ($\mu \in M_i$), or it has quorum but is not maximal, or it is safe but not maximal ($\mu \in S_i - M_i$), since no other cases are possible. (In particular if μ is not safe, then it must have quorum since it is the predecessor of a maximal mask.) The first implication ($\Rightarrow$) is equivalent to say that if there exists μ' immediate successor of μ such that $\mu' \notin S_i - M_i$ then $\mu \notin S_{i+1}$. From the above observation, it follows that $\mu' \notin S_i - M_i$ implies that μ' has quorum; hence, either μ' or one of its successors must be maximal, and μ cannot be safe having a maximal mask among its successors. To prove the second implication ($\Leftarrow$) it suffices to observe that if all the immediate successors μ' of μ are in $S_i - M_i$, they all do not have the quorum and, by the anti-monotonicity of $P_T(\mu)$, none of their successors upward in the lattice can have the quorum. Hence, μ is safe. $\square$

2.5.4 More efficient implementation

We now show how to exploit Lemma 7 during the traversal. First, since the queue D stores the set S_i, we examine the masks $\mu \in S_i$ and perform the check with $P_T(\mu)$ by Lemma 7(i). Second, we remove M_i from the queue D, which now stores the set $S_i - M_i$. In order to apply Lemma 7(ii), we recall that each of the immediate predecessors and immediate successors of a mask μ differs from μ in exactly one position. We generate all immediate predecessors of the masks in the queue $D = S_i - M_i$. In this way, we create a superset of S_{i+1} from which we select only the masks that have all their *immediate successors* in the queue.

We detail more this task. Recall that D denotes the set $S_i - M_i$ (this is indeed the content of the queue after the removal of the maximal motifs in it). We generate all the immediate predecessors of the masks in D as follows. Given a mask $\mu \in D$ of arbitrary length L, for any position j of a symbol `1` in μ, we generate the immediate predecessors of μ that have all symbols equal to those in μ except that it contains symbol `0` in position j. Let P_D be the multiset of immediate predecessors so built, which represent the predecessors of the masks in D. By Lemma 7, we have that P_D is a superset of S_{i+1} and a mask $\mu \in S_{i+1}$ if and only if μ has multiplicity $i+1$, that is, μ occurs $i+1$ times in the multiset P_D. For example, supposing $S_2 = \{$`10011`, `11001`, `10101`$\}$, we have $S_3 = \{$`10001`$\}$ since it appears three times in $P_D = \{$`00011`, `10001`, `10010`, `01001`, `10001`, `11000`, `00101`, `10001`, `10100`$\}$.

We can proceed in several ways for this checking. Either we sort the multiset P_D and output the masks that appear (consecutively) $i+1$ times in the sorted multiset or we build a trie on the strings in P_D and output those stored in the leaves with multiplicity $i+1$. From a theoretical point of view, we can build a perfect hash function f on the distinct values in P_D in $\mathcal{O}(|P_D|)$ time and space [50]. In this way, given any two masks μ and μ', we have that $f(\mu) = f(\mu')$ implies $\mu = \mu'$. We then use an array of counters C initially set to zero. For each mask $\mu \in P_D$, we increment $C[f(\mu)]$ by one. At the end, with a further scan of P_D, for each mask μ, if $C[f(\mu)] = i+1$ then we output μ and reset $C[f(\mu)]$ to zero.

2.5.5 Complexity

The overall cost for finding the sets S_i for all levels i can be bounded by $\mathcal{O}(\sum_{i=0}^{L-1}(|S_i|\, L))$ time since $|P_D| \leq |S_i|\,(L-i)$ for level $i+1$. Using the fact the $|S_i| \leq \binom{L}{i}$, we obtain a cost of $\mathcal{O}(\sum_{i=0}^{L-1}\binom{L}{i}\, L) = \mathcal{O}(L2^L)$ for all the masks (instead of paying this cost for a single mask as before).

We can now apply Algorithm 1 in which we do *not* run the test in line 13 but, rather, we follow the traversal indicate by sets S_i on each level i of the lattice of $\mathscr{L}$. In this way, the cost is dominated by $\mathcal{O}(2^L n)$ due to checking $P_T(\mu)$ for the masks $\mu \in \cup_{i=0}^{L} S_i$, since the cost of generating the sets S_i is $\mathcal{O}(2^L\, L) = \mathcal{O}(2^L n)$. The required space depends on the chosen visit strategy. Since we use the breadth-first traversal, the queue D can contain all the $\binom{L}{i}$ masks on level i (i.e. it may happen $|S_i| = \binom{L}{i}$). We have thus proved our main result.

Theorem 8. *Using the adaptive* KMR *approach, we can solve* MMP *computing $\mathscr{M}(L, T, q)$ in $\mathcal{O}(2^L n)$ worst-case time and space.*

As previously discussed in Section 2.1, the cost in Theorem 8 can be upper bounded by $\mathcal{O}(n^{\Theta(1+1/\log_2|\Sigma|)} + \min\{2^L n, (|\Sigma|+1)^L\})$. The latter is always better than the ideal bound of $\mathcal{O}(n^{\Theta(1)} + (|\Sigma|+1)^L)$, that is, than constant-time enumerating and checking all the potential $(|\Sigma|+1)^L$ patterns in T. As previously mentioned, more sophisticated techniques that are the state of the art cannot improve, in the worst case, over the bound given in Theorem 8.

We conclude this section by observing that our bounds hold also for the case in which MMP has the additional requirement of reporting one representative mask for each equivalence class of masks. We recall from Section 2.4.2 that any two masks μ and μ' of the same length are *equivalent* if they have the same number of 1s and μ can be obtained from μ' by a shift of the symbols (and vice versa). In our introductory example, 1110 and 0111 are equivalent and we take the leftmost shift as the representative. We can implement this extension with minor variations in our algorithms. In particular, we append $L-1$ copies of a new special symbol \$ that is an endmarker for the text T, and so it does not belong Σ. We then run our algorithms on the resulting text $T\$\$\cdots\$$ and traverse the lattice $\mathscr{L}$ by considering only the masks having 1 as first symbol (see the subtree induced by these masks in each of the binomial trees shown in Figure 2.6). In this way, whenever we consider a mask, it is always the leftmost shift of its equivalence class, and we cover all these kinds of masks.

Corollary 9. *Using the adaptive* KMR *approach, we can solve* MMP *modulo the equivalence between the masks, by selecting the leftmost shifts of the maximal masks in $\mathscr{M}(L, T, q)$ in $\mathcal{O}(2^L n)$ worst-case time and space.*

2.6 Conclusions and Future Work

In this chapter we introduced a new notion of motifs, called masks, that succinctly represent the repeated patterns for an input sequence T of n symbols drawn from an alphabet Σ. We have shown how to build the set of all frequent maximal masks of length L in $\mathcal{O}(2^L n)$ time and space in the worst case, using a variant of the Karp-Miller-Rosenberg approach, proving that our algorithm performs better than the method based on constant-time enumerating and checking all the potential $(|\Sigma|+1)^L$ candidate patterns in T, after a polynomial-time preprocessing of T. In the rest of this section we discuss some interesting lines of research that can be subject of further investigation.

Output sensitive mask discovery. In the worst case, the number of output maximal masks can be exponential in the mask size L. Obviously in this case there is no hope to enumerate all the maximal masks in polynomial time, because exponential time is required just to output all the maximal masks that have been computed. However, also when the number of returned maximal masks is polynomial in the input size, the adaptive KMR approach does not guarantee that the time required to enumerate the output masks is polynomial in the input size, because during the visit of the mask lattice $\mathscr{L}$, the number of visited non-maximal masks can be exponentially large.

It would be of interest to design an algorithm whose time complexity is polynomial in both the input and output sizes. More precisely, let A an enumeration algorithm for an enumeration problem Π, that takes in input an instance I, and outputs all solutions in the answer set $S(I)$ without duplicates. Let n the input size, m the size of the answer set, and $T(I)$ be the total running time of A to compute all solutions in $S(I)$. The algorithm A is *output-polynomial* if $T(I)$ is bounded by a polynomial $q(n, m)$ [10]. Probably, to achieve this goal a different lattice visit strategy is required.

Improving the lattice visit strategy. In Section 2.3 we introduced the mask lattice $\mathscr{L}$ together with the partial order relation $\mu \preceq \mu'$, formally defining the MMP problem as the problem of finding all maximal masks in $\mathscr{L}$, satisfying the quorum constraint q. This problem is well-known in the data mining community, as the problem of mining maximal frequent itemsets [91]. In fact, as explained in Section 2.2, the notion of mask comes naturally into play when performing data mining for frequent itemsets.

As far as we know, the "dualize and advance" algorithm [86, 87] is the best theoretical approach that can be obtained in terms of running time (although dozens of practical approaches have been proposed in recent years [81]). It sets up an interesting connection between mining itemsets in the lattice of 2^L masks and finding hypergraph traversals [21]. Suppose to have incrementally found some of the maximal masks, say $\mu_1, \mu_2, \ldots, \mu_k$. We build the corresponding hypergraph as follows:

there are L nodes numbered from 1 to L, and there is one hyperedge per mask, where the jth bit in the mask is `0` if and only if the node j is incident to the corresponding hyperedge ($1 \leq j \leq L$). In general, the ith hyperedge connects the nodes that correspond to the `0`s in the ith mask μ_i ($1 \leq i \leq k$). In order to find additional maximal masks (and hence add hyperedges), it suffices to find all the hypergraph traversals as starting points for upward paths in the lattice, where each traversal is a minimal hitting set for the current set of k hyperedges [21].

Following the above idea, we implemented an alternative version of the adaptive KMR approach that is based on the "dualize and advance" lattice visit strategy. However, our preliminary tests are not satisfactory, showing that the adaptive KMR approach is much faster than "dualize and advance" in practice, with respect to the number of masks that are queried for checking their quorum. In other words, the "dualize and advance" method needs to query many more masks than our safe masks, and it is more time-consuming due to the slow hypergraph traversals generation step, that is at the core of the algorithm.

Although the "dualize and advance" approach has shown to be inefficient for our purposes, maybe we can benefit from other approaches that have been successfully applied in the field of frequent itemsets mining, as that discussed in [81]. We plan to explore alternative solutions, based on different and more practical lattice visit strategies.

Cache friendly pattern discovery. In the previous sections we focused on the worst-case complexity of the adaptive KMR algorithm. However, given the scan-and-sort nature of our algorithm, we naturally obtain a cache-friendly solution to our problem as a byproduct. Indeed, our algorithm works also in the *ideal cache model*, introduced by Frigo et al. [73] to generalize the two-level memory model of Aggarwal and Vitter [3] and to deal with such a situation, where M is the size of the fast memory, and B is the size of the block in each transfer between fast and slow memories. The goal is to minimize the number of block transfers. For example, scanning n consecutive elements has a complexity of $\Theta(n/B)$ block transfers while the optimal complexity of sorting is $sort(n) = \Theta\big(\frac{n}{B}\log_{M/B}\frac{n}{B}\big)$ block transfers [37, 71, 73, 184].

In order to get a cache-friendly solution, note that both the construction of the safe masks and the concatenation of two masks of length $L/2$ needed to check $P_T(\mu)$ on each safe mask, require scanning and sorting. Since scanning of consecutive elements is trivially cache-friendly, it suffices to employ a cache-oblivious sorting algorithm, whose cost is $sort(n) = \Theta\big(\frac{n}{B}\log_{M/B}\frac{n}{B}\big)$ block transfers [37, 71, 73]. Note that the sorting cost is the dominating cost for each mask. We can easily see that using the adaptive KMR approach and a cache-oblivious sorting, we can build $\mathscr{Q}(L, T, q)$ and $\mathscr{M}(L, T, q)$ with $\mathcal{O}(2^L\, sort(n))$ block transfers, where $sort(n)$ is the cache complexity of sorting n items.

To our knowledge, this is the first cache friendly solution for a motif discovery problem. It remains an open issue to design an algorithm that is both cache-friendly and output sensitive.

Chapter 3

π-pattern Discovery

3.1 Introduction to π-patterns

Genomes evolve both through small-scale events, like single nucleotide mutations, and large-scale events, that reorganize the genetic materials inside the chromosomes [60]. Examples of the aforementioned large-scale events are mutations of a single gene due to the accumulation of single base mutations, gene loss, large scale deletion events, short and large reversals, and even whole-genome duplications [165, 179]. As observed on primates [171], and in bacteria strains [162], the results of these evolutionary events are that, when two or more genomes are compared in terms of gene order, it is unlikely that genes occur in the same order in all the compared genomes.

However, genes are not randomly shuffled across genomes. Nowadays, it is a common understanding in the scientific community that genes that occur together across genomes are functionally related, since they often code for interacting proteins [134].

In recent years, the investigation of these groups of genes that are called *conserved gene clusters* (*gene clusters* for short), has become a fertile field of investigation in comparative genomics. The most widely studied gene clusters are the *operons.* An operon is a functional unit of genomic material containing a cluster of genes that is controlled by a single regulatory promoter [128]. Operons are an atomic transcription unit. Namely, the physical adjacency of the genes in the operon, allow the transcription process to be regulated by one regulator promoter gene only, that is physically located in the genomic sequence preceding the gene cluster. This shared regulatory mechanism easily allow either to transcribe all the genes of the cluster or none of them by “switching on/off” one regulatory promoter only. Examples of operons are the *lac operon* that is responsible for transport and metabolism of lactose in *E. Coli* and some other bacteria [128], the *gal operon* that encodes enzymes necessary for galactose metabolism [189], and the *trp operon* that codes for the proteins that are responsible for the production of tryptophan [190].

Until recent years, operons were thought to exist solely in prokaryotes, since much more complicated mechanisms for gene regulation were known in eukaryotes (the three above mentioned operons have all been discovered in *E. Coli* and other bacteria). However, the discovery of the first operon in eukaryotes in the early 1990s radically changed the scenario [30]. Moreover, the availability of a large number of sequenced eukaryotic genomes has made the comparative approach, based on the discovery of operons through the detection of conserved gene clusters, competitive with previous approaches based on expensive in vitro analysis, attracting the interest of the algorithmic community.

Several genome models formalizing the notion of gene cluster have been proposed in literature in the last two decades. Different modeling choices and assumptions about input data yield to very different models, with different expressive power, and algorithms for cluster discovery having very different time complexities.

For example, genomes can be represented as *permutations*, where there is a one-to-one correspondence between genes of different genomes, or *strings*, where the same gene can occur multiple times inside a genome, and the number of occurrences in one string are not the same as the number of occurrences in another string. Moreover, we can consider the case of two input genomes, or the most general case of k input genomes, the case of *exact* or *approximate* gene clusters, etc. In Section 3.1.1 and 3.1.2, we review the main frameworks that have been proposed in literature, pointing out the main limitations of these models.

To overcome the above limitations in Section 3.2, we introduce a new model of gene cluster, *fixed length π-patterns*, discussing its relationship with the other models that have been proposed in literature.

The following sections discuss the two main computational steps that are involved in fixed length π-pattern discovery. More precisely, in Section 3.3 we show how to detect the π-patterns satisfying a given quorum threshold q, while in Section 3.4 we discuss how to select the most significant of them in the case of π-patterns with no repeated symbols.

Section 3.5 discusses the problem of selecting the most significant frequent π-patterns in the case of repeated symbols, proving the problem to be hard.

Finally, we draw our conclusions in Section 3.6.

3.1.1 Gene clusters in permutations

The simplest model of gene cluster is the *conserved segments* model [28]. Conserved segments model the idea of a set of genes that occurs in the same order and same orientation in the input sequences. More formally, a *signed string* x (a string where a positive or negative sign, representing the orientation of the gene, is associated to each symbol) is an *occurrence* of a set of symbols $S \subseteq \Sigma$, if it contains all and only the symbols of S (signs do not matter). Two signed strings $x = x_1 \dots x_l$ and $y = y_1 \dots y_l$ are equal either if $x_i = y_i$ for $1 \leq i \leq l$, or $x_i = -y_{l-i+1}$. In other words, each string is equal to itself, or to its reverse complement.

Let $C = \{\pi_1, \ldots, \pi_k\}$ be a set of signed permutations drawn on the alphabet Σ. A subset of Σ is a *conserved segment* if it has an occurrence in each permutation, and all the occurrences are equal. For example, given $\pi_1 = \mathtt{1}\ \mathtt{2}\ {-\mathtt{3}}\ \mathtt{4}\ \mathtt{5}\ \mathtt{6}$ and $\pi_2 = \mathtt{1}\ {-\mathtt{6}}\ \ {-\mathtt{5}}\ \mathtt{2}\ {-\mathtt{3}}\ \mathtt{4}$, the non-singleton conserved segments are sets $\{\mathtt{5},\mathtt{6}\}, \{\mathtt{2},\mathtt{3}\}, \{\mathtt{3},\mathtt{4}\}$, and $\{\mathtt{2},\mathtt{3},\mathtt{4}\}$.

Given two set of symbols $S_1, S_2 \subseteq \Sigma$, and a set of permutations C, S_2 is an *extension* of S_1 if and only if $S_1 \subseteq S_2$, and each occurrence of S_1 in a permutation of C is a substring of an occurrence of S_2. A *maximal conserved segments* is a conserved segments having no extension. In the above example, the maximal conserved segments are $\{\mathtt{1}\}, \{\mathtt{5},\mathtt{6}\}$, and $\{\mathtt{2},\mathtt{3},\mathtt{4}\}$.

Given k input permutations on the alphabet Σ, all the maximal conserved segments can be detected in $\mathcal{O}(k\,|\Sigma|)$ time complexity [23].

For large sets of permutations, we can relax the constraint that a conserved segment has to occur in all the input permutations, asking the occurrence in at least q permutations. In [27], this technique has been used to perform an investigation of animal phylogeny through the gene order on mitochondrial genomes.

The definition of conserved segments can be extended to strings and multisets, but most of the proprieties do not hold anymore when the input strings contain repeated symbols [29].

Common intervals are a generalization of conserved segments, where the symbols of the interval are required to be consecutive in all the input permutations, but they are not required to occur in the same order, or in the same orientation [181]. For example, given the two input permutations $\pi_1 = \mathtt{123456789}$, and $\pi_2 = \mathtt{765924138}$, where π_1 is the identity permutation, the set of symbols $S_1 = \{\mathtt{1},\mathtt{2},\mathtt{3},\mathtt{4}\}$ is a common interval of π_1 and π_2, while $S_2 = \{\mathtt{8},\mathtt{9}\}$ is not, since its symbols are not consecutive in π_2. Differently from the case of conserved segments, in the above example we omitted the signs of the input permutations, since the common intervals framework ignores the signs in the input permutations. Note that, when the first permutation is the identity permutation, all the common intervals are sets of consecutive integers.

Given a pair of permutations drawn from an alphabet Σ, the algorithm proposed in [181] requires $\mathcal{O}(|\Sigma| + N)$ time for extracting all the N common intervals. In [93], this result has been generalized to k permutations, designing a $\mathcal{O}(|\Sigma|k + N)$ time algorithm, for extracting all the N common intervals of the input k permutations. A simpler algorithm having the same time complexity is discussed in [22].

The maximum number of common intervals of a pair of permutations of $|\Sigma|$ symbols is $\mathcal{O}(|\Sigma|^2)$. For instance, if we compare two equal permutations, each substring represents a common interval, yielding $\frac{|\Sigma|\,(|\Sigma|+1)}{2}$ common intervals.

One approach to reduce the number of returned common intervals consists in computing for each common interval its statistical significance, returning only the most significant ones [155].

A different approach uses the rich combinatorial properties of common intervals to select a small set of common intervals (whose size is linear in $|\Sigma|$) as a basis generating all common intervals of the input set of permutations. This is the strategy pursued in [22], where *strong common intervals* are used as basis, and in [93], where *irreducible common intervals* are used instead.

A detailed description of strong common intervals and irreducible common intervals is beyond the scope of this section, and we refer the interested reader to the aforementioned papers. However, since both strong common intervals and irreducible common intervals can be hierarchically organized in the *PQ-tree* data structure that will be one of the main topics of the following sections, in the following we sketch the main idea of this hierarchical organization through an example.

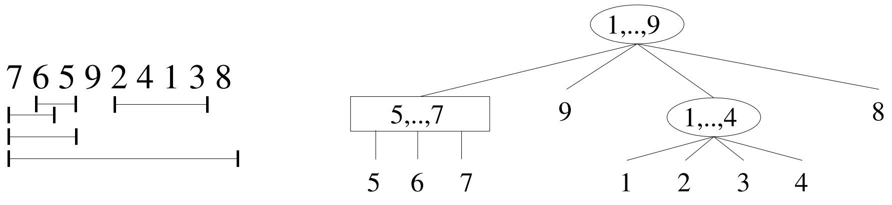


Figure 3.1: Non-singleton common intervals of $\pi_1 = \texttt{123456789}$, and $\pi_2 = \texttt{765924138}$. On the right, the PQ-tree of the strong common intervals.

Figure 3.1 shows all the 5 non-singleton common intervals of $\pi_1 = \texttt{123456789}$, and $\pi_2 = \texttt{765924138}$. As we can see, some of these intervals seem to be related. For example, $\{\texttt{5}, \texttt{6}\}$ and $\{\texttt{6}, \texttt{7}\}$ are subsets of $\{\texttt{5}, \texttt{6}, \texttt{7}\}$.

Let $C = \{\pi_1, \ldots, \pi_k\}$ be the input set of permutations drawn from an alphabet Σ. A *strong common interval* is a common interval of C that does not overlap with any other common interval of C (a set S_1 *overlaps* with S_2 if their intersection is not empty and neither S_1 is contained in S_2, nor viceversa). For example, in the above example $\{\texttt{5}, \texttt{6}, \texttt{7}\}$ is a strong common interval, while $\{\texttt{5}, \texttt{6}\}$ is not, since it overlaps with $\{\texttt{6}, \texttt{7}\}$. More precisely, all the non-singleton strong common intervals of π_1 and π_2 are $S = \{\{\texttt{1}, \texttt{2}, \texttt{3}, \texttt{4}, \texttt{5}, \texttt{6}, \texttt{7}, \texttt{8}, \texttt{9}\}, \{\texttt{5}, \texttt{6}, \texttt{7}\}, \{\texttt{1}, \texttt{2}, \texttt{3}, \texttt{4}\}\}$.

Since two strong common intervals are either disjoint or one is contained in the other, in the PQ-tree where each strong common interval is the child of the smallest strong common interval that properly contains it, each node has at least two children. If $|\Sigma|$ is the number of symbols in each permutation, it follows that the number of internal nodes of the tree, representing the non-trivial common intervals is $\mathcal{O}(|\Sigma|)$.

The ordered tree of strong common intervals for π_1 and π_2 is represented on the right of Figure 3.1. The root of the tree is the set $\{\texttt{1}, \ldots, \texttt{9}\}$, while the leaves are the singletons. Figure 3.1 shows that the internal nodes of the tree are classified into two types of node. Given a node u of the tree, if any union of consecutive children is a common interval, u is a Q-node (represented by rectangles). Otherwise, if no union of consecutive children of the node u is a common interval, except the union of all of its children, u is a P-node (represented by ellipsis). For example, the node representing the strong common interval $\{\texttt{5}, \texttt{6}, \texttt{7}\}$ is a Q-node, since both $\{\texttt{5}, \texttt{6}\}$ and

$$T_1 = \underline{\texttt{1} * \texttt{2 3 4}} * \boxed{\texttt{5}} * * \overline{\texttt{6 7 8}}$$
$$T_2 = \boxed{\texttt{5}} * * \underline{\texttt{2} * \texttt{3 4} \overline{\texttt{6 1}}}\overline{\texttt{8 7}}$$

Figure 3.2: Occurrences of the three gene teams $\{\texttt{1},\texttt{2},\texttt{3},\texttt{4}\}$, $\{5\}$, and $\{\texttt{6},\texttt{7},\texttt{8}\}$ in the input strings T_1 and T_2. The gap threshold is $\delta = 1$.

$\{\texttt{6},\texttt{7}\}$ are common intervals, while the node representing $\{\texttt{1},\texttt{2},\texttt{3},\texttt{4}\}$ is a P-node, since neither $\{\texttt{1},\texttt{2}\}$, nor $\{\texttt{2},\texttt{3}\}$, nor $\{\texttt{3},\texttt{4}\}$, nor $\{\texttt{1},\texttt{2},\texttt{3}\}$, nor $\{\texttt{2},\texttt{3},\texttt{4}\}$ are common intervals.

This tree is an example of a general structure, known as *PQ-tree* [33], that has been introduced to represent sets of permutations (we will discuss the PQ-tree data structure in more detail in Section 3.2). Given the PQ-tree of the strong common intervals of the set of permutations C, in [22], the authors proved that a set S is a common interval of C if and only if it is the union of consecutive children of a Q-node, or the union of all the children of a P-node.

Irreducible common intervals are defined differently. A common interval is *irreducible* if it is not the union of two overlapping common intervals. In the example in Figure 3.1, the set of non-singleton irreducible intervals is $I = \{\{\texttt{1},\texttt{2},\texttt{3},\texttt{4},\texttt{5},\texttt{6},\texttt{7},\texttt{8},\texttt{9}\}, \{\texttt{5},\texttt{6}\},\{\texttt{6},\texttt{7}\},\{\texttt{1},\texttt{2},\texttt{3},\texttt{4}\}\}$. The set $\{\texttt{5},\texttt{6},\texttt{7}\}$ is not irreducible since it is the union of the two overlapping intervals $\{\texttt{5},\texttt{6}\}$ and $\{\texttt{6},\texttt{7}\}$. As proved in [93], irreducible intervals form a basis of size $\mathcal{O}(|\Sigma|)$, that generates all common intervals by unions of overlapping irreducible intervals.

The most general model of gene cluster on permutations is that of *gapped common intervals*. A *gap* inside the occurrence of common intervals S is a substring of the occurrence, which does not contain symbols in S. For example, the string $x = \texttt{1743892}$ is an occurrence of $S = \{\texttt{1},\texttt{2},\texttt{3},\texttt{4}\}$ containing the gaps $y_1 = \texttt{7}$ and $y_2 = \texttt{89}$ of size 1 and 2, respectively.

The notion of gapped common interval has been formalized in [19, 24], under the name of *gene teams* (for a complete dissertation of the more relevant properties of gene teams, and the variants that have been analyzed in literature, the reader can refer to [96], where they are referred as *max-gap clusters*). Given a collection of strings C drawn from an alphabet $\Sigma \cup \{*\}$, such that each string of C is a permutation of Σ, when the symbols $\{*\}$ are removed, and a non-negative gap threshold $\delta \geq 0$, a subset of Σ is a *gene team* if it has an occurrence with maximum gap size δ in each string of C, and the occurrences cannot be extended. For example, given the two input strings $T_1 = \texttt{1} * \texttt{234} * \texttt{5} * * \texttt{678}$, and $T_2 = \texttt{5} * * \ 2 * \texttt{346187}$, Figure 3.1.1 highlights the occurrences of the three gene teams found for $\delta = 1$. Note that $\{\texttt{2},\texttt{3},\texttt{4}\}$ is not a gene team, since it is not maximal. It can be extended both in T_1 and T_2 to $\{\texttt{1},\texttt{2},\texttt{3},\texttt{4}\}$. Also note that, differently from ungapped common intervals, gene team occurrences can overlap. However, as proved in [24], their number is $\mathcal{O}(|\Sigma|)$, and, given k input strings, they can be detected in $\mathcal{O}(k|\Sigma| \log^2 |\Sigma|)$ time.

3.1.2 Gene clusters in strings with multiplicities

The constraint that the input sequences of the dataset must be permutations, with no repeated symbols, can be too stringent when modeling real world genomic sequences. More frequently, genes are found in several copies within the genome of a species. Some approaches have been proposed in literature to eliminate the duplicates, transforming the input strings into permutations [122, 164]. However, when repeated symbols inside the same string represent *paralogous* genes (genes sharing a structural similarity because they have been derived from a common ancestral gene), the removal of repeated genes, can result in an unacceptable loss of information.

The notion of common intervals that we have discussed in Section 3.1.1 can be generalized to strings with repeated symbols [55, 56]. Given a set of strings $C = \{s_1, \ldots, s_k\}$, drawn from alphabet Σ, a set $S \subseteq \Sigma$ is a common interval if it has an occurrence in each string of C. The multiplicities of the symbols inside an occurrence do not matter. For example, although the strings $x_1 = \texttt{abaac}$, and $x_2 = \texttt{bac}$ contain a different number of $\texttt{a}$ symbols, they are both valid occurrences of the common interval $S = \{\texttt{a}, \texttt{b}, \texttt{c}\}$.

Let n the sum of the lengths of the strings in C. The maximum number of common intervals in C is bounded by $\mathcal{O}(n^2)$. However, differently from the case of common intervals in permutations, where strong common intervals and irreducible common intervals form a linear space basis for all the common intervals, in the case of strings no such basis is known in literature.

The algorithms to detect common intervals in strings rely on the concept of *fingerprint* of a string [9]. A fingerprint of a string is the set of symbols that occurs in the string. For example, in the previous example both the string $x_1 = \texttt{abaac}$, and $x_2 = \texttt{bac}$ have the same fingerprint $\{\texttt{a}, \texttt{b}, \texttt{c}\}$.

As proved in [9], given an input collection of strings drawn from Σ, by using string fingerprint, the set of common intervals, together with their occurrences can be computed in time $\mathcal{O}(|\Sigma| \log |\Sigma| n \log(n))$, where n is the sum of the length of the input strings. In [56] the above fingerprint naming technique has been improved to $\mathcal{O}(|\Sigma| \log |\Sigma| n)$, while in [116] an $\mathcal{O}((occ + n) \log(|\Sigma|))$ time algorithm is discussed, where occ is the total number of occurrences of the fingerprints of C. A different approach not based on the above fingerprint naming technique, initiated in [55], and improved in [166, 56], allows for computing the common intervals of C in time $\mathcal{O}(n^2)$.

Gapped common intervals on strings are the most general formal model of common intervals [157, 92]. They are the string counterpart of the gapped common intervals, which we described in Section 3.1.1 for the case of permutations. However, differently from the case of permutations where efficient algorithms have been described in literature, in the case of input string with repeated symbols, the number of gapped common intervals on strings can be exponential in the size of the alphabet $|\Sigma|$ [23]. For this reason, efficient algorithms are only known when some additional hypothesis on the common interval structure are given. For the comparison of exactly two input genomes, a polynomial time algorithm is described in [92], while if the maximum gap size is bounded by a constant, a practical solution that has allowed the computation of gapped common intervals for genomes with thousands of genes is described in [157].

3.2 Fixed Length π-patterns

In Section 3.1.2, we observed that when strings with repeated symbols are given in input, instead of permutations, the rich set of combinatorial properties that characterizes the gene cluster model on permutations does not hold. To overcome this limitation, in the following sections we focus on different kinds of patterns, which are known in literature as *permutation patterns* or *π-patterns* [65].

However, our approach differs from that in [65] in two fundamental aspects. First, we focus on *fixed length* π-patterns. More precisely, we assume that the maximum size of the patterns to be discovered is part of the input. Although at first sight this assumption can appear too restrictive, since it requires additional information about the maximum size of π-patterns in the input dataset, this is not the case of gene clusters that are usually small with respect to the size of the dataset. For example, in Section 3.1 we recall that operons prediction is one of the main applications of gene cluster discovery. In this context, although the input strings of genes can contain thousands, or even millions of genes, the largest known operons contain no more than 5–10 genes [128]. Hence, it is of interest to design an algorithm whose time complexity depends on the maximum size of the π-patterns, and not only on the size of the input strings.

The second difference between our approach and that in [65] is that we do not rely on a notion of maximality to reduce the number of output π-patterns. Instead, we aim at defining a scoring function $score(p, \mathcal{L}(p))$, by which we rank higher π-patterns showing a highly-conserved structure in terms of common intervals of the pattern occurrences, than π-patterns whose structure is less conserved.

A final comment on one of the major phenomenons in genome evolution, namely gene duplication. Although our model is more general than the models based on permutations that cannot handle duplicated genes, we are aware that modeling duplicated genes as duplicated symbols is not a completely satisfactory choice. In fact, on the biological side, the outcome of a duplication event is not to maintain two (or more) isofunctional copies of the same gene (i.e. symbol), but to have genes sharing the same main bio-physical properties, but with different functions.

Although arguable, our selection of modeling duplicated genes as duplicated symbols is fair enough to model a broad range of situations, and the vast majority of works in the literature rely on the same assumption [122, 23, 19]. In defining the π-pattern discovery problem we are going to make the above simplifying assumption, assuming that duplicated copies of the same gene are isofunctional, and that can be modeled by multiple copies of the same symbol. However, before formalizing the

π-pattern discovery problem, we need some preliminary definitions.

3.2.1 Preliminary definitions

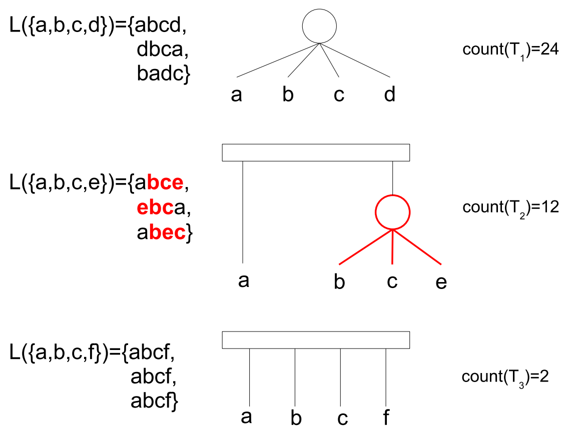


Figure 3.3: On the left, the three π-patterns $p_1 = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$, $p_2 = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{e}\}$, $p_3 = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{f}\}$, together with their occurrence lists. On the right, the corresponding minimal PQ-trees. Ranking π-patterns by increasing values of $count(T_i)$, we rank higher π-patterns showing a highly-conserved structure (in terms of common intervals of the occurrences of the pattern), than π-patterns whose structure is less conserved.

Let T a string drawn from an alphabet $\Sigma = \{\sigma_1, \ldots, \sigma_{|\Sigma|}\}$. The *fingerprint* of the string T, $fing(T)$, is the vector in $\{0,1\}^{|\Sigma|}$ such that $fing(T)[i] = 1$ if σ_i occurs in T, and $fing(T)[i] = 0$ otherwise. Its *signature*, $sign(T)$, is the vector in $\mathbb{N}^{|\Sigma|}$ such that its i-th component represents the number of times the character σ_i occurs in T. A *π-pattern* p of size m is a multiset of symbols drawn from Σ such that the sum of the multiplicities of the symbols in p is m. The π-pattern p *occurs* in the position i of T if $sign(p) = sign(T[i \ldots i+m-1])$ (i.e. $T[i \ldots i+m-1]$ is a permutation of p). The *occurrence list* of the π-pattern p, $\mathcal{L}(p)$, is the list of all the occurrences of p in the string T. For example, the π-pattern $p = \{\mathtt{a}, \mathtt{b}(2), \mathtt{c}\}$ occurs twice in the text $T = \mathtt{cabbca}$, and its occurrence list is $\mathcal{L}(p)$={0,1}.

Sets of permutations can be represented by the PQ-tree data structure that has been introduced by Booth and Lueker in [33]. Specifically, a PQ-tree is a rooted tree whose internal nodes are of two types: *P-nodes* that do not define any specific ordering among their children; *Q-nodes* whose children can appear either in left-to-right order or in right-to-left order. Each leaf of a PQ-tree T is labeled with a symbol of the input alphabet Σ, and the *frontier* of T, denoted by $front(T)$, is the sequence of the symbols obtained by reading the labels of the leaves from left to right. For example, Figure 3.3 shows three PQ-trees, having frontiers $front(T_1) = \mathtt{abcd}$, $front(T_2) = \mathtt{abce}$, and $front(T_3) = \mathtt{abcf}$, respectively. In the following sections we represent a PQ-tree in textual form as a parenthesized string, where the children of a P-node are represented between curly brackets, while the children of a Q-node are represented between round brackets. For example, the second PQ-tree in Figure 3.3 is represented as $T_2 = (\mathtt{a}, \{\mathtt{b}, \mathtt{c}, \mathtt{e}\})$.

Given two PQ-trees T and T', we say that T is *equivalent* to T' (written $T \equiv T'$) if one tree can be obtained from the other by permuting the children of one or more P-nodes, and by reversing the children of some Q-nodes. The set of the frontiers of all the trees that are equivalent to T is denoted by $Fr(T)$, and we say that T represents all the frontiers in $Fr(T)$. Moreover, we denote the size of the set of the frontiers of T as $count(T)$. For example, the tree $T'_2 = (\{\mathtt{c}, \mathtt{b}, \mathtt{e}\}, \mathtt{a})$ obtained by swapping the left subtree of T_2 in Figure 3.3 with the right subtree, and permuting the leaves c, b, and e is equivalent to T_2. It represents the same set of permutations as T_2, namely $Fr(T'_2) = \{\mathtt{abce}, \mathtt{abec}, \mathtt{acbe}, \mathtt{aceb}, \mathtt{aebc}, \mathtt{aecb}, \mathtt{bcea}, \mathtt{beca}, \mathtt{cbea}, \mathtt{ceba}, \mathtt{ebca}, \mathtt{ecba}\}$.

Since a P-node (or a Q-node) having one child can be removed from T without changing $Fr(T)$, and a P-node with two children can be replaced by a Q-node (it represents the left-to-right and right-to-left permutations only), we define the *canonical form* of a PQ-tree by constraining each Q-node to have at least two children, and each P-node to have at least three children. In the following sections, we assume that each PQ-tree is in canonical form.

Given a set of permutations drawn from an alphabet Σ, $C = \{\pi_1, \ldots, \pi_k\}$, the *minimal consensus PQ-tree* [122], provides a succinct representation of the permutation set C, that highlights which are the subsets of symbols appearing consecutively in all the permutations. In other words, it highlights which are all the common intervals of C. More precisely, the minimal consensus PQ-tree of C (*minimal PQ-tree* for short), is the PQ-tree T such that $C \subseteq Fr(T)$, and there exists no $T' \not\equiv T$ such that $C \subseteq Fr(T')$ and $|Fr(T')| \leq |Fr(T)|$. In other words, T represents all the permutations in C, and the number of permutations represented by T that are not in C is minimal. Figure 3.3 shows for each occurrence list on the left the corresponding minimal PQ-tree. For example, the minimal PQ-tree of the second pattern in Figure 3.3, $p_2 = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{e}\}$, that has occurrence list $\mathcal{L}(p_2) = \{\mathtt{abce}, \mathtt{ebca}, \mathtt{abec}\}$, is $T_2 = (\mathtt{a}, \{\mathtt{b}, \mathtt{c}, \mathtt{e}\})$. In fact, $\mathcal{L}(p_2) \subseteq Fr(T_2)$, and each other PQ-tree T'_2 representing all the permutations in $\mathcal{L}(p_2)$ represents a higher number of frontiers (i.e. $count(T'_2) \geq count(T_2)$).

3.2.2 Ranking π-patterns: the idea

We are now ready to define the π-pattern discovery problem.

Problem 2 (π-pattern Discovery Problem). *Input: a text $T = [0 \ldots n]$ drawn from an alphabet Σ, the maximum size L of the π-patterns, and the quorum threshold q. Output: the set $\mathscr{P}_{\leq L}$ containing all the π-patterns of maximum size L, occurring at least q times in T, together with their occurrence lists.*

In the following, we do not consider π-patterns of size 1 because it is trivial to detect the symbols of the alphabet occurring at least q times in T (i.e. $\mathscr{P}_{\leq L} = \mathscr{P}_2 \cup \ldots \cup \mathscr{P}_L$). The number of π-patterns of any size, $\mathscr{P}_n = \mathscr{P}_2 \cup \ldots \cup \mathscr{P}_{n-1}$, obviously depends on the input quorum q, but in the worst case it can be $\mathcal{O}(n^2)$, because each π-pattern can start at an arbitrary position i of T, ending at an arbitrary position $j > i$.

Differently than previous approaches where the information about symbol multiplicities is discarded, and some notion of maximality is used to reduce the number of output patterns, we pursue a different approach that aims at defining a scoring function $\mathit{score}(p, \mathcal{L}(p))$, by which we rank higher π-patterns showing a highly-conserved structure across their occurrences in terms of common intervals, than π-patterns whose structure is less conserved. The idea is simple. In [122], the authors show that higher the number of common intervals of C, the more structured is the minimal PQ-tree T for C, and lower is the number of frontiers represented by T. Consider the example in Figure 3.3, and the two π-patterns $p_1 = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$ and $p_2 = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{e}\}$ together with their occurrence lists $\mathcal{L}(p_1)$ and $\mathcal{L}(p_2)$. The set of non-singleton common intervals in $\mathcal{L}(p_1)$ is $S_1 = \{\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}\}$ because $\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$ is the only set whose symbols occur consecutively in all the occurrences of $\mathcal{L}(p_1)$, while the set of non-singleton common intervals in $\mathcal{L}(p_2)$ is $S_2 = \{\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}, \{\mathtt{b}, \mathtt{c}, \mathtt{e}\}\}$ because symbols $\mathtt{b}$, $\mathtt{c}$, and $\mathtt{e}$ occur consecutively in all the occurrences of $\mathcal{L}(p_2)$. It follows that, while the minimal PQ-tree for $\mathcal{L}(p_1)$ is $T_1 = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$ that represents $\mathit{count}(T_1) = 4\,! = 24$ frontiers (all the permutations of symbols $\mathtt{a}$, $\mathtt{b}$, $\mathtt{c}$, and $\mathtt{d}$), the minimal PQ-tree for $\mathcal{L}(p_2)$ is $T_2 = \{\mathtt{a}, (\mathtt{b}, \mathtt{c}, \mathtt{e})\}$, where symbols of the common interval $\{\mathtt{b}, \mathtt{c}, \mathtt{e}\}$ label the leaves of the subtree $T'_2 = (\mathtt{b}, \mathtt{c}, \mathtt{e})$ (highlighted in red in Figure 3.3). The number of frontiers represented by T_2 is smaller than that of T_1. In fact, since T_2 represents all the permutations of symbols $\mathtt{a}$, $\mathtt{b}$, $\mathtt{c}$, and $\mathtt{e}$ where symbols $\mathtt{b}$, $\mathtt{c}$, and $\mathtt{e}$ are consecutive, we have that $\mathit{count}(T_2) = 12$. Notice that, by our approach, although p_1 and p_2 are not the same set of symbols, and the set of common intervals of $\mathcal{L}(p_1)$ is different from that of $\mathcal{L}(p_2)$, we can compare the structure of $\mathcal{L}(p_1)$ and $\mathcal{L}(p_2)$ by looking at the number of frontiers represented by their minimal PQ-trees.

From above it follows that we can set $\mathit{score}(p, \mathcal{L}(p)) = \mathit{count}(T)$, where T is the minimal PQ-tree for $\mathcal{L}(p)$, restating Problem 2 as follows:

Problem 3 (Ranked π-pattern Discovery Problem). *Input: a text $T = [0 \ldots n]$ drawn from an alphabet Σ, the maximum size L of the π-patterns, the quorum threshold q, and the number of π-patterns, k, that must be returned. Output: the top-k π-patterns of maximum size L, occurring at least q times in T sorted by increasing value of the $\mathit{score}(p, \mathcal{L}(p))$ function, together with their occurrence lists.*

Figure 3.3 clarifies the idea. Here, the three π-patterns are sorted (bottom-up) by increasing value of $\mathit{score}(p_i, \mathcal{L}(p_i)) = \mathit{count}(T_i)$, where T_i is the minimal PQ-tree for the occurrence list of the pattern p_i. This ordering agrees with the fact that the non-singleton common intervals of $\mathcal{L}(p_3)$ are $\{\mathtt{a}, \mathtt{b}\}$, $\{\mathtt{b}, \mathtt{c}\}$, $\{\mathtt{c}, \mathtt{f}\}$, $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$,

$\{\mathtt{b},\mathtt{c},\mathtt{f}\}$, and $\{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{f}\}$, the non-singleton common intervals of $\mathcal{L}(p_2)$ are $\{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{e}\}$, and $\{\mathtt{b},\mathtt{c},\mathtt{e}\}$, while the only non-singleton common intervals of $\mathcal{L}(p_1)$ is $\{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d}\}$.

3.2.3 Two phase approach for ranked π-pattern discovery

Algorithm 2 Ranked π-pattern discovery algorithm.

```
Input: The input string $S$, the maximum $\pi$-patterns size $L$, the quorum threshold
    $q$, and the number of $\pi$-patterns that must be returned $k$.
Out: Top-$k$ $\pi$-patterns $\mathscr{P}_{\leq L}$.
1: Compute the set of $\pi$-patterns of maximum size $L$ occurring at least $q$ times in
   $S$, together with their occurrence lists.
2: For each occurrence list $\mathcal{L}(p)$ compute the minimum PQ-tree $T$ and $count(T)$.
3: Rank the patterns by increasing value of $count(T)$.
4: Return the top-$k$ $\pi$-patterns and their occurrence lists.
```

Algorithm 2 formalizes the idea described in Section 3.2.2, describing the main conceptual steps to compute the top-k π-patterns of the required maximum size, satisfying the input quorum threshold. First, we compute the set of π-patterns of maximum size L occurring at least q times in the input string S. For each π-pattern p (that can contain repeated symbols) we also compute its occurrence list $\mathcal{L}(p)$. Then, for each π-pattern p detected during the first step, we compute the minimum PQ-tree T for its occurrence list $\mathcal{L}(p)$ and $count(T)$, ranking the patterns by increasing value of $count(T)$. We finally return the top-k patterns.

At this point the main idea should be clear. The following sections present in more details the main steps of Algorithm 2. First, in Section 3.3 we discuss the first phase of Algorithm 2, showing how the fingerprinting algorithm presented in [9, 65] can be improved by saving a $\log(n)$ time factor. Then, in Section 3.4 and 3.5 we analyze the problem of constructing the minimal PQ-tree T, counting the number of frontiers represented by the tree. As discussed in Section 3.4, if the input π-pattern contains no repeated symbols (i.e. it is a set), both the minimal PQ-tree and the number of frontiers represented by the tree can be computed in polynomial time. However, what about in the case of π-patterns with repeated symbols? In Section 3.5 we discuss the hardness of this more general case, highlighting the strong connection between this problem and the generalization of the classic *consecutive ones problem* [33].

3.3 π-pattern Discovery: First Phase

The first phase of Algorithm 2 consists in computing the set of π-patterns of maximum size L occurring at least q times in T, together with their occurrence lists. First, in Section 3.3.1 we describe how the algorithm proposed in [9] can be extended to take into account symbol multiplicities. Then, in Section 3.3.2 we show how to improve the algorithm by saving a $\log(n)$ time factor.

3.3.1 π-pattern discovery by binary tagging tree

In [9] a simple and elegant algorithm has been proposed to compute the set $\mathscr{P}_l$ of the π-patterns of fixed size l having quorum. The algorithm has been originally proposed to group substrings of variable lengths having the same fingerprint (i.e. symbols multiplicities do not count), but it can be easily extended to the case of signatures.The key observation is that two substrings of size l contain the same characters with exactly the same multiplicities if and only if they have the same signatures. For example, $T[i_1 \dots j_1] = \texttt{abcc}$ and $T[i_2 \dots j_2] = \texttt{cabc}$ drawn from $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}\}$ have the same signature $[1, 1, 2, 0]$.

The above idea suggests that the patterns of size l can be grouped together moving a sliding window of size l over the input text T, computing for each substring $T[i \dots i+l-1]$ its signature, hashing the signature into a fingerprint h, and adding the pair (h, i) to an auxiliary data structure mapping fingerprints into occurrence lists (in the following the term fingerprint is not used with the meaning of Section 3.2, but as synonymous of tag).

Although at iteration $i > 1$ of the sliding-window approach, at most two components of the signature vector are changed with respect to the signature of the previous iteration (the component corresponding to the character $T[i+l-1]$ is increased by one, while the component corresponding to $T[i-1]$ is decreased by one), the naive algorithm computes the signature fingerprint of the current substring from scratch, paying $\Omega(l)$ at each iteration.

The algorithm proposed in [9] overcomes the above limitation by making use of the *binary tagging tree* data structure to compute a *perfect hash* of the given signature (each signature is assigned a unique fingerprint), as shown in Figure 3.4.

Assume, for the sake of simplicity, that $|\Sigma|$ is power of 2, if this is not the case add some null characters making its size power of 2: notice that the size of the new alphabet Σ' is no more than twice the size of Σ, hence its size is $\mathcal{O}(|\Sigma|)$ (the maximum number of null symbols, $2^k - 1$, is added when $|\Sigma| = 2^k + 1$ for some k, obtaining a new alphabet of size 2^{k+1}).

The binary tagging tree is a complete binary tree, whose leaves are labeled with the $|\Sigma'|$ components of the signature vector of the current text window (i.e. the first leaf is labeled with the number of occurrence of $\texttt{a}$ in the current window, the second with the number of $\texttt{b}$, etc). Each internal node u is labeled by a fingerprint computed by the fingerprints of its left and right child h_l and h_r. If the pair (h_l, h_r) has been already encountered in some previous iteration and in that case the father node has been tagged with h_f, then u is labeled with h_f. (This is the case, in Figure 3.4, of the root node during the third iteration, where the pair of child fingerprints $(9, 10)$ was already encountered during the previous iteration.) Otherwise, if the pair (h_l, h_r) has not been encountered so far a new fingerprint is assigned to u, as in the case

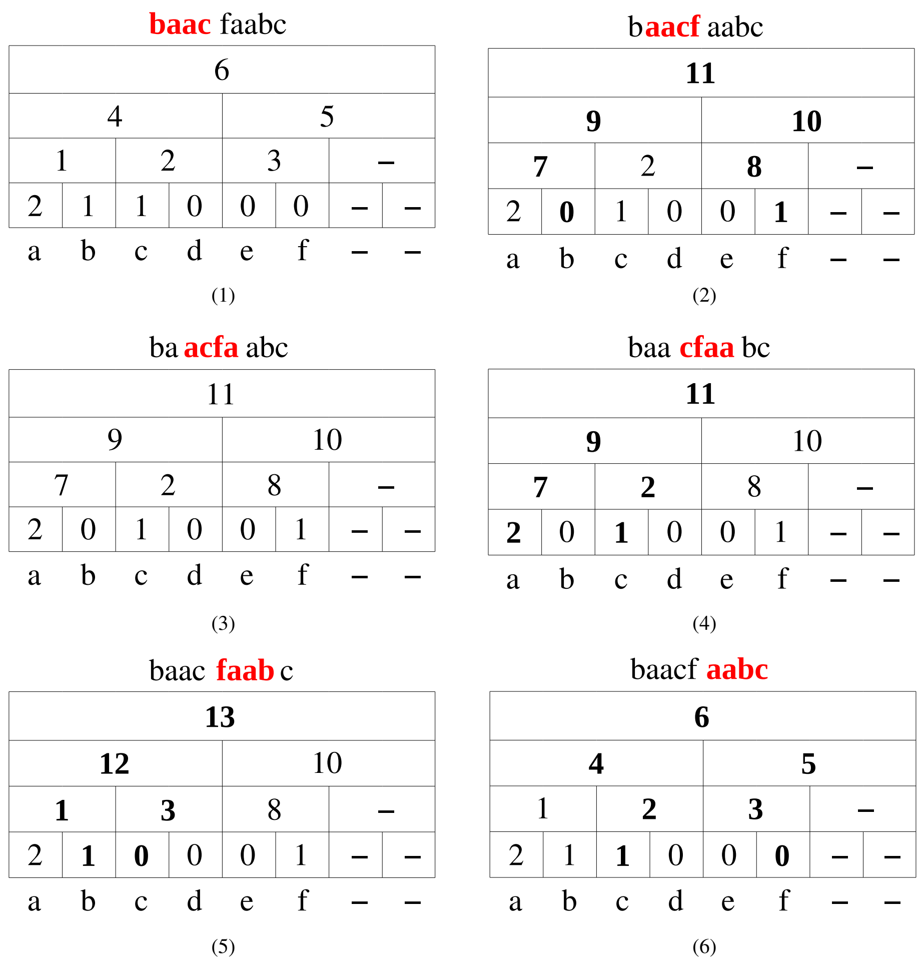


Figure 3.4: The binary tagging tree algorithm for π-pattern discovery, executed on T = baacfaabc with $l = 4$ and $q = 2$ returns the patterns $p_1 = \{\mathtt{a}(2), \mathtt{c}, \mathtt{f}\}$, and $p_2 = \{\mathtt{a}(2), \mathtt{b}, \mathtt{c}\}$ together with their occurrence lists $\mathcal{L}(p_1) = \{1, 2, 3\}$, and $\mathcal{L}(p_2) = \{0, 5\}$.

of the root node during the second iteration, when $(9, 10)$ is processed for the first time. The fingerprint of the current substring is the fingerprint of the root of the tagging tree.

It is easy to see that the above "labeling rule" assigns the same fingerprint to two different substrings having the same signature, since all the nodes of the tree are labeled in the same way (see for example the second and the third tree in Figure 3.4), and that it is not possible for two different signatures to be mapped to the same fingerprint.

Figure 3.4 simulates the algorithm on the input text T = baacfaabc with $l = 4$ and $q = 2$. During the first iteration the sliding window contains the substring $s_1 = T[0 \dots 3]$ = baac, the leaves of the first tagging tree are labeled by the components of

$v_1 = sign(s_1)$, which is computed from scratch, while the labels of the other nodes are all fresh fingerprints, who are assigned bottom up, left to right in increasing order: the fingerprint 6 is assigned to the root and the fingerprint-text position pair $(6, 0)$ is saved into an auxiliary data structure mapping fingerprints in lists of text positions.

At the second iteration the sliding window moves one position to the right, popping out the leftmost `b` and pushing in a new `f`: the new signatures v_2 can be computed by difference from v_1, since only the number of `b` and the number of `f` need to be updated, while the other counters stay the same. Obviously after this update the labels of some nodes need to be changed (they are highlighted in bold), but the number of these labels are at most logarithmic in the number of leaves, since in the worst case it is the number of nodes of two paths from the root to the two different leaves labeled with the updated counters.

The above steps are iterated for each substring in T of size l and in the end the fingerprints occurring at least q times in T are returned together with their occurrence lists. The algorithm has to be repeated $L-1$ times to compute $\mathscr{P}_{\leq L}$ for all the substring sizes in $[2, \ldots, L]$.

The time complexity of the algorithm depends on the maximum number of different fingerprints that can be created scanning the input text, and how much does it cost, given a pair of children fingerprints (h_l, h_r), to decide if it has been already encountered at some previous iteration.

Theorem 10 ([9])**.** *Given an input text T of size n drawn on the alphabet Σ, and the size l of the π-patterns to compute:*

1. *The maximum number of fingerprints generated by the algorithm is $t = \mathcal{O}(|\Sigma| + n \log |\Sigma|)$.*
2. *The maximum number of fingerprints generated at each level is $\mathcal{O}(n)$.*
3. *Given a pair of fingerprints (h_l, h_r) the lookup operation can be implemented in $\mathcal{O}(\log n)$ time.*
4. *The time complexity of the algorithm is $\mathcal{O}(n \log n \log |\Sigma|)$.*
5. *The space complexity of the algorithm is $\mathcal{O}(n \log |\Sigma|)$.*

By running the binary tagging tree algorithm independently for each π-pattern size $l \leq L$ we have that:

Theorem 11. *By using the binary tagging tree algorithm in [9], the π-pattern discovery problem can be solved in $\mathcal{O}(Ln \log n \log |\Sigma|)$ time, and $\mathcal{O}(n \log |\Sigma|)$ space.*

3.3.2 π-pattern discovery by levelwise binary tagging tree

In Section 3.3.1 we showed how $\mathscr{P}_{\leq L}$ can be computed in $\mathcal{O}(L \log |\Sigma|\, n \log n)$ time, and $\mathcal{O}(n \log |\Sigma|)$ space. In this section we improve the algorithm based on binary tagging trees, by saving a $\log(n)$ time factor.

The idea behind the new algorithm is simple. The algorithm described in Section 3.3.1 works in a "depth-first" fashion. In fact, in the running example described in Figure 3.4 first we compute the binary tagging tree for the substring $T[0 \ldots 3] = \texttt{baac}$ from its leaves to its root, then we move to the next substring $T[1 \ldots 4]$, and so on. The fingerprint of each internal node u is computed by looking at the already computed fingerprints of its left and right children (h_l, h_r). If the pair has already been encountered in the previous iteration, then u is labeled with the old fingerprint, otherwise a fresh fingerprint is assigned to u. This lookup operation can be implemented in $\mathcal{O}(\log n)$ time, by using an array of balanced binary search trees indexed by the first fingerprint of the pair (see [65] for details), and must be repeated $\mathcal{O}(\log |\Sigma'|)$ times for each window shift.

The new algorithm rely on a simple observation. Multiple nodes of different binary tagging trees (or even in the same tree) can be labeled with the same fingerprint, but they always occur in the same level. For example, in Figure 3.4, the fingerprint 6 occurs both in the first binary tagging tree, and in the last one. However, in both cases it occurs at level three of the tagging trees (in the following we number the level of the binary tagging tree from the leaves to the root starting from level 0).

This observation suggests that, fixed the substring size l, all the binary tagging trees for the substrings of size l, occurring in the input text T can be computed in parallel, by computing the fingerprints of their nodes levelwise. In other words, first we compute the fingerprints of all the leaves of all the binary tagging trees that are at level one, then the fingerprints for the nodes at level 1, up to the fingerprints of the roots of the trees.

A similar approach has been pioneered in [56] for the problem of detecting all the maximal locations of a given character set $S \subseteq \Sigma$. However, the algorithm described in this section differs from that in [56] in three relevant aspects.

First, while the algorithm in [56] has been designed to handle sets of symbols, our algorithm handle multisets of symbols, where the equality between multiset must take into account not only the symbols that occur in the set, but also their multiplicities.

Second, in [56] the authors are interested in maximal locations of each character set. For example, given the set $S = \{\texttt{a}, \texttt{b}\}$ and the input text $T = \texttt{baaca}$, the prefix $\texttt{baa}$ is a maximal occurrence of S, while the prefix $\texttt{ba}$ is not, since it could be extended one symbol to the right, without changing the set of symbols contained in the string. As we discussed in Section 3.2, we do not make use of any notion of maximality to reduce the number of output patterns, but we pursue a different approach that aims at ranking the output patterns, returning the top-k π-patterns only (note the drawback of the notion of maximality in [56], that sacrifices the information about the symbol multiplicities).

Finally, while the algorithm in [56] (but also the original algorithm in [9]), handles each substring size $l \in [2, \ldots, L]$ independently, our algorithm handles all the substring size in the range $[2, \ldots, L]$ at once.

L_0

b	a	a	c	f	a	a	b	c
b t_0=(1,1,0,0)	a t_4=(1,0,1,1)	a t_8=(1,0,2,2)	c t_{12}=(1,2,3,3)	f t_{16}=(1,5,4,4)	a t_{20}=(1,0,5,5)	a t_{24}=(1,0,6,6)	b t_{27}=(1,1,7,7)	c t_{29}=(1,2,8,8)
a t_1=(1,0,0,1)	a t_5=(2,0,1,2)	c t_9=(1,2,2,3)	f t_{13}=(1,5,3,4)	a t_{17}=(1,0,4,5)	a t_{21}=(2,0,5,6)	b t_{25}=(1,1,6,7)	c t_{28}=(1,2,7,8)	
a t_2=(2,0,0,2)	c t_6=(1,2,1,3)	f t_{10}=(1,5,2,4)	a t_{14}=(1,0,3,5)	a t_{18}=(2,0,4,6)	b t_{22}=(1,1,5,7)	c t_{26}=(1,2,6,8)		
c t_3=(1,2,0,3)	f t_7=(1,5,1,4)	a t_{11}=(2,0,2,5)	a t_{15}=(2,0,3,6)	b t_{19}=(1,1,4,7)	c t_{23}=(1,2,5,8)			

P_1

b	a	a	c	f	a	a	b	c
b (0,1,0,0,0)	a (1,0,0,1,1)	a (1,0,0,2,2)	c (1,0,1,3,3)	f (0,1,2,4,4)	a (1,0,0,5,5)	a (1,0,0,6,6)	b (0,1,0,7,7)	c (1,0,1,8,8)
a (1,1,0,0,1)	a (2,0,0,1,2)	c (1,0,1,2,3)	f (0,1,2,3,4)	a (1,0,0,4,5)	a (2,0,0,5,6)	b (1,1,0,6,7)	c (1,0,1,7,8)	
a (2,1,0,0,2)	c (1,0,1,1,3)	f (0,1,2,2,4)	a (1,0,0,3,5)	a (2,0,0,4,6)	b (2,1,0,5,7)	c (1,0,1,6,8)		
c (1,0,1,0,3)	f (0,1,2,1,4)	a (2,0,0,2,5)	a (2,0,0,3,6)	b (2,1,0,4,7)	c (1,0,1,5,8)			

M_1

0,0−>0 0,1−>1 1,0−>2 1,1−>3 2,0−>4 2,1−>5

L_1

b	a	a	c	f	a	a	b	c
(1,0,0,0)	(2,0,1,1)	(2,0,2,2)	(2,1,3,3)	(1,2,4,4)	(2,0,5,5)	(2,0,6,6)	(1,0,7,7)	(2,1,8,8)
(3,0,0,1)	(4,0,1,2)	(2,1,2,3)	(1,2,3,4)	(2,0,4,5)	(4,0,5,6)	(3,0,6,7)	(2,1,7,8)	
(5,0,0,2)	(2,1,1,3)	(1,2,2,4)	(2,0,3,5)	(4,0,4,6)	(5,0,5,7)	(2,1,6,8)		
(2,1,0,3)	(1,2,1,4)	(4,0,2,5)	(4,0,3,6)	(5,0,4,7)	(2,1,5,8)			

P_2

b	a	a	c	f	a	a	b	c
(1,0,0,0,0)	(2,0,0,1,1)	(2,0,0,2,2)	(0,2,0,3,3)	(1,0,1,4,4)	(2,0,0,5,5)	(2,0,0,6,6)	(1,0,0,7,7)	(0,2,0,8,8)
(3,0,0,0,1)	(4,0,0,1,2)	(2,2,0,2,3)	(1,0,1,3,4)	(2,0,0,4,5)	(4,0,0,5,6)	(3,0,0,6,7)	(1,2,0,7,8)	
(5,0,0,0,2)	(4,2,0,1,3)	(1,0,1,2,4)	(2,2,0,3,5)	(4,0,0,4,6)	(5,0,0,5,7)	(3,2,0,6,8)		
(5,2,0,0,3)	(1,0,1,1,4)	(4,2,0,2,5)	(4,2,0,3,6)	(5,0,0,4,7)	(5,2,0,5,8)			

M_2

0,0−>0 0,2−>1 1,0−>2 1,2−>3 2,0−>4 2,2−>5
3,0−>6 3,2−>7 4,0−>8 4,2−>9 5,0−>10 5,2−>11

L_2

b	a	a	c	f	a	a	b	c
(2,0,0,0)	(4,0,1,1)	(4,0,2,2)	(1,0,3,3)	(2,1,4,4)	(4,0,5,5)	(4,0,6,6)	(2,0,7,7)	(1,0,8,8)
(6,0,0,1)	(8,0,1,2)	(5,0,2,3)	(2,1,3,4)	(4,0,4,5)	(8,0,5,6)	(6,0,6,7)	(3,0,7,8)	
(10,0,0,2)	(9,0,1,3)	(2,1,2,4)	(5,0,3,5)	(8,0,4,6)	(10,0,5,7)	(7,0,6,8)		
(11,0,0,3)	(2,1,1,4)	(9,0,2,5)	(9,0,3,6)	(10,0,4,7)	(11,1,5,8)			

P_3

b	a	a	c	f	a	a	b	c
(2,0,0,0,0)	(4,0,0,1,1)	(4,0,0,2,2)	(1,0,0,3,3)	(0,2,0,4,4)	(4,0,0,5,5)	(4,0,0,6,6)	(2,0,0,7,7)	(1,0,0,8,8)
(6,0,0,0,1)	(8,0,0,1,2)	(5,0,0,2,3)	(1,2,0,3,4)	(4,2,0,4,5)	(8,0,0,5,6)	(6,0,0,6,7)	(3,0,0,7,8)	
(10,0,0,0,2)	(9,0,0,1,3)	(5,2,0,2,4)	(5,2,0,3,5)	(8,2,0,4,6)	(10,0,0,5,7)	(7,0,0,6,8)		
(11,0,0,0,3)	(9,2,0,1,4)	(9,2,0,2,5)	(9,2,0,3,6)	(10,2,0,4,7)	(11,0,0,5,8)			

M_3

0,2−>0 1,0−>1 1,2−>2 2,0−>3 3,0−>4 4,0−>5 4,2−>6
5,0−>7 5,2−>8 6,0−>9 7,0−>10 8,0−>11 8,2−>12
9,0−>13 9,2−>14 10,0−>15 10,2−>16 11,0−>17

L_3

b	a	a	c	f	a	a	b	c
(3,0,0,0)	(5,0,1,1)	(5,0,2,2)	(1,0,3,3)	(0,0,4,4)	(5,0,5,5)	(5,0,6,6)	(3,0,7,7)	(1,0,8,8)
(9,0,0,1)	(11,0,1,2)	(7,0,2,3)	(2,0,3,4)	(6,0,4,5)	(11,0,5,6)	(9,0,6,7)	(4,0,7,8)	
(15,0,0,2)	(13,0,1,3)	(8,0,2,4)	(8,0,3,5)	(12,0,4,6)	(15,0,5,7)	(10,0,6,8)		
(17,0,0,3)	(14,0,1,4)	(14,0,2,5)	(14,0,3,6)	(16,0,4,7)	(17,0,5,8)			

Figure 3.5: The levelwise binary tagging tree algorithm for π-patterns discovery, executed on T = `baacfaabc`, setting $L = 4$ and $q = 2$ returns the patterns $p_1 = \{\mathtt{a}(2), \mathtt{c}, \mathtt{f}\}$, and $p_2 = \{\mathtt{a}(2), \mathtt{b}, \mathtt{c}\}$ together with their occurrence lists $\mathcal{L}(p_1) = \{1, 2, 3\}$, and $\mathcal{L}(p_2) = \{0, 5\}$.

Before going in details, formally describing the algorithm, for π-pattern discovery, we describe the main idea of the algorithm by the running example in Figure 3.5.

Given the input text $T =$ `baacfaabc` we compute the set of π-patterns having maximal size $L = 4$, occurring at least $q = 2$ times in T as follows. We make use of an auxiliary array A, called *signature array*, having size $|\Sigma'| = 8$ (see Section 3.3.1 for the definition of Σ'), that stores the signature of the current string, and that is initialized to $[0, 0, 0, 0, 0, 0, 0, 0]$.

In the first step we compute the fingerprints for the leaves of the binary tagging trees. They represent the multiplicities of the current symbol in the current substring, and are represented as quadruplets in the list L_0. More precisely, each quadruplet $t_j = (n_c, c, i, r)$ represents the multiplicity n_c of the c-th symbol of the alphabet (zero based), in the substring $T[i \dots r]$. For example, in Figure 3.5 the first quadruplet t_0 encodes that only one `b` occurs in the substring $T[0 \dots 0]$, t_1 encodes that only one `a` occurs in $T[0 \dots 1]$, t_2 that two `a` symbols occur in $T[0 \dots 2]$, t_3 encodes that one `c` occurs in $T[0 \dots 3]$, etc.

After L_0 has been initialized, we read each quadruplet $(n_c, c, i, r) \in L_0$, updating the corresponding element in position c of the signature array A. We append to the list P_1 a quintuplet $u_j = (h_l, h_r, m, i, r)$ whose first two components represent the pair $(A[c-1], A[c])$ if c is odd, $(A[c], A[c+1])$ otherwise. In other words, h_l and h_r are the fingerprints of the left and the right children, which are used to compute the fingerprint of the father node u. The other values of the quintuplet represent the index of the pair encoded by the quintuplet (m), and the indices in T where the corresponding string start and end. For example, in Figure 3.5 the first quadruplet being read from L_0 is $t_0 = (1, 1, 0, 0)$. The signature array A is modified into $A_0 = [\mathbf{0}, \mathbf{1}, 0, 0, 0, 0, 0, 0]$, by updating the value in position 1 (note that, A_0 represents the signature of $T[0 \dots 0] =$ `b`). At this point we append the first quintuplet $u_0 = (\mathbf{0}, \mathbf{1}, 0, 0, 0)$ representing the pair of values $(A_0[0], A_0[1])$ to P_1 (the values are highlighted in bold in A_0). This pair of values will be used in the next iteration of the algorithm to assign the fingerprint of the father node in the binary tagging tree.

After t_0, we read $t_1 = (1, 0, 0, 1)$, updating the signature array to
$A_1 = [1, 1, 0, 0, 0, 0, 0, 0]$ and appending the quintuplet $u_1 = (1, 1, 0, 0, 1)$ to P_1. We repeat the same steps for t_2 and t_3, that correspond to the substring $T[0 \dots 2]$ and $T[0 \dots 3]$, respectively, appending $u_2 = (2, 1, 0, 0, 2)$ and $u_3 = (1, 0, 1, 0, 3)$ to P_1. Since the substring $T[0 \dots 3]$ is the string of maximal size starting at position 0 (we recall that $L = 4$), we conceptually shift the sliding window one position to the right, resetting the signature array to $[0, 0, 0, 0, 0, 0, 0, 0]$ before processing t_4.

We repeat the above steps for each quintuplet in L_0, resetting the signature array every time that the sliding window is moved one position to the right (i.e. after reading t_3, t_7, t_{11}, t_{15}, t_{19}, t_{23}, t_{26}, t_{28}, t_{29}).

The quintuplets in P_1, represent the consecutive pairs of fingerprints tagging the leaves of all the binary tagging trees of all the substrings of T having size $1, 2, 3, 4$. We conceptually follow the idea described in Section 3.3.1 by grouping the quintuplets in P_1 by their first two values h_l and h_r, assigning a unique name to each pair (h_l, h_r) in $u_j = (h_l, h_r, m, i, r)$. The new names are shown in Figure 3.5 in M_1, while L_1 is the new list of quadruplet obtained from P_1, by substituting the first two values of each quintuplet with the fresh fingerprint associated with this pair by the mapping M_1.

This is the main difference between the algorithm described in Section 3.3.1 and this levelwise variant. While in the former we compute the fingerprints of the binary tagging tree associated with the first sliding window before that of the nodes of the tree of the second sliding window, in the latter we compute the fingerprints levelwise, by assigning the fingerprints to the leaves of all the binary tagging trees, then to their fathers, and so on, up to the roots of the trees.

The quadruplets in L_1 represent the fingerprints labeling the nodes at level 1 of all the binary tagging trees. Hence, we perform the same steps as above, by reading each quadruplet in L_1, updating the signature array A, appending the corresponding quintuplet to P_2, and when all the quadruplets in L_1 have been processed, by grouping all the quintuplets in P_2 by the first two values, computing the list L_2.

One more iteration is required to compute the list L_3. Each quadruplet in L_3 corresponds to one substring of size $[1, \ldots, 4]$ of T, while the first value of each quadruplet represents the fingerprint of the root of the binary tagging tree associated with the string.

Hence, by grouping the quadruplets in L_3 by their first value, we group together the substrings of size $[1, \ldots, 4]$ of T having the same signature. A final scan of the groups, filtering the groups corresponding to strings of size one, and groups containing strictly less than $q = 2$ strings, yield the π-patterns having size $l = 2, 3, 4$, occurring at least $q = 2$ times in T.

Algorithm 3 formalizes the main computational steps that we discussed in the running example above. To compute the set of π-patterns of size $[2, \ldots, L]$ occurring at least q times in the input text T, together with their occurrence lists the algorithm makes use of the auxiliary signature array A having size $|\Sigma'|$ ($|\Sigma'| = 2^{\lceil \log_2 |\Sigma| \rceil}$). Lines 1–8 initialize the L_0 list of quadruplets. The quadruplets in L_0 represent how the signature array A is modified (the value of the modified element, and its position in A), every time that we read a symbol of T. More precisely, by scanning each substring $T[i \ldots r]$ of size $[1, \ldots, L]$, we append to L_0 one quadruplet (n_c, c, i, r) for each substring. The first value of the quadruplet represents the number of occurrences of the symbol $T[r]$ inside the substring $T[i \ldots r]$ ($occs(T[r], T[i \ldots r])$). Note that we assumed that each symbol in T is representable as an integer through the function $int(T[r])$.

After that L_0 has been initialized, the main loop of lines 9–32, performs $\log |\Sigma'|$ iterations to compute the fingerprints of the roots of all the binary tagging trees corresponding to the strings of T of size $[1, \ldots, L]$. The computation proceeds levelwise. The k-th iteration computes the list of fingeprints L_k of level k by assigning a unique fingerprint to the pair of fingerprints of level k that are contained in L_{k-1} (we recall that level 0 is that of the leaves, while level $\log |\Sigma'|$ is that of the roots of the binary tagging tree).

This computation is organized in two steps. First in lines 13–24 we construct

Algorithm 3 Levelwise binary tagging tree.

Input: The input string T of size n, the maximum π-patterns size L, the quorum threshold q.

Out: The set of π-patterns of size $[2, \dots, L]$ occurring at least q times in T, together with their occurrence lists.

```
 1: $L_0 = [\,]$
 2: for $i$ in $0, \dots, n-1$ do
 3:     for $r$ in $i, \dots, \min\{n-1, i+L-1\}$ do
 4:         $c = int(T[r])$
 5:         $n_c = occs(T[r],\ T[i \dots r])$
 6:         $append(L_0,\ (n_c, c, i, r))$
 7:     end for
 8: end for
 9: for $k$ in $1, \dots, \log|\Sigma'|$ do
10:     $A = [0, \dots, 0]$
11:     $P_k = [\,]$
12:     $j = i$
13:     for $(h, c, i, r)$ in $L_{k-1}$ do
14:         $A[c] = h$
15:         if $odd(c)$ then
16:             $append(P_k,\ (A[c-1], A[c], \frac{c-1}{2}, i, r))$
17:         else
18:             $append(P_k,\ (A[c], A[c+1], \frac{c}{2}, i, r))$
19:         end if
20:         if $j - i + 1 == L$ or $j == n-1$ then
21:             $reset(A)$
22:         end if
23:         $j++$
24:     end for
25:     Group $P_k = [(h_l, h_r, c, i, r)]$ by the pair $(h_l, h_r)$.
26:     Assign to each group a new name.
27:     $L_k = [\,]$
28:     for $(h_l, h_r, c, i, r)$ in $P_k$ do
29:         Let $h_f$ the name of the group for the pair $(h_l, h_r)$
30:         $append(L_k,\ (h_f, c, i, r))$
31:     end for
32: end for
33: Group $L_{\log|\Sigma'|} = [(h, c, i, r)]$ by the fingerprint $h$.
34: Discard the groups corresponding to strings of size 1.
35: Discard groups with less than $q$ quadruplets.
36: return remaining groups.
```

the list P_k of all non-overlapping consecutive fingerprints that occur in the signature array A, by storing each pair of consecutive fingerprint $A[c-1]$ and $A[c]$ as the first two elements of a quintuplet of P_k (if the index c is even, the consecutive fingerprints are $A[c]$ and $A[c+1]$). In line 20 the signature array A is reset to $[0, \ldots, 0]$, when the sliding window that we are conceptually moving across the input text T is moved one position to the right. This happens when the current substring $T[i \ldots j]$ has reached the maximum size L, or when it cannot be extended to the right because j point to the last symbol of T. In Figure 3.5, this is required after the quadruplet of L_k in positions $3, 7, 11, 15, 19, 23, 26, 28, 29$ have been processed.

After the list P_k has been computed, we group its quintuplets by their first two components, assigning to each group a new fingerprint. These new fingerprints are used in lines 28–31 to rename the quintuplets in P_k, by creating the list L_k of the fingerprints at level k.

After $\log |\Sigma'|$ iterations, the list $L_{\log |\Sigma'|}$ contains the fingerprints of all the substrings of size $[1, \ldots, L]$ in the string T. Substrings have the same fingerprint, if and only if they have the same signature.

By grouping the quadruplets in $L_{\log |\Sigma'|}$ by their fingerprint (the first component), and successively filtering the strings having size 1, and the groups containing less than q quadruplets, we obtain the set of π-patterns of size $[2, \ldots, L]$ occurring at least q times in T, together with their occurrence lists.

Since Algorithm 3.5 mimics the steps of the algorithm in Section 3.3.1, by computing the fingerprints of the node of the binary tagging trees levelwise, the correctness of the algorithm follows from that of the algorithm in Section 3.3.1.

Theorem 12. *Algorithm 3 correctly computes the set of π-patterns of size $[2, \ldots, L]$ occurring at least q times in $T[0 \ldots n-1]$, together with their occurrence lists.*

The complexity of the algorithm is analyzed in the following theorem.

Theorem 13. *Algorithm 3 requires $\mathcal{O}(L \log |\Sigma|\, n)$ time and $\mathcal{O}(Ln)$ space.*

Proof. In the following we assume that $|\Sigma| < n$, recalling that Σ' is at most twice larger than Σ. The initialization of L_0 can be implemented in $\mathcal{O}(nL)$ time, by using an auxiliary array of size Σ' to implement the function *occs*$(-)$. To prove that the time complexity of the algorithm is $\mathcal{O}(L \log |\Sigma|\, n)$ we prove that each iteration requires $\mathcal{O}(Ln)$ time. First, we observe that all the lists L_k and P_k have size Ln, because this is by construction the size of L_0, and all the other lists have one tuple for each quadruplet in L_0. Then we observe that, the value of the fingerprints in the first two components of the quintuplets in P_k is bounded by $\mathcal{O}(n)$. In fact, in each quintuplet $(h_l, h_r, c, i, r) \in P_k$, h_l and h_r are fingerprints that have been generated at the previous iteration. Theorem 10 guarantees that the number of distinct fingerprints at each level is $\mathcal{O}(n)$. Hence, the grouping step in Line 25, and the following initialization of the list L_k, can be implemented in $\mathcal{O}(Ln)$ time by lexicographically sorting the quintuplets in P_k by radix sort.

Some care is required in Line 20 to reset the signature array A every time that the sliding window is moved one position to the right. We do not reset all the components of A, but only the L components (at most) that have been updated from the last reset operation, paying one operation for each one of the Ln quadruplet in L_k.

Both the final grouping step in line 33 (that can be implemented by radix sort), and the final filtering step, require time linear in the size of the list $L_{\log \Sigma'}$, namely $\mathcal{O}(Ln)$. It follows that the overall time complexity of the algorithm is $\mathcal{O}(L \log |\Sigma|\, n)$.

The $\mathcal{O}(Ln)$ space complexity follows from the fact that all the L_k and P_k lists have $\mathcal{O}(Ln)$ size, and that the number of distinct fingerprints for a fixed substring length $l \in [1, \ldots, L]$ at each level is $\mathcal{O}(n)$. From above, it follows that the initialization step at Lines 1–8, each grouping step at Line 25, together with the final grouping step at Line 33 can be performed in $\mathcal{O}(Ln)$ space. Since we assumed that the size of the auxiliary signature array A is $|\Sigma'| \leq n$, we have that the space complexity of the algorithm is $\mathcal{O}(Ln)$. $\square$

3.4 Ranking π-patterns with No Repeated Symbols

In Section 3.3.2 we improved the algorithm in [9], to compute the set of π-patterns of size $[2, \ldots, L]$ occurring at least q times in the input text T, together with their occurrence lists, in $\mathcal{O}(L \log |\Sigma|\, n)$ time, and $\mathcal{O}(Ln)$ space, by saving a $\log(n)$ time factor. This shows that the π-patterns discovery task can be efficiently performed, still preserving the information about the symbol multiplicities occurring inside each pattern.

We now discuss the second step of Algorithm 2. Namely, given a π-pattern p, together with its occurrence list $\mathcal{L}(p)$, we have to compute the minimal PQ-tree T for its occurrence list, counting the number of frontiers represented by T, $\mathit{count}(T) = |\mathit{Fr}(T)|$, because, as discussed in Section 3.2, they are a clear measure of how much conserved p is across its occurrences.

More precisely, in the current section we discuss the case when no symbol in p is repeated (i.e. p is a set). We discuss the more general case of multisets in Section 3.5.4, where we also discuss the relationship between these problems, and the generalization of the classic *consecutive ones problem* to the case of multisets.

Under the assumption that each leaf of the given PQ-tree T is labeled with a distinct symbol of Σ, the computation of $\mathit{count}(T)$ is simple. In fact, if all the leaves are labeled with distinct symbols there is a one-to-one correspondence between the number of frontiers $\mathit{count}(T)$, and the number of PQ-trees equivalent to T, that can be obtained by permuting the children of a P-node, or reversing the order of the children of a Q-node.

We count the number of equivalent PQ-trees inductively. A leaf is only equivalent to itself. In the case of a Q-node with l children $u_1, \ldots, u_l$, since we have to select one of the possible rearrangements for each one of the l subtrees and then concatenating their frontiers either left-to-right or right-to-left, the number of equivalent trees is $2 \times \mathit{count}(u_1) \times \ldots \times \mathit{count}(u_l)$. In the case of a P-node each one of the $l\,!$ possible permutations of the subtrees is suitable. It follows that the $\mathit{count}(u)$ function can be defined as:

$$\mathit{count}(u) = \begin{cases} 1 & \text{if } u \text{ is a leaf} \\ 2 \times \mathit{count}(u_1) \times \ldots \times \mathit{count}(u_l) & \text{if } u \text{ is a } Q \text{ node} \\ l\,! \times \mathit{count}(u_1) \times \ldots \times \mathit{count}(u_l) & \text{if } u \text{ is a } P \text{ node} \end{cases}$$

Theorem 14. *If the leaves of the PQ-tree T are labeled with distinct symbols, then $\mathit{count}(T)$ can be computed in time linear in the number of nodes of T.*

If the π-pattern p contains no repeated symbols, then the computation of the minimal PQ-tree for $\mathcal{L}(p)$ is simple. Note that since no symbol is repeated in p, $\mathcal{L}(p)$ is a list of permutations of the symbols in p.

In [33], Booth and Leuker defined an efficient algorithm to compute the function $\mathit{reduce}(T', C)$, that given a collection C of subsets of Σ, and a PQ-tree T' whose leaves are labeled with the symbols of Σ, builds a PQ-tree T such that $s \in \mathit{Fr}(T)$ iff $s \in \mathit{Fr}(T')$ and each subset in C occurs as a consecutive substring of s. In other words, the function $\mathit{reduce}(-,-)$ transforms the tree T', in a new tree T, such that the strings belonging to $\mathit{Fr}(T)$ satisfy all the consecutiveness constraints in the collection C (i.e. $\{\mathtt{a}, \mathtt{c}, \mathtt{d}\} \in C$ means that the symbols $\mathtt{a}$, $\mathtt{c}$, and $\mathtt{d}$ are required to be consecutive in the frontiers of T).

This function can be used as the main building block to compute the minimal PQ-tree for the occurrence list $\mathcal{L}(p)$, starting from the *universal PQ-tree* T_U that represents the set of all the permutations of the symbols in Σ (T_U consists of a single P-node with $|\Sigma|$ children that are the leaves labeled with the symbols of $|\Sigma|$). For example, in Figure 3.3, the PQ-tree T_2 represent all the permutations of $S = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{e}\}$ where the symbols $\mathtt{b}$,$\mathtt{c}$, and $\mathtt{e}$ are consecutive. It can be constructed starting from the universal tree $T_U = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{e}\}$, which represents all the permutations of the symbols $\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{e}$, by constraining, through the *reduce* function, the symbols $\mathtt{b}$,$\mathtt{c}$, and $\mathtt{e}$ to be consecutive (i.e. $T_2 = \mathit{reduce}(T_U, \{\{\mathtt{b}, \mathtt{c}, \mathtt{e}\}\})$).

The idea behind Algorithm 4, that given $\mathcal{L}(p)$ computes the minimum PQ-tree, is to transform T_U, into the tree whose frontiers satisfy all the consecutive constraints represented by the common intervals of $\mathcal{L}(p)$. The algorithm can be formalized as follows.

Since the common intervals of $\mathcal{L}(p)$ can be succinctly represented by the set of irreducible intervals [93], Algorithm 4 uses this set instead of the common intervals, due to its linear size in $|p|$.

The correctness of the algorithm is discussed in [122], where it is also shown that its time complexity is $\mathcal{O}(|\mathcal{L}(p)||p| + |\Sigma|^2)$, while $\mathcal{O}(|p|)$ space is required. In the same

```
Algorithm 4 Computation of the minimal PQ-tree.
Input: The $\pi$-pattern $p$ with no repeated symbol, and its occurrence list $\mathcal{L}(p)$.
Out: The minimal PQ-tree $T$ for $\mathcal{L}(p)$.
1: Compute the set of Irreducible Intervals $I$ for $\mathcal{L}(p)$ [93].
2: Compute $T = reduce(T_U, I)$ [33].
3: return $T$
```

paper a more involved algorithm running in $\mathcal{O}(|\mathcal{L}(p)||p|)$ time and $\mathcal{O}(|p|)$ space is also presented.

From above, it follows that:

Theorem 15. *Given a π-pattern p with no repeated symbols, together with its occurrence list $\mathcal{L}(p)$, the minimum PQ-tree T for $\mathcal{L}(p)$, and the number of frontiers represented by T, $count(T)$, can be computed in $\mathcal{O}(|\mathcal{L}(p)||p|)$ time and $\mathcal{O}(|p|)$ space.*

3.5 Ranking π-patterns with Repeated Symbols

In Section 3.4 we discussed the complexity of constructing the minimal PQ-tree T for the occurrence list $\mathcal{L}(p)$ of the π-pattern p with no repeated symbols, and the complexity of counting the number of frontiers represented by T, $count(T)$. We showed that in the restricted case when the π-pattern p is a set of symbols (i.e. no symbol is repeated) the above tasks can be performed in $\mathcal{O}(|\mathcal{L}(p)||p|)$ time, and $\mathcal{O}(|p|)$ space.

We now discuss the case of π-patterns with repeated symbols (i.e. p is a multiset). The results presented in the following sections wipe out the possibility of an efficiently implementation of the $score(p, \mathcal{L}(p))$ function introduced in Section 3.2, also in the case of π-patterns with repeated symbols.

More precisely, in Section 3.4 we described the main conceptual steps involved in the computation of the minimal PQ-tree T given the occurrence list $\mathcal{L}(p)$, of the π-pattern p. First, we compute the set of common intervals of $\mathcal{L}(p)$ (that is succinctly represented by the set I of the irreducible intervals). Then, given the set of irreducible intervals we compute the minimal PQ-tree T. Finally, we count the number of frontiers represented by the PQ-tree T, $count(T)$.

In Section 3.5.4, our first result is to prove that the problem (denoted #FRONT) of counting the frontiers of a PQ-tree whose leaves are labeled with the (repeated) symbols of a multiset is $\#\mathcal{P}$-complete. We recall that, the complexity of this problem has been left as an open issue in [155].

One could hope that a polynomial solution to compute $count(T)$ might exist without relying on PQ-trees. In other words, given the set of irreducible intervals of $\mathcal{L}(p)$, can we directly compute the number of frontiers represented by the minimal PQ-tree without computing the tree itself?

To answer to this question we introduced the #FMO problem: given a *multiset* of symbols R, and a family of *multisets* $F = \{Q_1, \ldots, Q_m\}$ such that $Q_i \subset R$ for $1 \leq i \leq m$, how many strings x can be drawn from *all* symbols in R, respecting the multiplicities of the symbols in R, so that each Q_i occurs as a consecutive substring in x?

Note that the problem of computing the number of frontiers of the minimal PQ-tree given the set of irreducible intervals is a particular instance of the #FMO problem, where $R = p$, and F is the family of the irreducible intervals of $\mathcal{L}(p)$. Note also that in the #FMO problem, both R and Q_is are multisets. This is the fundamental difference between the classic *consecutive ones property* (C1P), and #FMO, which is a generalization of C1P to multisets.

A family of sets $F = \{Q_1, \ldots, Q_m\}$, where each Q_i is a subset of the alphabet Σ, satisfies the C1P if the symbols in Σ can be permuted such that the elements of each set $Q_i \in F$ occur consecutively as a contiguous segment of the permutation. The counting version of the problem consists in counting the number of such permutations.

Booth and Leuker [32, 33] showed how to enumerate and count all the solution permutations of a given C1P instance in time linear in the sum of the sizes of the Q_i sets (see Section 3.5.2 for alternative approaches). Can we also do the same in the case of multisets?

Our second result is that #FMO is $\#\mathcal{P}$-complete. We refer the reader to Section 3.5.5 for a discussion.

An interesting implication of our findings is the relation with the well-known counting version #HAM of the Hamiltonian path problem [75]. As previously mentioned, a *direct* mapping of the orderings for the C1P in multisets into the frontiers of PQ-trees has some intrinsic ambiguity. On the other hand, we can prove that both the counting problems #FRONT and #FMO are $\#\mathcal{P}$-complete using a reduction from #HAM. By the completeness properties, it follows that there must exist a relation between the latter two problems. However, simply composing the reductions does not immediately produce a single PQ-tree, having the same properties as the input instance, which is an open problem. Our approach is of independent interest, and the counting nature of the problems emphasizes the combinatorial properties of the strings (orderings) thus generated by the reductions.

However, before discussing the complexity of #FRONT and #FMO, in Section 3.5.1 and 3.5.2 we formally define the consecutive ones problem, showing its relationship with other well known problems and the PQ-tree data structure. The terminology that will be used in the following sections will be introduced in Section 3.5.3.

3.5.1 Introduction to C1P

A binary matrix M of size $m \times n$ satisfies the *consecutive ones property* (C1P) if its n columns can be permuted such that the 1s in each row of the resulting matrix are consecutive. An equivalent definition holds for the columns by permuting the rows. The property is often formulated in terms of sets. A family of sets $F = \{Q_1, \ldots, Q_m\}$, where each Q_i is a subset of the universe of symbols $R = \{r_1, \ldots, r_n\}$, satisfies the C1P if the symbols in R can be permuted such that the elements of each set $Q_i \in F$ occur consecutively as a contiguous segment of the permutation of the symbols in R.

For example, consider the universe $R = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}\}$. The C1P is not satisfied by the family $F = \{\{\mathtt{a}, \mathtt{b}\}, \{\mathtt{b}, \mathtt{c}\}, \{\mathtt{b}, \mathtt{d}\}\}$, since `b` can have at most two adjacent symbols in any permutation of R. On the other hand, the family $F = \{\{\mathtt{b}, \mathtt{c}\}, \{\mathtt{b}, \mathtt{d}\}\}$ satisfies the C1P: one feasible permutation of R is $x = \mathtt{eacbd}$, but not all permutations of R are feasible (e.g. $y = \mathtt{abcde}$ is not, because the symbols $\{\mathtt{b}, \mathtt{d}\}$ are not consecutive in y).

The C1P on sets can be formulated as a C1P problem on the binary matrix M obtained by associating row i with set $Q_i \in F$, and column j with element $r_j \in R$. Specifically, $M_{ij} = \mathtt{1}$ iff $r_j \in Q_i$, as shown below for our example.

$$\begin{array}{lccccc}
 & \mathtt{a} & \mathtt{b} & \mathtt{c} & \mathtt{d} & \mathtt{e} \\
\{\mathtt{b}, \mathtt{c}\} & \mathtt{0} & \mathtt{1} & \mathtt{1} & \mathtt{0} & \mathtt{0} \\
\{\mathtt{b}, \mathtt{d}\} & \mathtt{0} & \mathtt{1} & \mathtt{0} & \mathtt{1} & \mathtt{0}
\end{array}
\qquad\qquad
\begin{array}{ccccc}
\mathtt{e} & \mathtt{a} & \mathtt{c} & \mathtt{b} & \mathtt{d} \\
\mathtt{0} & \mathtt{0} & \mathtt{1} & \mathtt{1} & \mathtt{0} \\
\mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{1} & \mathtt{1}
\end{array}$$

The problem of finding the *orderings*, namely, the permutations of R that are generated by the C1P, arises in several situations. It was first solved efficiently by Fulkerson and Gross [74] in their study on the incidence matrix of interval graphs, using an $O(mn^2)$ time algorithm. Ghosh [76] applied the problem to information retrieval, where R is the set of input records and each Q_i is the set of records satisfying a query: for each Q_i, the C1P guarantees that the corresponding records can be retrieved from consecutive storage locations. Booth and Leuker [32, 33] showed how to find any such ordering in linear time, with respect to the number of `1`s in M, with applications to some graph problems such as planarity testing. They employed the *PQ-tree* data structure to represent *compactly all the orderings* yielding the C1P for the given matrix M.

The PQ-tree corresponding to our example is denoted by T_1 in Figure 3.6. The leaves of the PQ-tree contain the symbols of R: when reading these symbols by traversing the leaves in preorder, we obtain a string that is one of the frontiers of the PQ-tree. As it can be seen, the frontier is one of the orderings yielding the C1P in our example tree T_1. Further orderings can be obtained by rearranging the children of the nodes of the PQ-tree, since they implicitly encode the sets in F.

By conceptually performing all the feasible rearrangements of the nodes in the PQ-tree as discussed in Section 3.2, we obtain the set of frontiers that are generated by the PQ-tree. These frontiers are in *one-to-one* correspondence with all the orderings yielding the C1P for matrix M, as it can be verified by inspecting our example for T_1: we can represent them as the strings $x_1 = \mathtt{acbde}$, $x_2 = \mathtt{adbce}$, $x_3 = \mathtt{aecbd}$, $x_4 = \mathtt{aedbc}$, $x_5 = \mathtt{cbdae}$, $x_6 = \mathtt{dbcae}$, $x_7 = \mathtt{cbdea}$, $x_8 = \mathtt{dbcea}$, $x_9 = \mathtt{ecbda}$, $x_{10} = \mathtt{edbca}$, $x_{11} = \mathtt{eacbd}$, and $x_{12} = \mathtt{eadbc}$.

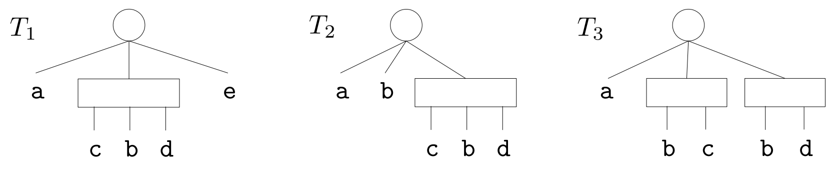


Figure 3.6: Some examples of PQ-trees.

Since its inception, the C1P has found many applications under several incarnations. Recent fields of application are stringology and bioinformatics. Motivated by the combinatorial aspects of sequences with repeated symbols, we consider the scenario for the C1P in which the symbols in the input set R are *not* necessarily distinct.

We investigate the problem of how to satisfy the C1P when R and the Q_is are *multisets*. To get the flavor of the problem, consider the universe $R = \{\mathtt{a}, \mathtt{b}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$ and the family $F = \{\{\mathtt{b}, \mathtt{c}\}, \{\mathtt{b}, \mathtt{d}\}\}$. The situation arises from the fact that the symbol b in both $Q_1 = \{\mathtt{b}, \mathtt{c}\}$ and $Q_2 = \{\mathtt{b}, \mathtt{d}\}$ can either match the same occurrence of b in R or not. The former case gives rise to the PQ-tree T_2 in Figure 3.6, while the latter gives rise to the PQ-tree T_3. The set of frontiers are now strings with repeated symbols: the set of frontiers generated by one PQ-tree is *not* contained in the set of the other PQ-tree. However, the two occurrences of b in R are *indistinguishable*.

In the following sections, we consider problems arising from repeated symbols, and show that dealing with the C1P on multisets is hard. Specifically, we study the problem of *counting* the number of orderings. This is “simpler” than listing all the orderings. As discussed in Section 3.2 the counting problem using standard PQ-trees on *sets* takes polynomial time, since we can use the aforementioned one-to-one correspondence between the orderings and the frontiers.

3.5.2 Testing the C1P: related work

Testing the C1P can be done using variants of the PQ-tree data structure. Although optimal from a theoretical viewpoint, Booth and Leuker’s algorithm [33] is quite difficult to implement since it builds the PQ-tree by induction on the number of rows of the matrix. For each row, it performs a second induction from the leaves towards the root, using one of nine templates at each node encountered in order to understand how the other nodes must be restructured.

The *PC-tree* is an alternative data structure introduced by Shih and Hsu in [174] to address these difficulties, which can also be used to check the C1P as shown in [99]. Both the above tree structures have remarkably simple definitions as mathematical objects applying previously-known theorems on set families to this domain. Also, the PC-tree gives a representation of the *circular ones orderings* of the matrix M just as the PQ-tree gives a representation of all the C1P orderings.

The *PQR-tree* is another alternative data structure introduced by Meidanis et al. [138] to devise a tree also for the case when the input does not satisfy the C1P. In particular, the R-node is like the P-node, except that it captures the portion of the frontier that violates the C1P.

As previously mentioned, the C1P has several interesting applications since several apparently unreleated problems reduce to it. One of such problems is to decide if a given graph G is an *interval graph*: in [74] the authors proved that a graph G is an interval graph if and only if its *clique matrix* has the C1P by rows.

Another important application is in graph *planarity testing*: given a graph G return a planar embedding for G and if it does not exist return a Kuratowski subgraph isolator [121]. In this case, the C1P is used as a step in the Booth and Lueker algorithm [33] to check planarity in linear time: this approach adds one vertex at a time, updating the PQ-tree to keep track of possible embeddings of the subgraph induced by vertices so far. (A much more simpler approach based on PC-tree has been developed in [174].)

However not all pairs of F and R enjoy the C1P. In that case, either duplication of symbols, or "breaking" some set in F into subsets, must be allowed in order to arrange linearly the input symbols. The former scenario gives rise to the problem of minimizing duplication of symbols. The latter gives rise to the problem of minimizing the number of subsets the input sets are splitted into (sometimes referred in literature as the *consecutive block minimization* problem). Both problems, in their decision version, have been proved in [117] to be $\mathcal{NP}$-complete (an 1.5 approx algorithm for the block minimization problem is described in [89]). For example, the C1P instance where $R = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$ and $F = \{\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}, \{\mathtt{a}, \mathtt{c}, \mathtt{d}\}, \{\mathtt{b}, \mathtt{d}\}\}$ has no solution. If we allow duplication of symbols, two strings satisfying the constraints in F are $x = \mathtt{bacdb}$ and $y = \mathtt{dbacd}$ (where $\mathtt{b}$ and $\mathtt{d}$ are repeated twice in x and y respectively), while if we allow some constraints not being satisfied an optimal solution is $z = \mathtt{bacd}$ where only the set $\{\mathtt{b}, \mathtt{d}\}$ is broken into two subsets $\{\mathtt{b}\}$ and $\{\mathtt{d}\}$.

3.5.3 Definitions and terminology

We consider a class of strings defined over multisets, where the usual notions of inclusion, equality, and union, take into account the multiplicities of the elements in the multisets. We say that a string $s \equiv s_1 s_2 \cdots s_n$ is drawn from a multiset R of symbols if and only if the multiset $S = \{s_1, s_2, \ldots, s_n\}$ satisfies the condition $S \subseteq R$, where s_i denotes the symbol stored into position i of s, for $1 \leq i \leq n$.

We also say that a *multiset* P *occurs* in a *string* s (or equivalently P is *contained* in s), if there is a substring $s_i s_{i+1} \cdots s_j$ of s, where $1 \leq i, j \leq n$, such that $P = \{s_i, s_{i+1}, \ldots, s_j\}$.[1] In the latter case, we say that P occurs at position i in s (and

[1]In order to simplify the notation, we will always assume that an index i is well defined, without explicitly writing its range when it can be deduced from the context. For example, a nonempty substring $s_i s_{i+1} \cdots s_j$ has $1 \leq i \leq j \leq n$.

P is called π-pattern [9]). For example, $P = \{\mathtt{a}, \mathtt{c}, \mathtt{a}\}$ occurs at position $i = 1$ in $s = \mathtt{aacb}$, while P is not contained in $s_2 = \mathtt{aabc}$.

In the following sections we also use the notion of Sperner collection [63]. A collection of multisets $Q_1, Q_2, \ldots, Q_m \subset R$ is a *Sperner Collection* (or Sperner Family, or Sperner System) if it is an anti-chain in the inclusion lattice over the powerset of R; namely, no multiset Q_i is contained in any other multiset Q_j of the collection ($i \neq j$). If no set Q_i is contained in the union of the others, $\cup_{j \neq i} Q_j$, then the Sperner Collection is said to be *strict*.

Given a decision problem $\mathcal{A}$, we will denote by $\#\mathcal{A}$ its *counting version*, where we are required to count the number of the solutions of $\mathcal{A}$ [182]. A formal description of the $\#\mathcal{P}$ class is beyond the scope of this section, and we refer the interested reader to the textbooks in [11, 75, 153]. However, we are going to use the notion of $\#\mathcal{P}$-completeness to address the difficulty of our combinatorial problems, and so we recall some basic definitions.

Let f be an integral function defined over strings in Σ^*, for a given alphabet Σ. We say that $f \in \#\mathcal{P}$ if there exists a binary relation $T(-,-)$ such that:

- There exists a polynomial p such that, if $(y, x) \in T$, then $|x| \leq p(|y|)$.
- It can be verified in polynomial time that a pair (y, x) belongs to T.
- For every input $y \in \Sigma^*$, $f(y) = |\{x : (y, x) \in T\}|$ is the number of solutions for y.

Given two integral functions f, g defined over Σ^*, we say that there exists a *polynomial Turing reduction* from g to f if the function g can be computed in polynomial time by using a (polynomial) number of calls to an oracle for f. The reduction is *parsimonious* if it preserves the number of solutions.[2] A function f is $\#\mathcal{P}$*-hard* if for every $g \in \#\mathcal{P}$ there is a polynomial reduction from g to f. As usual, a function is $\#\mathcal{P}$*-complete* if it is both $\#\mathcal{P}$-hard and in $\#\mathcal{P}$.

We are now ready to introduce the #FMO problem, which formalizes the problem of extending the Booth-Leuker approach [33] for the C1P to multisets.

Problem 4 (#FMO = Counting Full Multiset Orderings). *Input: an instance $\langle R, F \rangle$, where R is a multiset of symbols, and $F = \{Q_1, \ldots, Q_m\}$ is a family of multisets $Q_i \subset R$. Output: how many strings x can be drawn from* all *symbols in R ($|x| = |R|$), so that each Q_i is contained in x?*

For example, given $R = \{\mathtt{a}, \mathtt{b}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$ and $F = \{\{\mathtt{b}, \mathtt{c}\}, \{\mathtt{b}, \mathtt{d}\}\}$, $x = \mathtt{abcbd}$, is one of the feasible solutions of the $\langle R, F \rangle$ #FMO instance.

We now introduce our second problem, related to Problem 4, where we are required to count the number of strings represented by the input PQ-tree.

[2]Hence it allows for non-emptiness testing in the decisional version of the problems.

Problem 5 (#FRONT = Counting PQ-trees Frontiers)**.** *Input: a PQ-tree T, where its leaves are labeled with symbols that are not necessarily distinct. Output: what is the size of the set of frontiers $Fr(T)$ of T?*

3.5.4 Hardness results for #FRONT

We begin by discussing the completeness of the #FRONT problem. We use a reduction from the well-known counting version of Hamiltonian Path (#HAM). We are given an undirected graph G, a source vertex $w \in G$, and a destination vertex $s \in G$. We want to know how many paths H in G start in w and end in s, such that all the vertices in G are traversed exactly once by each H. For example, one such path is $H = \langle 1, 3, 2, 4, 5 \rangle$ in the graph G shown in Figure 3.7. In the following sections, we assume that G is connected, w and s have degree at least one, and the other vertices have degree at least two (otherwise there is no Hamiltonian path). We also assume that there are no multiple edges between the same pair of vertices and no self-loops.

Construction of the PQ-trees

The main idea is to code the structure of the given graph G in three suitable PQ-trees, T_G, T_V, and T_E, such that each Hamiltonian path H is in one-to-many correspondence with a suitable set of strings from their frontiers. We now describe our reduction from $G = \langle V, E \rangle$ to T_G, T_V, and T_E, using Figure 3.7 as an illustrative example.

The root of T_G is a Q-node having two PQ-trees T_V and T_E as children.

Tree T_E encodes all the feasible permutations of the edges in E. The root of T_E is a P-node having $|E| + 2$ children. Two of them are special "endmarkers," and are labeled with \$ and #. Each of the remaining children is a Q-node that encodes an edge $e = \{i, j\}$ by two leaves labeled with i and j, respectively, as children. In our example, T_E has $|E| = 7$ Q-nodes with children labeled by $\{1, 2\}$, $\{1, 3\}$, $\{1, 4\}$, $\{2, 3\}$, $\{2, 4\}$, $\{3, 4\}$, and $\{4, 5\}$, plus the endmarkers \$ and #.

Tree T_V enforces a classification of the edges as "coding" a Hamiltonian path, or "non-coding" otherwise. Specifically, the root of T_V is a Q-node with four children: one leaf labeled with \$, a PQ-tree T_C for the coding edges, one more leaf labeled with #, and a PQ-tree T_N for the non-coding edges. The root of T_C is a Q-node with three children. The first child is a leaf labeled with the source w and the last is a leaf labeled with the destination s. The middle child is a P-node with $|V| - 2$ children, each of which is a Q-node with two leaves labeled with the same symbol i, for $i \in V \setminus \{w, s\}$. In our example $w = 1$, $s = 5$, and $|V| = 5$. The root of the non-coding tree T_N is a P-node having $2(|E| - |V| + 1)$ leaves as children. Letting d_i denote the degree of vertex i, there are $d_w - 1$ leaves labeled with w, $d_s - 1$ leaves labeled with s, and $d_i - 2$ leaves labeled with $i \neq w, s$. In our example, the leaves are labeled with $1, 1, 2, 3, 4, 4$, where $2(|E| - |V| + 1) = 6$.

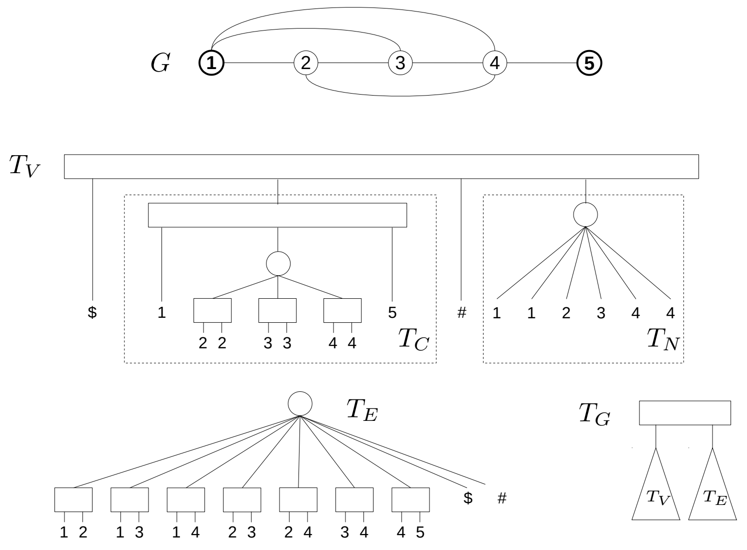


Figure 3.7: The PQ-tree T_G associated with the input graph G, where the source and the destination vertices are $w = 1$ and $s = 5$, and T_V and T_E are shown individually.

The above construction requires polynomial time, and the rationale will be given in Section 3.5.4.

Lemma 16. *Given an undirected graph $G = \langle V, E\rangle$, its corresponding PQ-trees T_G, T_V, and T_E can be built in $\mathcal{O}(|V| + |E|)$ time.*

Properties of the PQ-trees

Consider the Hamiltonian path $H = \langle 1, 3, 2, 4, 5\rangle$ in our example. (Observe that the reversal of H, namely $\langle 5, 4, 2, 3, 1\rangle$, is also a Hamiltonian path, but we consider it to be different from H for counting purposes.) The corresponding strings α^H belonging to the frontiers $Fr(T_G)$ are characterized as follows. First of all, each α^H is a square, namely, the concatenation $\alpha^H = \alpha\,\alpha$ of two equal strings α, where α belongs to both the frontiers $Fr(T_V)$ and $Fr(T_E)$, and is of length $2|E| + 2$. For example, $\alpha = \$13322445\#121434$ is one such feasible string. We can characterize the general structure of the strings α by observing that they match one of the following two patterns. Let π denote an arbitrarily chosen permutation of the pairs in $\{1, 2\}, \{1, 4\}, \{3, 4\}$, which represents the edges *not* traversed by H. (That is, π belongs to the frontiers of the PQ-tree resulting from $\{\{1, 2\}, \{1, 4\}, \{3, 4\}\}$.) The former pattern for α is $\$\,13322445\,\#\,\pi$, where the initial symbols are fixed and only π may vary; analogously, the latter is $\pi\,\#\,13322445\,\$$. For example, $\alpha = 413421\,\#\,13322445\,\$$ matches the latter pattern.

Having introduced the structure of $\alpha^H = \alpha\,\alpha$ in our example, we show how to make α satisfy the implicit conditions encoded in T_V and T_E. Indeed, T_E guarantees that the two integers in each of the pairs corresponding to the edges in E always occur consecutively in α. Moreover, the subtree T_C in T_V constraints each vertex $i \in V \setminus \{w, s\}$ to appear exactly twice in the chosen subset of edges, while w and s are required to appear just once. Note that the purpose of the subtree T_N is that of "padding" the edges in E that are not traversed by H, since we do not know a priori which ones will be touched by H.

We now generalize the above observations on α. In the following we can restrict our focus on paths of the form $i_1, i_2, \ldots, i_{|V|}$, that are permutations of $\{1, 2, \ldots, |V|\}$ with $i_1 = w$ and $i_{|V|} = s$ (otherwise they cannot be Hamiltonian paths from w to s). Moreover, we introduce the notation $\mathit{Perm}(Q)$ for a set $Q = \{\{a_1, b_1\}, \{a_2, b_2\}, \ldots, \{a_r, b_r\}\}$ of unordered pairs. It represents the set of all the permutations of $a_1, b_1, a_2, b_2, \ldots, a_r, b_r$ such that a_l and b_l occupy contiguous positions for $1 \leq l \leq r$. For example, given $Q = \{\{1, 2\}, \{1, 4\}, \{3, 4\}\}$, we have that 413421 is a valid permutation in $\mathit{Perm}(Q)$, while 413241 is not.

We now show in Lemmas 17–19 that there exists a one-to-many correspondence between the Hamiltonian path H in G and the strings $\alpha \in \mathit{Fr}(T_V) \cap \mathit{Fr}(T_E)$.

Lemma 17. *Let $G = \langle V, E\rangle$ be an undirected graph, and T_G, T_V, and T_E be its corresponding PQ-trees. For any string $\alpha \in \mathit{Fr}(T_V) \cap \mathit{Fr}(T_E)$, there exists a corresponding Hamiltonian path H of G from w to s.*

Proof. Consider a string $\alpha \in \mathit{Fr}(T_V) \cap \mathit{Fr}(T_E)$. We first show that the symbols in α follow a special pattern.

Since $\alpha \in \mathit{Fr}(T_V)$, the symbols \$ and # in it match those in the leaves of T_V by construction. Assume w.l.o.g. that the first symbol of α is \$. (The other case in which \$ is the last symbol of α is analogous.) Then, α is of the form $\alpha = \$\,\tau\,\#\,\pi$ by construction, where $\tau = \tau_1\tau_2 \cdots \tau_{2|V|-2}$ and π should follow the patterns described next. First, $\tau = w\tau' s$ where $\tau' \in \mathit{Perm}(\{i, i\}_{i \neq w,s})$, since $\tau' \in \mathit{Fr}(T_C)$: hence, $\tau_i = \tau_{i+1}$ for even values of $i \in [2 \ldots 2|V| - 4]$. Second, π is a permutation of the symbols in the multiset obtained by removing the symbols of τ from $\bigcup_{\{i,j\} \in E} \{i, j\}$.

Now, the fact that α belongs also to $\mathit{Fr}(T_E)$ puts additional constraints on τ and π. Indeed, the Q-nodes in T_E guarantee that τ_1 and τ_2 are children of the same Q-node, τ_3 and τ_4 are children of the next Q-node, and so on. Thus in general, $\tau_i, \tau_i + 1$ for odd i belong to the same Q-node: hence, $\{\tau_i, \tau_i + 1\} \in E$, for even values of $i \in [2 \ldots 2|V| - 4]$. Combining the latter with the fact that $\tau_i = \tau_{i+1}$ for odd values of i, we obtain that $H = \langle w, \tau_2, \ldots, \tau_{2|V|-4}, s\rangle$ is a Hamiltonian path.

Note that the rest of the Q-nodes in T_E induce also some contiguity constraints on π, which will be relevant later for the counting argument (see Lemma 20). The case $\alpha = \pi\,\#\,\tau\,\$$ is analogous. □

Lemma 18. *Let $G = \langle V, E\rangle$ be an undirected graph, and T_G, T_V, and T_E be its corresponding PQ-trees. For any Hamiltonian path H of G from w to s, there exists at least one corresponding string $\alpha \in \mathit{Fr}(T_V) \cap \mathit{Fr}(T_E)$.*

Proof. Let $H = \langle i_1, i_2, \ldots, i_{|V|}\rangle$ be a Hamiltonian path, where $i_1 = w$ and $i_{|V|} = s$. We define $\alpha = \$\tau\#\pi$ where τ and π are as follows. First, we choose $\tau = i_1 i_2 i_2 \cdots i_{|V|-1} i_{|V|-1} i_{|V|}$, so that $\tau \in Fr(T_C)$. Second, let $E' = E \backslash \{\{i_j, i_{j+1}\}\}_{1\le j\le |V|-1}$ be the set of edges not traversed by H. Let list the edges of E' as $\{a_1, b_1\}, \ldots, \{a_r, b_r\}$. Then we choose $\pi = a_1 b_1 \cdots a_r b_r$, so that $\pi \in Fr(T_N)$.

Consequently, α should belong to $Fr(T_V)$. It remains to see that α belongs also to $Fr(T_E)$. Note that the \$ and # symbols in α clearly match the two endmarker leaves in T_E. Also, by our construction of τ and π, for any edge $\{i, j\}$ in E, we have that i and j appear in consecutive positions of either τ or π. This concludes the proof implying that $\alpha \in Fr(T_V) \cap Fr(T_E)$. □

Lemma 19. *Let $\Sigma_H \subseteq Fr(T_V) \cap Fr(T_E)$ denote the set of all the strings corresponding to a given Hamiltonian path H, as stated in Lemma 18. Then, for any two Hamiltonian paths $H \neq H'$ of G from vertex w to vertex s, it is $\Sigma_H \cap \Sigma_{H'} = \varnothing$.*

Proof. For any $\alpha \in \Sigma_H$ and $\alpha' \in \Sigma_{H'}$, we show that $\alpha \neq \alpha'$. If one of the strings begins with the \$ symbol, while the other does not, they are different since neither τ or π contains any endmarker (e.g. $\alpha = \$\tau\#\pi$ is different from $\alpha' = \pi'\#\tau'\$$). Hence, consider the case when both α and α' begin with \$. Since the corresponding Hamiltonian paths H and H' are different, also the corresponding "coding" strings τ and τ' will be different by construction, implying that $\alpha \neq \alpha'$. □

Reduction from #HAM to #FRONT

We now show how to reduce the problem #HAM of counting the Hamiltonian paths in $G = \langle V, E\rangle$, to the problem #FRONT of counting the frontiers of PQ-trees, namely, T_G, T_V, and T_E. We denote the number of frontiers for a PQ-tree T by $|Fr(T)|$. Here is the polynomial time reduction for the input graph G and its two vertices w and s:

- Build the PQ-trees T_G, T_V, and T_E (see Lemma 16).
- Return the following integer as the number of Hamiltonian paths from w to s in G:

$$\frac{|Fr(T_V) \cap Fr(T_E)|}{|\Sigma_H|} = \frac{2\,|Fr(T_V)| \times |Fr(T_E)| - |Fr(T_G)|}{2\,(|E| - |V| + 1)! \times 2^{|E|-|V|+1}} \tag{3.1}$$

Clearly, the formula in (3.1) can be computed in polynomial time. We now show its correctness.

Lemma 20. *Let $\Sigma_H \subseteq Fr(T_V) \cap Fr(T_E)$ denote the set of strings corresponding to a Hamiltonian path H. Then, for any Hamiltonian path H from w to s, we have $|\Sigma_H| = 2\,(|E| - |V| + 1)! \times 2^{|E|-|V|+1}$.*

Proof. Consider a string $\alpha \in \Sigma_H$. As previously mentioned in the proof of Lemma 17, α matches either the pattern $\$\,\tau\,\#\,\pi$ or $\pi\,\#\,\tau\,\$$. Note that the string τ is uniquely determined by construction of T_C, and the contiguity condition imposed by T_E, for the given H. Hence, $|\Sigma_H|$ is twice the number of strings π that we can obtain from T_N, under the contiguity condition imposed by T_E. Therefore, $|\Sigma_H| = 2\,|Perm(E')|$, where $E' \subseteq E$ is the set of edges not traversed by H. Since $|E'|$ is $p = |E| - |V| + 1$, we have $p!$ permutations of these edges and, for each of them, we have two ways to permute every $\{i, j\} \in E'$. This gives a total of $p!\,2^p$ strings π. Note that we cannot generate twice the same string in this way, because the edges are distinct as unordered pairs and, for each pair $\{i, j\} \in E'$, it is $i \neq j$. Hence the result follows. □

Lemma 21. $|Fr(T_G)| = 2\,|Fr(T_V)| \times |Fr(T_E)| - |Fr(T_V) \cap Fr(T_E)|$

Proof. Let $L_V = Fr(T_V)$, $L_E = Fr(T_E)$, and $L_G = Fr(T_G)$. Consider $L_{VE} = Fr(T_V) \cap Fr(T_E)$, so that we can rewrite $L_V = L'_V \cup L_{VE}$ and $L_E = L'_E \cup L_{VE}$. Now, by construction of T_G, we know that $L_G = L_V \cdot L_E \cup L_E \cdot L_V$, where the standard operation "$\cdot$" denotes the extension of the string concatenation to sets of strings (i.e. $A \cdot B = \{ab \mid a \in A, b \in B\}$). By expanding L_V and L_E, we obtain that $L_G = (L'_V \cup L_{VE}) \cdot (L'_E \cup L_{VE}) \cup (L'_E \cup L_{VE}) \cdot (L'_V \cup L_{VE})$. By simple algebra, we have that $|L_G| = |L_V \cdot L_E| + |L_E \cdot L_V| - |L_E \cap L_V|$. The result follows, since $|L_V \cdot L_E| = |L_E \cdot L_V| = |L_E| \times |L_V|$. □

We now have all the ingredients to prove the $\#\mathcal{P}$-completeness of the #FRONT problem.

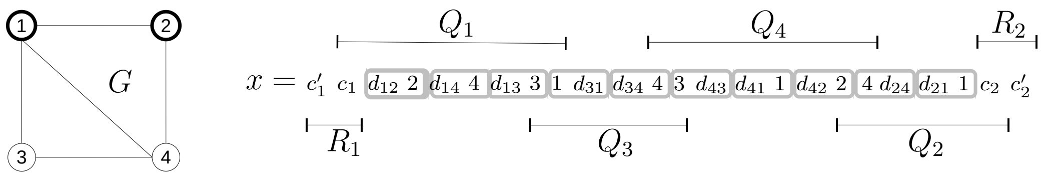
Theorem 22. *#FRONT is $\#\mathcal{P}$-complete.*

Proof. The membership to $\#\mathcal{P}$ trivially holds. In order to prove that the formula in (3.1) is correct, observe that the sets Σ_H for all the Hamiltonian paths H from w to s, are a partition of $I = Fr(T_V) \cap Fr(T_E)$. To see why, note that for each string in I, there is a Hamiltonian path by Lemma 17. Moreover, $\Sigma_H \subseteq I$ by Lemma 18. Finally, the sets Σ_H are pairwise disjoint by Lemma 19.

Formula (3.1) is based on the fact that $|I|$ can be obtained from $|T_G|$, $|T_V|$, and $|T_E|$ by using Lemma 21. Moreover, sets Σ_H have all the same size, as stated in Lemma 20. Hence, dividing these two quantities gives an integer as a result, which is the number of Hamiltonian paths as in (3.1). Note that our reduction requires polynomial time. □

3.5.5 Hardness results for #FMO

We now show how to reduce the #HAM problem to the counting version of the Full Multiset Problem (#FMO). For the given undirected graph $G = \langle V, E\rangle$, together with the source and the destination vertices, w and s, we make the same assumptions as in Section 3.5.4. In Section 3.5.5, we walk through the example in Figure 3.8 to


1
2
3
4
G
Q_1
Q_4
R_2
$x = c'_1\ c_1\ d_{12}\ 2\ d_{14}\ 4\ d_{13}\ 3\ 1\ d_{31}\ d_{34}\ 4\ 3\ d_{43}\ d_{41}\ 1\ d_{42}\ 2\ 4\ d_{24}\ d_{21}\ 1\ c_2\ c'_2$
R_1
Q_3
Q_2

$$
\begin{aligned}
Q_1 &= \{1, d_{12}, d_{13}, d_{14}, 2, 3, 4, c_1\}\\
Q_2 &= \{2, d_{21}, d_{24}, 1, 4, c_2\}\\
Q_3 &= \{3, 3, d_{31}, d_{34}, 1, 4\}\\
Q_4 &= \{4, 4, d_{41}, d_{42}, d_{43}, 1, 2, 3\}\\
R_1 &= \{c_1, c'_1\}\\
R_2 &= \{c_2, c'_2\}
\end{aligned}
\qquad
\begin{aligned}
Q_{12} &= \{d_{12}, 2\}\\
Q_{13} &= \{d_{13}, 3\}\\
Q_{14} &= \{d_{14}, 4\}\\
Q_{24} &= \{d_{24}, 4\}\\
Q_{34} &= \{d_{34}, 4\}
\end{aligned}
\qquad
\begin{aligned}
Q_{21} &= \{d_{21}, 1\}\\
Q_{31} &= \{d_{31}, 1\}\\
Q_{41} &= \{d_{41}, 1\}\\
Q_{42} &= \{d_{42}, 2\}\\
Q_{43} &= \{d_{43}, 3\}
\end{aligned}
$$

$$
\begin{aligned}
F &= \{\ Q_1, Q_2, Q_3, Q_4, R_1, R_2, Q_{12}, Q_{21}, Q_{13}, Q_{31}, Q_{14}, Q_{41}, Q_{24}, Q_{42}, Q_{34}, Q_{43}\ \}\\
R &= \{\ d_{12}, d_{21}, d_{13}, d_{31}, d_{14}, d_{41}, d_{24}, d_{42}, d_{34}, d_{43}, 1, 1, 1, 2, 2, 3, 3, 4, 4, 4, c_1, c'_1, c_2, c'_2\ \}
\end{aligned}
$$

Figure 3.8: Example of reduction from a Hamiltonian Path instance for a graph G, where the source and the destination vertices are $w = 1$ and $s = 2$, into a #FMO instance $\langle R, F\rangle$. Sets Q_{ij} are shown boxed in string x.

describe the reduction. In Section 3.5.5, we characterize the structure of each string satisfying the constraints in the #FMO instance. In Section 3.5.5, we prove our hardness result on counting how many strings correspond to the same Hamiltonian path H in G.

Instance construction

Consider the example in Figure 3.8. On the left we show the input undirected graph G, where the source and the destination vertices $w = 1$ and $s = 2$ are in boldface. The corresponding #FMO instance $\langle R, F\rangle$ is reported on the right, while one of the solution string x, corresponding to the Hamiltonian path $H = \langle 1, 3, 4, 2\rangle$ is represented at the bottom.

We build an instance of #FMO as follows. For each vertex i, we construct the multiset Q_i containing two occurrences of the symbol i (if $i \neq w, s$), or one occurrence of i and one of the special symbol c_i (if $i = w, s$). We also add symbols d_{ij} and j to Q_i, for every incident edge $\{i, j\}$. As a result, each undirected edge $\{i, j\}$ is represented by two different symbols $d_{ij} \in Q_i$ and $d_{ji} \in Q_j$. Formally,

$$
Q_i = \begin{cases} \bigcup_{\{i,j\}\in E}\{d_{ij}, j\} \cup \{i, c_i\} & i = w, s\\ \bigcup_{\{i,j\}\in E}\{d_{ij}, j\} \cup \{i, i\}, & i \neq w, s \end{cases}
$$

To guarantee the condition that w and s are the source and the destination vertices, respectively, we introduce two symbols c'_w and c'_s, and two sets $R_w = \{c_w, c'_w\}$ and $R_s = \{c_s, c'_s\}$, which do not correspond to any vertex of the input graph. They are used to guarantee that Q_w and Q_s will always occur as the first and the last multiset of any solution string x for our #FMO instance.

In general, the intersection between two multisets Q_i and Q_j can contain more symbols than just i and j. For example, the intersection between Q_1 and Q_4 is $I_{14} = \{1, 4, 2, 3\}$ because it contains also 2 and 3, each of them corresponding to the vertex forming a triangle with 1 and 4, respectively. To avoid this situation, $2\,|E|$ auxiliary multisets $Q_{ij} = \{d_{ij}, j\}$ are used to constraint the intersection between the multisets inside each solution string x, such that it contains exactly two symbols. Observe that each edge $\{i, j\} \in E$ gives rise to two multisets Q_{ij} and Q_{ji}. In the string x shown in Figure 3.8, the purpose of the multisets Q_{ij} and Q_{ji} is to enforce the intersection between Q_1 and Q_3 inside x to be $\{1, 3\}$, between Q_3 and Q_4 to be $\{3, 4\}$, and so on.

We finally choose the multiset $R = Q \setminus R'$ where $Q = \bigcup_i Q_i \cup \{c'_w, c'_s\}$ and $R' = \bigcup_{i \neq w,s} \{i, i\} \cup \{w, s\}$. We also choose $F = \{Q_1, \ldots, Q_{|V|}\} \cup \{R_w, R_s\} \cup \{Q_{ij}, Q_{ji}\}_{\{i,j\} \in E}$. The idea behind the construction of R and F is illustrated in our example. Each Hamiltonian path H from $w = 1$ to $s = 2$ contains only one edge incident to w ($\{1, 3\}$ in our example), one edge incident to s ($\{2, 4\}$), and two edges incident to each of the other vertices in H ($\{1, 3\}$ and $\{3, 4\}$ incident to 3, and $\{3, 4\}$ and $\{2, 4\}$ incident to 4). The path H can always be represented by a string x having size $|R|$. The multisets Q_i occur inside x in the same order as that of the vertices i inside H. The intersection between consecutive Q_i and Q_j is now guaranteed to contain just i, j in consecutive positions of x. For example, Q_1, Q_3, Q_4, and Q_2 correspond to the vertices in $H = \langle 1, 3, 4, 2 \rangle$, while their intersections correspond to the edges used in H. Here is the role of R': since we do not know a priori which edges will be traversed by H, we can rely just on the multiset given by their endpoints, thus giving rise to R'. Even if we have to remove R' from Q to obtain R, we still guarantee that $\langle R, F \rangle$ is a valid #FMO instance.

Lemma 23. *Each multiset $M \in F$ is contained in R.*

Proof. We recall that $F = \{Q_1, \ldots, Q_{|V|}\} \cup \{R_w, R_s\} \cup \{Q_{ij}, Q_{ji}\}_{\{i,j\} \in E}$, and that $R = Q \setminus R'$ where $Q = \bigcup_i Q_i \cup \{c'_w, c'_s\}$ and $R' = \bigcup_{i \neq w,s} \{i, i\} \cup \{w, s\}$. Since we assumed that the degree of w is at least one, w has at least one incident edge $\{w, j\}$. By construction of the Q_i multisets, it follows that the symbol w has at least two occurrences in Q: one occurrence belongs to Q_w, while the second occurrence belongs to the multiset Q_j associated to the vertex j. Same as above for the destination vertex s, which occurs at least two times in Q. Since we assumed each one of the remaining vertex $i \neq w, s$, to have at least two neighbors in G, (let say j, l,) it follows that the symbol i has at least four occurrences in Q: two occurrences belong to Q_i, the third occurrence belongs to Q_j, while the fourth one belongs to Q_l.

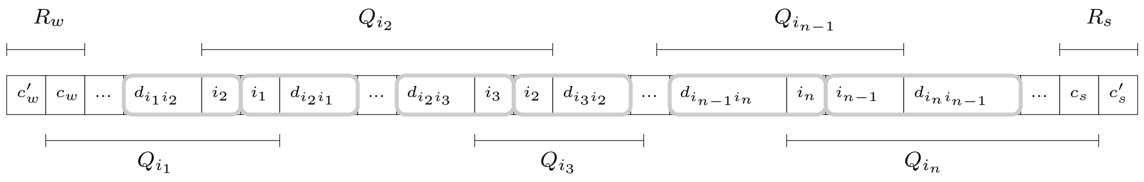

R_w
Q_{i_2}
$Q_{i_{n-1}}$
R_s
c'_w c_w ... $d_{i_1 i_2}$ i_2 i_1 $d_{i_2 i_1}$ ... $d_{i_2 i_3}$ i_3 i_2 $d_{i_3 i_2}$ ... $d_{i_{n-1} i_n}$ i_n i_{n-1} $d_{i_n i_{n-1}}$ ... c_s c'_s
Q_{i_1}
Q_{i_3}
Q_{i_n}

Figure 3.9: The string x coding the Hamiltonian path $H = \langle i_1, \ldots, i_n \rangle$ of G. Intersections between Q_i and Q_j have size 2 in x and are constrained to be $\{i, j\}$.

From the above, it follows that $R = Q \setminus R'$ contains at least one occurrence of w, one occurrence of s, and two occurrences of each $i \neq w, s$.

At this point, we have all the ingredients to prove that $Q_w \subseteq R$. The multiset Q_w contains exactly one occurrence of w, and at most one occurrence for every other symbol $i \neq w$. Moreover, for each $d_{ij} \in Q_w$, it holds that $d_{ij} \in R$, since $R \subseteq Q$, and no one d_{ij} is in R'. Also the symbol c_w is contained in R, since $c_w \in Q_w$, but $c_w \notin R'$. Same as above for Q_s, and the remaining Q_i multisets, with $i \neq w, s$. In the case of the Q_i multisets, the symbol i occurs two times inside each Q_i, but this is not an issue since, as discussed above, R contains at least two occurrences of each symbol $i \neq w, s$.

To prove that each $Q_{ij} = \{d_{ij}, j\}$ and $Q_i j = \{d_{ji}, i\}$ is contained in R, it is enough to note that $d_{ij}, d_{ji} \in Q$, but $d_{ij}, d_{ji} \notin R'$, and that for every symbol i or j there is at least one occurrence in R.

Finally, we observe that $R_w = \{c_w, c'_w\}$ and $R_s = \{c_s, c'_s\}$ are contained in R, since the symbols c_w, c_s, c'_w, c'_s are in Q, but they are not in R'. □

Lemma 24. *Given an undirected graph $G = \langle V, E \rangle$, together with a source and a destination vertex, w and s, the corresponding instance $\langle R, F \rangle$ of #FMO, can be built in $\mathcal{O}(|V| + |E|)$ time.*

Characterization of the solutions

We need some technical lemmas, as in Section 3.5.4. In particular, Lemmas 25–27 follow the same route as that traced in Lemmas 17–19 for #FRONT.

Lemma 25. *Let $G = \langle V, E \rangle$ be an undirected graph, and $\langle R, F \rangle$ be its corresponding #FMO instance. For any string x that is solution of $\langle R, F \rangle$, there exists a corresponding Hamiltonian path H of G from w to s.*

Lemma 26. *Let $G = \langle V, E \rangle$ be an undirected graph, and $\langle R, F \rangle$ be its corresponding #FMO instance. For any Hamiltonian path H of G from w to s, there exists at least one corresponding solution x of $\langle R, F \rangle$.*

Lemma 27. *Let Σ_H denote the set of all the solutions of $\langle R, F \rangle$ corresponding to a given Hamiltonian path H, as stated in Lemma 26. Then, for any two Hamiltonian paths $H \neq H'$ of G from vertex w to vertex s, it is $\Sigma_H \cap \Sigma_{H'} = \varnothing$.*

We now prove Lemma 25, leaving the proof of Lemmas 26–27 at the end of the section. We consider a solution x of $\langle R, F\rangle$, and make three conceptual steps.

(a) We prove that the multisets Q_i follow a total order $\prec_x$ induced by x.

(b) We show that each Q_i occurs exactly once in x.

(c) For any two consecutive Q_i and Q_j in the total order $\prec_x$, we demonstrate that their intersection in x corresponds to edge $\{i, j\} \in E$.

Observe that steps (a) and (b) select all possible permutations of the vertices in V, while step (c) selects only those permutations (if any) that correspond to paths in G. Putting (a)–(c) together, we can see that the Hamiltonian path corresponding to x is $H = \langle i_1, i_2 \ldots, i_{|V|}\rangle$, where $Q_{i_1} \prec_x Q_{i_2} \prec_x \cdots \prec_x Q_{i_{|V|}}$ is the total order induced by x.

We show a slightly more general property than that stated in (a), using the following lemma.

Lemma 28 (Strict Sperner Property)**.** *The collection of multisets $C = \{R_w, R_s, Q_1, \ldots, Q_{|V|}\}$, is a Strict Sperner collection: no multiset is contained in the union of the others. Hence, there exists a total order $\prec_x$ on the multisets in C.*

Proof. First of all, we observe that each $Q_i \in C$ contains at least one symbol d_{ij} that is unique in x and does not belong to any other multiset. Hence, Q_i cannot be contained in the union of the other multisets. Also the multisets R_w and R_s contain unique symbols, namely, c'_w and c'_s. Hence, C is a strict Sperner collection: this property, combined with the fact that each multiset in C occurs in x, implies that a left-to-right scan of x provides a total order of the multisets in C. That is, for any pair Q_i and Q_j either $Q_i \prec_x Q_j$ or $Q_j \prec_x Q_i$. □

We prove the property stated in step (c) by the following lemma.

Lemma 29 (Intersection Size)**.** *Let x be a string of size $|R|$, drawn from all the symbols in R, and containing all the multisets in $C_2 = \{R_w, R_s, Q_i, Q_{ij}\}$. Let $I_{ij} = Q_i \cap Q_j$ denote the intersection between two multisets Q_i and Q_j that occur consecutively in x. Then, (i) $|I_{ij}| = 2$; (ii) $I_{ij} = \{i, j\}$; (iii) $\{i, j\} \in E$.*

Proof. (i) First, let $l_1, l_2, \ldots$ denote some generic vertices that are adjacent to both i and j. By construction of the multisets Q_i, note that I_{ij} can only contain the symbols i, j or l_p for $p = 1, 2, \ldots$. Formally:

$$Q_i \cap Q_j = \begin{cases} \{i, j\} \cup \bigcup_{\{i,l_p\},\{l_p,j\} \in E} \{l_p\} & \{i, j\} \in E \\ \bigcup_{\{i,l_p\},\{l_p,j\} \in E} \{l_p\} & \text{otherwise} \end{cases} \tag{3.2}$$

Assume that $|I_{ij}| = 3$. Then, four cases are possible when considering the sets Q_{fg} where $f, g \in \{i, j, l_1, l_2, \ldots\}$ and $f \neq g$:

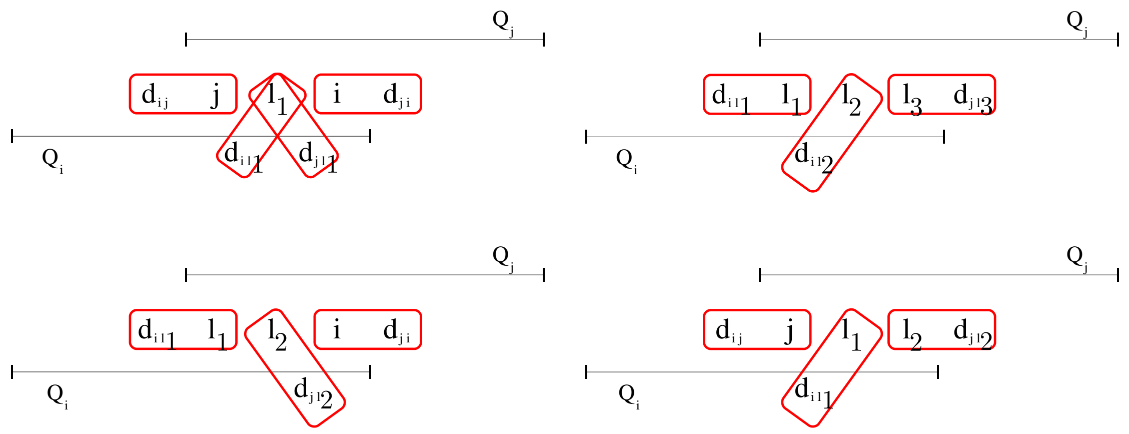


Figure 3.10: The four possible cases if $I_{ij} = 3$. From left to right, top-down, the case where $I_{ij} = \{i, j, l_1\}$, $I_{ij} = \{l_1, l_2, l_3\}$, $I_{ij} = \{i, l_1, l_2\}$, or $I_{ij} = \{j, l_1, l_2\}$.

1. $I_{ij} = \{i, j, l_1\}$
2. $I_{ij} = \{i, l_1, l_2\}$
3. $I_{ij} = \{j, l_1, l_2\}$
4. $I_{ij} = \{l_1, l_2, l_3\}$

We discuss case 1 (since cases 2–4 are similar), which is represented on the top left of Figure 3.10. Here, it is shown that the symbols in the four multiset Q_{ij}, Q_{ji}, Q_{il_1} and Q_{jl_1}, corresponding to the three edges $\{i, j\}$, $\{i, l_1\}$ and $\{j, l_1\}$, cannot occur inside Q_i or Q_j, because each symbol d_{ij} only belongs to Q_i (hence it cannot be a member of the intersection I_{ij}), and we only have one occurrence of l_1 inside Q_i and one occurrence inside Q_j.

The cases where the intersection has size larger than 3 are similar. In these cases we can always select from I_{ij} a subset of three symbols, reducing to one of the above cases: if $|I_{ij}| > 3$, we can apply the above argument to i, j and an arbitrary vertex in $I_{ij} \setminus \{i, j\}$.

Given the above upper bound on the size of an intersection, we now prove that $|I_{ij}|$ cannot be smaller than 2. By Lemma 28 we know that each multiset Q_i cannot be contained in the union of the other multisets, hence in order to construct a string x of size $|R|$ containing all the multisets in C_2, the combined size of the intersections between the Q_i multisets must be $2(|V| - 1)$. Assuming that at least one of such intersections has size 1, then some other intersection would have size 3, contradicting the previous upper bound. From the previous upper and lower bounds it follows that each intersection must have size $|I_{ij}| = 2$.

(ii) To prove that $I_{ij} = \{i, j\}$, let us assume by contradiction that $I_{ij} = \{i, l\}$, where $l \neq j$ is a vertex forming a triangle in the input graph G together with i and j.

As in point (i), it is easy to prove that the two sets $Q_{il} = \{d_{il}, l\}$, $Q_{jl} = \{d_{jl}, l\}$ cannot occur inside the solution string x, since Q_j and Q_i only contain one occurrence of the symbol l each. The d_{jl} symbol cannot be contained in the intersection I_{ij} since only the symbols i and l are inside.

The proofs for the other cases $I_{ij} = \{j, l_1\}$ and $I_{ij} = \{l_1, l_2\}$ are identical to this one.

(iii) The conclusion follows from the point (ii) and from the intersection property highlighted in Equation (3.2), stating that if $\{i, j\} \subseteq I_{ij}$, then $\{i, j\} \in E$. □

Finally, the property stated in step (b) is based on the lemma below.

Lemma 30 (Occurrence Uniqueness). *Given a solution x of $\langle R, F\rangle$, each multiset $Q_i \in F$ occurs exactly once inside x.*

Proof. We recall that each Q_i occurs at least once inside x since the latter is a valid solution. Suppose by contradiction that there exists a multiset Q_{i^*} which occurs twice or more inside x.

First, we show that all the occurrences of Q_{i^*} form a *run*, that is, any two such occurrences must overlap and there is no occurrence of Q_k $(k \neq i^*)$ between them. This is easy to see, since each d_{i^*j} occurs only once in x.

Second, consider all the runs in x, where a multiset occurring once is seen as a degenerate run. If two runs intersect, their intersection contains exactly two symbols by Lemma 29.

Third, the run of Q_{i^*} must be degenerate, thus contradicting the hypothesis that there are at least two occurrences. Indeed, if the run of Q_{i^*} is not degenerate, then $|x| > |R|$, which is not possible. To see why, we recall that a valid solution x of $\langle R, F\rangle$ is required to have size $|x| = |R| = 4|E| + 4$. Since $q = |\bigcup_i Q_i| = 4|E| + 2|V|$, some overlaps between consecutive runs are required. As previously mentioned, the intersection of two consecutive runs contains two elements. Hence, $r = 2|V| - 2$ is the number of symbols in the overlaps between pairs of consecutive runs in x. In order to fit the required length $|R|$, the first run must also intersect R_w in c_w, while the last one must intersect R_s in c_s. We also should add to these q elements, the two special symbols c'_w and c'_s, totalizing $|x| = |R| = (q + 2) - r$ elements in x (and so many in R as well). If the run of Q_{i^*} is non-degenerate, then its size will be at least $|Q_{i^*}| + 1$, implying that there are at least $(q + 2 + 1) - r > |R|$ symbols in x. Consequently, $|x|$ would be strictly larger than $|R|$, contradicting the validity of x as solution of $\langle R, F\rangle$. □

It remains to prove Lemma 26 and Lemma 27.

Let us discuss Lemma 26. Given a Hamiltonian path $H = \langle i_1, i_2, \ldots, i_{|V|}\rangle$ of G, where $i_1 = w$ and $i_{|V|} = s$, in order to construct a solution x of the corresponding #FMO instance $\langle R, F\rangle$, we arrange the multisets Q_i in the same order as the corresponding vertices in H, as shown in Figure 3.9. The first symbol of x is c'_w and the last one is c'_s. Between them, $Q_{i_1}, Q_{i_1}, \ldots, Q_{i_{|V|}}$ appears in x, where the first symbol of Q_{i_1} is c_w, and the last symbol is i_1, and the first symbol of $Q_{i_{|V|}}$ is $i_{|V|}$ and the last symbol is c_s. For the remaining Q_{i_l}, the first three symbols are i_l,i_{l-1}, and $d_{i_l i_{l-1}}$, and the first two of them overlap with $Q_{i_{l-1}}$ by Lemma 29. Analogously, the last three symbols are $d_{i_l i_{l+1}}$, i_{l+1} and i_l, and the last two of them overlap with $Q_{i_{l+1}}$. The remaining symbols in Q_{i_l} are $d_{i_l j}, j$ for all edges $\{i_l, j\} \in E$, such that $j \neq i_{l-1}, i_{l+1}$.

Each multiset Q_{i_l} intersects $Q_{i_{l+1}}$ in $\{i_l, i_{l+1}\} \in E$. Note that, since H is a Hamiltonian path, the symbols belonging to the union of all the intersections are $R' = \bigcup_{i \neq w,s}\{i, i\} \cup \{w, s\}$. To prove that x is a solution of $\langle R, F\rangle$, note that x contains each multiset Q_i, R_w, R_s by construction. As for each $Q_{ij} = \{d_{ij}, j\}$, we observe that its occurrence is contained in the occurrence of Q_i in x. Moreover, x contains the multiset R and x has size $|R|$, since x is drawn from the multiset $\bigcup_i Q_i \cup \{c'_w, c'_s\} \setminus R'$, which is exactly the way R is defined in $\langle R, F\rangle$. The above discussion proves Lemma 26.

To prove Lemma 27, consider a string $x \in \Sigma_H$, and $x' \in \Sigma_{H'}$ where $H' = \langle i'_1, i'_2, \ldots, i'_{|V|}\rangle$. Since $H \neq H'$, they must differ in at least one position l (i.e. $i_l \neq i'_l$). Assume w.l.o.g. that $|Q_{i_l}| \leq |Q_{i'_l}|$, and select the position k of the leftmost symbol $d_{i_l j} \in Q_{i_l}$ occurring in x for some j. Since the order of the multisets in x is the same as that of the vertices in the Hamiltonian paths, $Q_{i_l} \neq Q_{i'_l}$ (since $i_l \neq i'_l$). By construction of the multisets, we have $d_{ij} \notin Q_{i'_l}$, then the k-th symbol in x and x' differs, thus proving the claim.

Reduction from #HAM to #FMO

The #FMO problem is clearly in $\#\mathcal{P}$, since we can take a solution string x as a certificate. Therefore, we focus on its completeness.

We are given an undirected graph $G = \langle V, E\rangle$, along with its source w and its destination s. The reduction goes as follows.

- Build an instance $\langle R, F\rangle$ as described in Section 3.5.5.
- Let z be the number of solutions for the instance $\langle R, F\rangle$.
- Let $a = \prod_{i=1}^{|V|} \alpha_i \neq 0$, where α_i is defined as follows for a vertex i of degree d_i:

$$\alpha_i = \begin{cases} 2^{(d_i-1)}\,(d_i-1)! & i = w, s \\ 2^{(d_i-2)}\,(d_i-2)! & i \neq w, s \end{cases}$$

- Return the integer z/a.

The above reduction takes polynomial time. To see its correctness, it suffices to show that $|\Sigma_H| = a$ for every Hamiltonian path $H = \langle i_1, i_2, \ldots, i_{|V|}\rangle$ in G.

We already proved in Section 3.5.5 that each solution $x \in \Sigma_H$ has the form reported in Figure 3.9. Here, the occurrence of each Q_i is a sequence of pairs

$Q_{ij} = \{d_{ij}, j\}$ except the first and the last symbol of Q_i. If $i \neq w, s$, the first and the last pairs always stay the same, while the remaining $d_i - 2$ pairs can be permuted in $(d_i - 2)!$ ways. For each such a way, we can permute each pair internally, thus giving an extra factor of $2^{d_i - 2}$. If $i = w, s$, we have $d_1 - 1$ pairs that can be permuted, yielding $2^{(d_i - 1)}(d_i - 1)!$ permutations.

Theorem 31. *#FMO is #$\mathcal{P}$-complete.*

Corollary 32. *Testing the C1P on multisets is $\mathcal{NP}$-complete.*

3.6 Conclusions and Future Work

In recent years, the investigation of conserved gene clusters has become a fertile field of investigation in comparative genomics. However, modeling the input sequence of the dataset as permutations, with no repeated symbols, can be too stringent for real world genomic sequences.

The generalization of the common intervals frameworks that have been proposed in literature for the case of permutations to the case of strings with multiplicities is not straightforward, and all the combinatorial properties that are known in the case of permutations do not hold anymore in the case of strings. In particular, while in the case of permutations several notion of maximality have been proposed in literature to reduce the number of returned patterns, this is not the case of strings.

In the current chapter we explored a different approach, that does not rely on any notion of maximality, and that without sacrificing the information about the symbol multiplicities, ranks the π-patterns according to the conserved subpatterns detected in their occurrence lists. This approach has been formalized in Problem 3, and in Section 3.2 a two-phase approach has been proposed to solve it. The π-pattern detection phase has been deeply analyzed in Section 3.3.1 and in Section 3.3.2, where we proposed a novel $\mathcal{O}(L \log |\Sigma| n)$ time, and $\mathcal{O}(Ln)$ space algorithm, that improves the state of the art algorithm by a $\log(n)$ time factor.

While the π-pattern detection phase can be efficiently performed even in the case of repeated symbols, the complexity of the ranking phase heavily depends on the presence of repeated symbols in the pattern p to rank.

In fact, while in Section 3.4 we showed that given a π-pattern p, together with its occurrence list $\mathcal{L}(p)$, if p is a set of symbols then the number of frontiers represented by its minimum PQ-tree can be computed in $\mathcal{O}(|\mathcal{L}(p)||p|)$ time and $\mathcal{O}(|p|)$ space. In Section 3.5.4, we proved that even the computation of the number of the frontiers of a given PQ-tree where the label of the leaves are not necessarily distinct is #$\mathcal{P}$-complete.

In general, the above hardness result is an issue for biological applications where the number of repeated symbols in a pattern is high. However, there are some concrete cases where the small number of repeated symbols make the computation feasible. For example in the case study discussed in [122], the authors show how to construct the minimum PQ-trees for patterns containing thousands of symbols, with few repetitions, by a brute force solution. Given the small number of nodes in the above trees, and the small number of repeated symbols, the computation of the number of frontiers is feasible, and can be easily performed by slightly modifying the algorithm that counts the number of frontiers in the case of no repeated symbols. Defining a practical algorithm that can efficiently count the number of frontiers of a PQ-tree at least in the case where the number of repeated symbols is small, is the main open problem that we planned to investigate in the future.

The above open issue together with some other interesting lines of research that can be the subject of further investigations, is detailed in the rest of this section.

Improving the π-pattern discovery phase. In Section 3.3.2 we proposed a novel $\mathcal{O}(L \log |\Sigma|\, n)$ time algorithm to detect all the π-patterns in the input text T, together with their occurrence lists.

However, this algorithm cannot take advantage on the quorum threshold, because the lists of tuples constructed at each iteration by the algorithm cannot be pruned by removing the non-frequent tuples. It would be of interest to design an algorithm whose time complexity depends not only on the size of the input text n, on the maximum size of the searched π-patterns L, and the size of the alphabet Σ, but also on the value of the quorum threshold q.

	abcdefe	eabcdfe	efeabcd
(1)	1234567	5123467	5671234
(2)	1234567	5123467	7651234
(3)	1234567	7123465	5671234
(4)	1234567	7123465	7651234

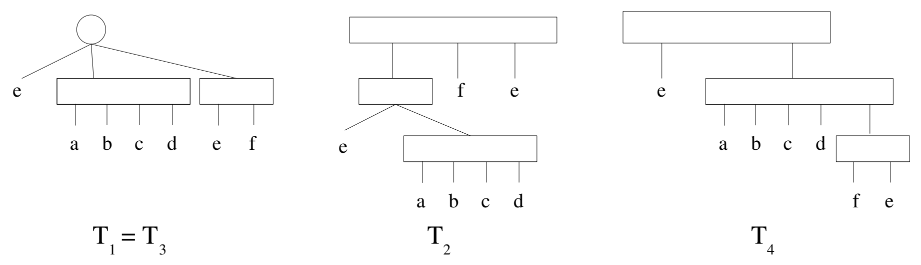


Figure 3.11: The PQ-trees corresponding to the four possible relabeling of the occurrence list $\mathcal{L}(p) = \{\texttt{abcdefe}, \texttt{eabcdfe}, \texttt{efeabcd}\}$ of the pattern $p = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}, \texttt{e}(2), \texttt{f}\}$. T_2 and T_4 are both minimal.

Construction of the minimal PQ-tree in the case of repeated symbols. In Section 3.4 we showed that given a π-pattern p, together with its occurrence list

$\mathcal{L}(p)$, if p is a set of symbols, then the minimum PQ-tree for $\mathcal{L}(p)$ can be computed in $\mathcal{O}(|\mathcal{L}(p)||p|)$ time and $\mathcal{O}(|p|)$ space

The problem when the input pattern p is a multiset and $\mathcal{L}(p)$ is no more a list of permutations of p but it is a list of strings with repetitions, is that the minimal PQ-tree is no more unique, as showed in Figure 3.11, where the pattern is the multiset $p = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}(2), \mathtt{f}\}$, that has three occurrences $\mathcal{L}(p) = \{\mathtt{abcdefe}, \mathtt{eabcdfe}, \mathtt{efeabcd}\}$, and that in the following we refers as s_1, s_2, and s_3.

Relabeling each character of the first occurrence with a distinct integer and accordingly the other occurrences in the other strings we obtain $s_1 = 1234567$, $s_2 = [57]12346[57]$, $s_3 = [57]6[57]1234$, where [57] means that the corresponding $\mathtt{e}$ can be either relabeled with 5 if it is considered as an occurrence of the first $\mathtt{e}$ in s_1 or with 7 if it is considered an occurrence of the second $\mathtt{e}$.

As shown in Figure 3.11 these possible relabeling choices lead to 4 different cases, and 3 different PQ-trees. In fact, the tree constructed in the first case is equivalent to the tree constructed in the third case (i.e. $T_1 \equiv T_3$). The trees T_2 and T_4, are both minimal, since they both represent 8 strings, although they are not equivalent.

Taking into account this problem the second step of algorithm 2 should consider all the possible relabelings of the occurrences of p, constructing for each of them the corresponding minimum PQ-tree. Remembering that $sign(p)[i]$ is the multiplicity of the symbol σ_i in the pattern p, if the occurrence list of p is $\mathcal{L}(p) = \{s_1, \ldots, s_k\}$, then the number of possible relabeling cases is $(\prod_{i=1}^{|\Sigma|} sign(p)[i]!)^{k-1}$, since we have to relabel each string s_i but the first, which is labeled by the identity function, assigning to the occurrences of the symbol σ_i in s_i a permutation of the $sign(p)[i]$ integers relabeling the occurrences of σ_i in s_1. In the above example, since the first occurrence of $\mathtt{e}$ in s_1 is labeled with 5 and the second with 7, the occurrences of $\mathtt{e}$ in s_2 and s_3 are labeled with a permutation of $\{5, 7\}$.

This naive algorithm obviously requires exponential time in the multiplicities of the repeated symbols. Can we do better? What is the complexity of the problem of constructing the minimal PQ-tree given a list of strings with repeated symbols?

Counting the number of frontiers. While in Section 3.4 we described a simple recursive procedure to compute the size of the set of frontiers represented by a PQ-tree T whose leaves are labeled with distinct symbols, Theorem 22 in Section 3.5.4 proves that counting the number of frontiers represented by a PQ-tree whose leaves are not labeled with distinct symbols is $\#\mathcal{P}$-complete.

Although in the general case of repeated symbols, counting the number of frontiers of a given PQ-tree T is hard, it would be interesting to design an algorithm that can approximate this count, at least in the case when there are few repeated symbols, and most of the symbols are unique.

$$\begin{tikzcd}
\#HAM \arrow[r, "3.5.4"] \arrow[d, "3.5.5"] & \#FRONT \arrow[dl, squiggly, leftrightarrow] \\
\#FMO &
\end{tikzcd}$$

Figure 3.12: Relation between the counting problems described in Section 3.5.5 and 3.5.4.

Orderings for the C1P in multisets and PQ-tree frontiers. An interesting implication of our findings in Section 3.5.5 is the one illustrated in Figure 3.12 where #HAM denotes the counting version of the Hamiltonian path problem. As previously mentioned, a *direct* mapping of the orderings for the C1P in multisets into the frontiers of PQ-trees has some intrinsic ambiguity. On the other hand, we proved in Section 3.5.4 and 3.5.5 that both the counting problems #FRONT and #FMO are $\#\mathcal{P}$-complete using a reduction from #HAM. By the completeness properties, it follows that there must exist an *indirect* mapping between the latter two problems, but we do not know how to build it explicitly and directly (apart from the obvious composition). It would be interesting to find a direct and "natural" reduction between the two problems, without using the counting version of the Hamiltonian path as an intermediate problem.

Chapter 4

Mobilomics in *S. cerevisiæ*: a Concrete Example of Pattern Discovery

For a long time the mainstream view of genome evolution has associated organism complexity with the number of protein coding genes. More recently, the sequencing of the human genome has shown that the number of genes in the human genome is only twice the number of genes in *Drosophila* [123], highlighting the relevance of non-protein coding gene pathways in controlling the differentiation and diversity of organisms [178]. In [82, 45] the authors write that only 2% of the human genome codes for proteins, while the function of the rest of the genome is still unknown. However, some of these non-coding regions are well conserved across species, as shown in recent large-scale studies [20]. The empirical fact that in the human genome 42% of these well conserved non protein-coding regions are built by *transposable elements* (*transposons* for short), raises some questions about the role of transposons in genome evolution [45].

In the past, it has been thought that the genome was a static entity, not being subject to the movements of any of its parts. The revolutionary idea that genomes contain segments of mobile genetic material, has been formalized in the 1950s, thanks to the work of McClintock, that discovered the first transposon in maize genome [136, 137]. Roughly speaking, transposons are DNA sequences that are able to move from one genome location to another. A complete transposons classification is difficult, since the transposon structure show a high variability in different species, new transposons are reported every day, and even the transposition mechanism can change from one class of transposons to another (for a up to date classification the interested reader can refer to [104, 187], while an alternative classification based on the transposition mechanism can be found in [70]).

Although transposons are *universal*, since they have been detected in all the sequenced prokaryotic and eukaryotic species, their abundance and their effects on genome stability widely vary among different organisms, and it is still argument of debate in the scientific community. For example, in the human genome retrotransposons (one class of transposons whose behavior resemble that of retroviruses) constitute almost one half of the human genome, but they are responsible for only 0.2% of spontaneous mutation [109], while in *Drosophila*, they are the source of more than 50% of mutations having a phenotypic effect [61].

The effects of transposon movements depend on the insertion site. If the insertion disrupts a gene, affecting the fitness of the organism, with high probability the natural selection will eliminate the mutation, whereas if the insertion is in a non-coding region, we may expect it to be maintained if it has no impact on host fitness. Furthermore, insertion of transposons in proximity of a gene can modify the regulatory pathways and the expression patterns the gene is involved into [158]. Tracking the movements of a genome transposons, and forecasting their future transpositions is a fascinating and intriguing challenge, that is crucial to better understand the genome evolution and dynamics. In recent years, it has has become the main topic of a new research field of genomics: the *mobilomics*.

An inevitable first step towards understanding the transposon dynamic consists in the identification of all the transposons that occur in a given genome. So far, this task has been approached by the bioinformatics community mainly by using consensus-like searches, by searching an already identified set of transposons in the given genome (see [25] for a survey of the mainstream methodologies).

This approach mainly suffers of two limitations. First, since the transposons to be searched must be given in input together with the genome to be analyzed, we must know a priori all the classes of transposons that occur in the given genome. Moreover, this approach can only detect transposons that are highly similar to the input ones, but it fails in identifying a previously unknown class of transposons. This is not an issue in species, where the structure of transposable elements is well characterized (for example in yeasts), but it is unapplicable where transposons show a much more variable structure, as for example in the human genome [84].

The second issue of the pattern search approach concerns the noisy nature of the analyzed genomes. In fact, modern sequencing technologies reduced the cost of the sequencing process, but the released genomic sequences usually have a low-coverage, and they are rich of *unresolved* bases. Namely, some locations of the released DNA sequence do not contain `A`, `C`, `G`, or `T`, but `N`, meaning that the sequencing process failed to identify that base. The presence of unresolved bases, which from a stringology point of view can be thought as wildcard symbols, makes the traditional approaches based on pattern search largely uneffective, since the occurrence of a given transposon can be "masked" by the presence of unresolved bases in the genomic sequence.

To overcome the above limitation, we describe an alternative approach that can be applied when multiple copies of the input genome are available. This is not a futuristic scenario. In recent years the new sequencing technologies are dramatically reducing the cost of the sequencing process. Hence, an increasing number of *population genomic* datasets, containing the genomes of different organisms of the same species (or strongly correlated species) are becoming available [130, 58].

Our alternative approach relies neither on homology nor structural features of transposons, but on their behavior. Transposons are mobile elements. Hence, if we align the given genome with the other copies, we can detect transposition event by detecting large insertion or deletion observed in the global alignment of the input sequences [146, 88].

This approach has two major advantages when compared with the traditional pattern search approaches. First, since it does not require transposons in input, it can detect a new class of transposons, whose structure is different from the known one. The second advantage is the applicability of our approach to low coverage genomes, which are rich of unresolved bases, where the traditional approaches are largely uneffective (this issue will be discussed in details in Section 4.4).

Notice the different perspective we are looking the transposon detection problem at. We recast a pattern search problem where the transposons to be searched are part of the input, as a pattern discovery problem where only some constraints characterizing the combinatorial structure and the occurrence list of the patterns that must be returned are given in input. More precisely, the pattern discovery problem can be formalized as follow. Given two copies of the same genomic sequence, T_1 and T_2, a substring s of T_1 is *conserved* in T_2 if all of its symbols are matching symbols in the optimal global alignment of T_1 and T_2. Otherwise, the substring s is *non-conserved* (or *mobile*).

Definition 4 (Mobile Element Discovery Problem). *Given two copies of the same genomic sequence, T_1 and T_2, find all the mobile substrings occurring in T_1 and T_2.*

While all the transposons are mobile elements (by definition), the converse does not hold, since not all the mobile elements are transposons. In the following sections we will assess the precision and the recall of our pattern discovery approach by a case study where we analyzed the dataset of 38 strains of the *S. cerevisiæ* yeast that has been recently released in [130]. As a reference strain, we choose the `s288c` strain, that has been fully sequenced and its transposons were annoted in the SGD database [170]. To avoid ambiguity, in the following we refer to this strain as `RefSeq` to indicate the fully sequenced release at the SGD database, which does not contain any unresolved base. The low-coverage assembly released in [130] will be referred as `s288c`.

Different motivations have suggested to choose the *S. cerevisiæ* yeast to perform this experimental study. First, although the *S. cerevisiæ* dataset in [130] has a low-coverage (one-to-fourfold), and it is rich of unresolved bases, to the best of our knowledge, *S. cerevisiæ* is the only organism having 39 sequenced strains. Second, yeast is probably one of the most intensively studied eukaryotic model in molecular and cell biology, and a deeply understood organism at molecular viewpoint. Finally, the `RefSeq` genome is accurately annoted and its transposons are described in the SGD database [170]. Although our pattern discovery approach, differently than the traditional pattern search approach, does not require any information about known transposons already detected in the input sequences, these annotations will be used to validate the results of our approach. To our knowledge, this is the first time that such a large dataset is analyzed to detect transposons.

The following sections are organized as follows. Section 4.1 introduces the essential biological concepts about transposons, discussing their importance in evolving genomes. After describing in Section 4.2 the *S. cerevisiæ* dataset that we used in our experimental study, we analyzed the limitations of the existing approach in Section 4.3, where we show the difficulties of the pattern search approach in handling low coverage sequences, and in Section 4.4, where an alternative sequence alignment approach is analyzed.

In Section 4.5 we describe the anchor-based approach for sequence alignment, which can overcome the inefficiency of the global sequence alignment method. Section 4.6 describes the `Regender` approach for transposon discovery, describing how the anchor-based approach can be modified in order to exploit the high similarity of the input sequences in a typical population genomic dataset.

In the last sections, we experimentally compared `Regender` with other 9 state of the art global alignment tools. To our knowledge it is the first time that such a large scale investigation is performed on a complete population genomic dataset. First, in Section 4.7.1, we evaluate the computational efficiency of the `Regender` approach. Then, in Section 4.7.2 we compared the quality of the output obtained by `Regender` with the output of the other tools. Finally, in Section 4.7.3 we validate the results computed by `Regender` through the transposon annotations that are available for `RefSeq` at SGD database [170].

In Section 4.8 we draw our conclusions, also discussing some promising lines of research.

4.1 Transposons in Yeast Genomes

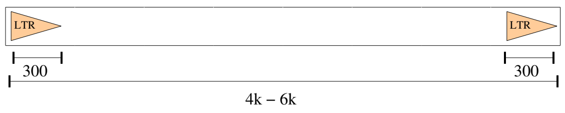


Figure 4.1: Structure of a yeast transposon (`TY` element). All the known yeast transposons show this structure. The size of the whole transposon range over 4000–6000 bases. The leading *Long Terminal Repeat* sequence is about 300 bases long, and it is repeated at the end of the transposon.

Genomes evolve both by acquiring new sequences and by rearranging existing sequences. The introduction of new sequences results from the ability of vectors to carry genetic information between genomes. Extrachromosomal elements move information horizontally by mediating the transfer of genetic material. For example, in eukaryotes, some viruses, notably the retroviruses, can transfer genetic information during an infective cycle. Rearrangements, instead, are promoted by processes that are internal to the genome. For example, duplication of sequences within a genome provides a major source of new sequences. One copy of the sequence can retain its original function, while the other may evolve by acquiring a new function.

Another major cause of variation is provided by *transposable elements* (*transposons* for short). In some sense they can be thought as the internal counterpart to the vectors that can horizontally transport sequences between genomes. Transposons are sequences in the genome that are mobile, namely they are able to transport themselves to other locations of the genome, without using any extrachromosomal vector (such as viruses), autonomously moving from one site of the genome to another.

A complete description of all the known types of transposable elements, and their role in the evolution of genomes is beyond the scope of this thesis and we address the interested reader to [151, 140, 34, 114]. However, in the rest of this section we briefly summarize the reasons why transposable elements are believed to have a prominent role in influencing the evolution of complexity of eukaryotes.

Since their discovery in [136], transposable elements (transposons for short) are believed to have the capacity to repattern genomes [137]. More recently, numerous papers have supported this idea by characterizing transposable elements, as generator of variations upon which natural selection can act [67, 31]. Transposons act to increase the evolvability of their host genome and provide a means of generating genomic modifications.

In the past, all the categories of transposons have been considered as "genomic parasites" [152, 94], but more recently, significant beneficial attributes for facilitating evolution have been recognized [67, 31].

It is now generally accepted that the emergence of increasingly complex eukaryotic life forms was accompanied by an increase in genome complexity characterized by an increase of gene number, and more elaborate gene regulation [150, 41]. Punctual modification of genetic sequences in the form of single base insertion, deletion, and substitutions cannot explain this deep evolution. Only DNA recombination in the form of gene or segmental duplications, insertions and deletions can account for this massive increase in gene number and the complexity of their regulation [150, 41].

Different types of transposons in different organisms show different structures. In the following sections we focus on yeasts, in particular *S. cerevisiæ*, that is one of the most extensively studied eukaryotic model organisms in molecular and cell biology [128]. All the transposons of *S. cerevisiæ* that have been identified and their annotations are available online at the SGD database [170].

Yeast transposons are usually referred as `TY`s (abbreviation of "transposon yeast"). All the known `TY` elements share the same general structure, illustrated in Figure 4.1 [128]. The size of each `TY` element ranges over 4000–6000 bases. The last 300 bases at each end are called *Long Terminal Repeat* (`LTR`). The two `LTR` elements of an individual `TY` element are likely to be identical or at least very closely related.

However, the annotations in the SGD database [170] show several relevant variations of the above structure. In fact, the size of the `TY` elements shows a high variability (see Table 4.2). The trailing `LTR` sequence can be structurally different from the leading one, and in some cases one of the two terminal repeat can be missing. Moreover, the dataset in [170] shows an abundant number of *solo*-`LTR` elements, which are single `LTR` elements.

These empirical observations suggest that we cannot rely on the structure of transposable elements only to identify them (another issue related to the "noisy" nature of datasets is discussed in Section 4.3).

4.2 Dataset Statistics

Chrm	Chrm Len.			Unres. Bases (%)			Non Tel. Unres. Bases (%)			Unres. Seg.	Unres. Seg. Len.		
1	245,970	269,662	256,777	14.45	33.03	26.08	0	5.09	1.55	31.79	1	24,820	2,113.22
2	796,583	829,582	809,904	0.32	5.25	2.58	0.11	3.51	1.31	12.66	1	10,438	1,654.10
3	318,126	350,787	336,516	3.96	15.09	9.42	3.08	11.52	7.43	16.87	1	15,047	1,892.71
4	1,493,070	1,552,866	1,528,192	1.12	6.11	4.41	0.73	4.36	2.97	44.87	1	20,189	1,505.67
5	572,021	609,362	592,993	1.92	12.07	7.53	0.05	7.75	4.31	18.47	1	16,536	2,430.67
6	273,660	299,532	288,771	2.35	16.09	11.13	0	8.18	4.60	21.03	1	16,793	1,541.48
7	1,099,952	1,133,555	1,116,038	2.75	8.68	6.54	2.11	6.64	4.56	37.29	1	20,166	1,960.68
8	560,052	581,929	568,868	2.37	10.95	6.57	0	2.15	0.89	21.03	1	14,663	1,781.86
9	446,596	482,241	469,020	2.31	16.27	11.96	0.80	7.84	5.18	32.29	1	29,293	1,747.14
10	758,004	789,108	774,498	3.65	13.07	9.92	1.37	6.50	4.38	32.71	1	30,673	2,352.38
11	676,435	707,470	696,323	0.81	7.92	4.49	0.12	4.40	2.59	26.11	1	14,769	1,202.81
12	1,067,059	1,112,810	1,088,492	2.53	9.55	7.03	1.44	6.64	4.08	28.87	1	21,358	2,655.13
13	914,484	940,352	925,395	1.78	5.92	3.65	0.27	3.50	1.83	22.34	1	16,388	1,513.10
14	781,629	833,923	813,182	1.74	9.86	6.80	1.07	7.02	4.48	24.87	1	21,893	2,231.49
15	1,095,496	1,129,663	1,115,167	1.73	7.76	5.82	0.75	4.17	2.97	29.13	1	19,764	2,233.89
16	933,498	974,582	955,289	1.35	7.11	4.93	0.32	5.31	3.28	18	1	16,842	2,626.35

Table 4.1: Statistics reporting the min/max/average length of each chromosome in the 38 strains of the dataset containing unresolved symbols, their min/max/average percentage of unresolved bases (both in the whole chromosomes and in the non-telomeric region only), the average number of unresolved segments, and min/max/average length of the unresolved segments.

As explained in the previous sections, our comparative approach for transposon discovery can only be applied when multiple copies of the input genome are available.

Recently, a dataset suitable for our purposes has been made available: 38 different strains of *S. cerevisiæ* have been sequenced, and the relative genomes published without annotations [130]. The dataset is not only important since it is the first time that such a large number of genomes of the same species is made available to the scientific community, but also because it is a typical example of a population genomic dataset, whose genomic sequences are obtained by the new sequencing technologies (see [130] for the details of the sequencing process).

The coverage of the dataset is relatively low (one-to-fourfold), and the genomic sequences contain a relevant fraction of unresolved bases (i.e. some locations of the DNA sequence do not contain `A`, `C`, `G` or `T`, but `N`). Each strain is composed by 16 chromosomes. Due to the presence of unresolved bases, the chromosomes do not have the same length in all the strains. Table 4.1 reports for each chromosome the min/max/average length of each chromosome in each of the 38 *S. cerevisiæ* strains.

The average length of the chromosomes is quite variable, since the longest chromosome, `Chr` 4, is about six times longer than `Chr` 1 that is the shortest one, while the aggregate size of the size of chromosomes in each strain is about 12 Mbases. The percentage of unresolved bases due to the relatively low coverage is not negligible, and as reported in Table 4.1 it ranges from 2.5% in `Chr` 8 to 26% in `Chr` 1. Unresolved bases are not uniformly distributed along the chromosomes, but tend to be clustered in the telomeric regions of each chromosome (*telomeres* are the leading and the trailing part of each chromosomic sequence, that protects the end of the chromosome from deterioration). In fact, as shown in Table 4.1, if we compare the percentage of unresolved bases found in the non-telomeric regions, with the overall number of unresolved bases, we discover that, on average, the 51% of the unresolved bases are contained in the telomeric regions of the chromosomes.

Table 4.1 also reports the average number of segments of unresolved bases, that are substrings of consecutive unresolved bases (i.e. "`NN`. . . `N`"). The number of unresolved segments is small, between 10 and 50 segments per chromosome, while the average length of an unresolved segment is about $2,000$ bases. In other words, this means that unresolved bases are not uniformly interspersed along the chromosomes, but they tend to be clustered in the telomeric regions, and are clustered in long unresolved segments.

This is a big issue in processing the dataset. In fact, if we search a sequence against one such chromosome, a single unresolved base can be handled by using standard wildcard pattern matching techniques [145, 49, 47, 88]. Nothing can be done for an unresolved segment of $2,000$ bases which completely masks a large substring of the chromosome (we will analyze this issue in more details, discussing the limits of the pattern search approach in Section 4.3).

Chrm	TYs	LTRs	Max/Min/Avg. TY Len.			Max/Min/Avg. LTR Len.		
1	1	7	5,925	5,925	5,925	340	133	279.14
2	3	16	5,959	5,916	5,930.67	371	79	256.25
3	2	16	5,959	3,144	4,551.50	361	89	287.13
4	8	22	5,959	5,493	5,880.38	679	28	299
5	2	28	5,924	5,727	5,825.50	715	67	310.79
6	1	9	5,959	5,959	5,959	332	76	255.22
7	6	32	5,961	5,351	5,836.83	371	62	276.50
8	2	20	6,223	6,028	6,125.50	341	92	250.75
9	1	8	5,428	5,428	5,428	371	190	308.13
10	3	19	6,226	5,922	6,023.33	667	92	266.79
11	0	14	-	-	-	340	131	301
12	5	22	5,959	5,443	5,832.40	344	171	297.73
13	4	16	5,926	5,903	5,914.25	371	127	286.81
14	3	12	5,914	5,297	5,703.67	371	69	279.67
15	4	24	5,961	5,914	5,940	342	84	268.54
16	5	22	6,223	5,918	5,984	344	43	256.82

Table 4.2: Statistics of the `TY`s and `LTR`s annotations downloaded from the SGD web site [170]. Each row reports the number of `TY`s and `LTR`s annoted in a specific chromosome, and the most relevant statistics about their lengths. Statistics about `TY`s in `Chr` 11 is omitted since no `TY` is annotated in that chromosome.

Table 4.2 shows the most significant statistics about the `TY`s and `LTR`s elements that are annoted in the `RefSeq` strain. These annotations will be used to assess the accuracy of our tool `Regender` (see Section 4.7.2). The annotations are publically available and can be downloaded from the SGD web site [170]. Note the different lengths between `TY` and `LTR` elements. While the average length of a `TY` is about 5,000 bases, `LTR` elements are much shorter: their average length is about 300 bases. The above statistics agree with the literature of yeast transposons [128], and they will play a prominent role in designing a methodology to overcome the unresolved bases issue, discriminating between `TY` and `LTR` elements.

4.3 The Limitations of the Pattern Search Approach

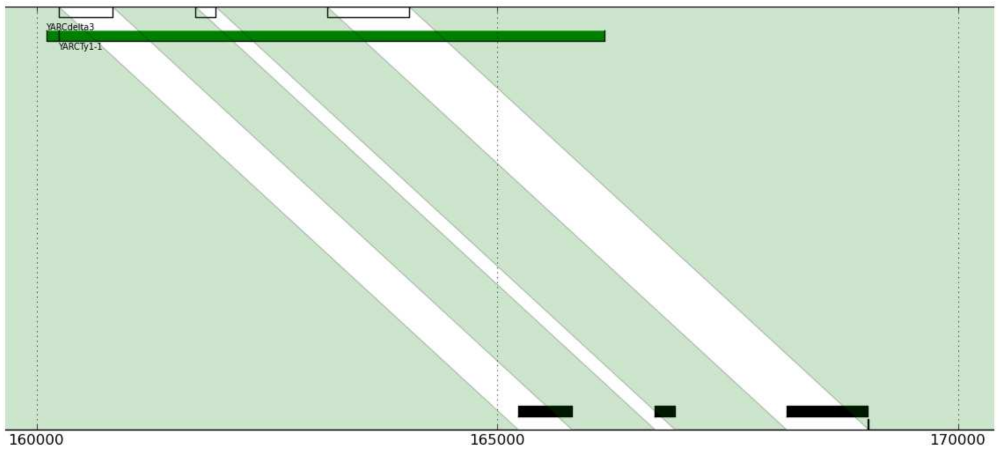


Figure 4.2: Global alignment of the `Chr` 1 of the `s288c` strain (on bottom) with the homologous chromosome of the `RefSeq` strain (on top). Green regions represent aligned segments, while the white regions represent non-aligned segments. The black rectangles on bottom represents segments of unresolved bases. The green rectangles on top represent the `YARCdelta3` and `YARCTy1-1` transposable elements, that are annoted on the `RefSeq` chomosome at the SGD database [170]. The global alignment shows that both the transposable elements are conserved in the `s288c` strain, but they are partially masked by the unresolved bases. Searching by `Blast` results in a false negative.

The mainstream approach to detect transposable elements uses `Blast` [8] or some other search tool [147, 111, 103] to search a known transposon `TY` in the input genomic sequences [25].

This approach suffers of several drawbacks. First, in order to search a query sequence `TY` in a target sequence, `TY` must be provided in input. Hence, this approach is only applicable if we know *a priori* all the transposons that are present in the input dataset. Moreover, this approach can only detect transposons that are highly similar to the input ones, but it fails in identifying previously unknown type of transposons. This is reasonable for genomes, as yeast, where the structure of transposable elements is well characterized, but it is unapplicable where transposons show a much more variable structure, as for example in the human genome [84].

The second issue of the pattern search approach concerns the structure of our dataset, which has been deeply analyzed in Section 4.2.

As we know from Section 4.2, the percentage of unresolved bases in our dataset is not negligible (see Table 4.1 for details). The presence of a consistent number of unresolved bases is not a feature of our dataset only. It characterizes all the datasets that are obtained by next generation sequencing technology [135, 172], hence we must tackle this issue to better analyze these datasets.

To understand why the pattern search approach fails in detecting transposable elements in low-coverage datasets, let us consider the example in Figure 4.2. Figure 4.2 shows part of the global alignment of the `Chr` 1 `s288c` strain (on bottom) with the homologous chromosome of the `RefSeq` strain (on top). We recall that `RefSeq` and `s288c` are supposed to be equal since they are the results of two different sequencing processes applied to the same input genome. The main difference is that `RefSeq` has been sequenced at higher coverage and it does not contain any unresolved base. In Figure 4.2, green regions represent conserved regions, while the white regions represent non-conserved segments. As expected, the global alignment clearly shows that both `YARCdelta3` and `YARCTy1-1` are conserved in the `s288c` strain, although the occurrence of `YARCTy1-1` in the `s288c` strain is interleaved with three segments of unresolved bases. Now we pose the following questions, what if we search the `YARCTy1-1` transposons in the `s288c` strain? Does `Blast` find them?

Since we are interested in finding the whole `YARCTy1-1 TY` we have to discard all the matches such that the ratio between the length of the matching segment in the `s288c` strain, and the length of the whole transposons is too small, below the 90% (otherwise, the number of small substring of `YARCdelta3` found in the `s288c` strain would generate a high number of false positive matches).

Given the situation in Figure 4.2, since `YARCTy1-1` is interleaved in the `s288c` strain by three long segments of unresolved bases, `Blast` returns four matches (among the others), corresponding to the four substrings that are in between the unresolved segments. Each one of these matches is too short to meet the aforementioned length threshold and it is discarded, so the `Blast` search results in a false negative claiming that `YARCTy1-1` does not occur in the `s288c` strain.

4.4 Transposons Detection by Global Pairwise Alignment

Size	Time
1000	0.33s
2000	0.39s
4000	0.73s
8000	1.95s
10000	2.78s
20000	10.84s
40000	47.63s
80000	3.28m
100000	5.16m
200000	20.90m
400000	84.77m
800000	> 5h

Table 4.3: Time required to globally align a prefix of the `Chr` 4 of the `RefSeq` strain with a prefix of the same size of the `Chr` 4 of the `Y55` strain. The size of the prefix is reported in the first column, while the alignment time in seconds (or minutes) is reported in the second column. To compute the global alignment we used the `Stretcher` tool, that is part of the `EMBOSS` suite [163]. We run `Stretcher` with the default parameters for DNA sequences. Tests have been performed on an Intel Core 2 Duo T5500 notebook, with 2GB of RAM. Missing values indicate that the computation has not been completed in 5 hours.

As discussed in Section 4.3, transposon detection methods that rely on the structure of the transposons are sensitive to the presence of unresolved bases in the genomic sequences. A simple idea consists in relying on the behavior of the transposons instead that on their structure. Transposons are mobile by definition, hence we can detect transposition event by detecting large insertions or deletions observed in the global alignment of the input sequences [146, 88]. In other words, given a pair of homologous chromosomes $\texttt{Chr}N_{\texttt{A}}$ and $\texttt{Chr}N_{\texttt{B}}$, if we align the two input sequences, by identifying the non conserved regions that are subject to large scale insertion or deletion that are compatible in size with the length of a transposon, we can easily detect the mobile elements of the two chromosomes.

However, although the quadratic space cost of the standard dynamic programming algorithm for global alignment can be attenuated [95], the quadratic time cost makes global alignment unpractical for long sequences. In state of the art implementations of global alignment algorithms several variations of the standard dynamic programming algorithm and heuristics are used to speedup the computation [88]. However, how fast are state of the art implementations in practice?

Table 4.3 shows the time required to align a prefix of the `Chr` 4 of the `RefSeq` strain with a prefix of the same size of the homologous chromosome of the `Y55` strain. The size of the prefix is reported in the first column, while the alignment time in seconds is reported in the second column. To compute the alignment we used the `Stretcher` tool, that implements the algorithm described in [143, 83], and that is part of the `EMBOSS` suite [163]. We run `Stretcher` with the default parameters for DNA sequences. Tests have been performed on an Intel Core 2 Duo T5500 notebook, with 2GB of RAM.

As we can see from this table, the computation is feasible for sequences up to 400 Kbases, while in the case of longer sequences the tool has been stopped after 5 hours without producing any output. Since the longest chromosome of our dataset is about 1.4 Mbases long, while the aggregate size of all the chromosomes of each strain is 12 Mbases, it follows that a different approach is required to detect the mobile segments of the dataset.

4.5 Anchor-Based Alignment

Alignment algorithms based on dynamic programming are invaluable tools for the analysis of short sequences. However, there are many reasons that limit their use for comparing *complete* chromosome (or genome).

The first issue, which was already discussed in Section 4.4, is their quadratic running time that makes them unsuitable when long sequences are compared [95, 143]. This weakness was first observed in the 1990s when the first complete genomes were sequenced [52].

The second issue comes from the nature of the sequences that we compare: comparing whole chromosomes/genomes differs fundamentally from comparing short sequences such as genes or proteins. Besides mutation events, large segments of a genome can be inserted, deleted, inverted, or change location. These large-scale modification events require the introduction of large gaps of thousands of base pairs or more. Even with the *affine gap cost model* (a variant of the standard dynamic programming algorithm that favors the extension of gaps more than matching characters in between and opening a new one), such large gaps are not tolerated [88, 112]. In fact, scoring schemes on the character level imply that long gaps are less likely to occur than short gaps. This is true for short related sequences, but it is far from reality for genomic sequences representing whole chromosomes and genomes in which longer gaps are often observed [143].

The unfeasible running time and the limitations in accommodating large gaps motivated researchers to develop different methods for comparing long genomic sequences. Nowadays, the strategy of choice for comparing whole chromosomes or complete genomes is the so called *anchor-based* alignment strategy [149].

The anchor-based strategy consists in three main phases:

1. Detecting a set of *fragments* shared by all the input genomic sequences (substring occurring highly conserved in all the input sequences).

2. Select a subset of fragments, called *anchors*, that will be the "matching" regions of the returned alignment.

3. Apply a standard dynamic programming technique to align the regions surrounding the anchors.

The anchor-based strategy has become the strategy of choice due to its efficiency, running in subquadratic time, and to its ability to overcome the limitations of traditional alignment algorithms. A complete description of the anchor-based strategy is beyond the scope of this section, and we refer the interested reader to [149]. However, understanding the basic idea is necessary to understand the way the `Regender` tool works, since it is going to be described in Section 4.6. Hence, in Section 4.5.1 we formally define what is a fragment and how to compute them, while in Section 4.5.2 we describe how to select a subset of anchors from the given set of fragments. For an introduction to the sequence alignment problem, we refer the interested reader to [88].

4.5.1 Fragment generation

Let T_i, $1 \leq i \leq k$, be an input text of size n_i drawn over the alphabet Σ. The sequence T_i can be seen as an array $T[0 \dots n-1]$ of symbols, where symbol $T_i[h] \in \Sigma$ is stored into position h, for $0 \leq h \leq n-1$. A substring $T[l_i]T[l_i+1] \cdots T[r_i]$ is represented as $T[l_i \dots r_i]$. A *fragment* f consists of two pairs $\texttt{beg}(f) = (l_1, \dots, l_k)$ and $\texttt{end}(f) = (r_1, \dots, r_k)$ such that the substrings $T_1[l_1 \dots r_1], \dots, T_k[l_k \dots r_k]$ are equal. In other words, a fragment is a substring occurring in all the input strings. A fragment is called *exact*, or *multiple exact match*, if it is composed of exact matches (i.e. $T_1[l_1 \dots r_1] = \cdots = T_k[l_k \dots r_k]$). If the equality $T_1[l_1-1] = \cdots = T_k[l_k-1]$ does not hold, then the multiple exact match is called *left maximal* (i.e. it cannot extend to the left in the input sequences). Similarly if the equality $T_1[r_1+1] = \cdots = T_k[r_k+1]$ does not hold, then the multiple exact match is called *right maximal.* The multiple exact math is called *maximal*, also referred as *multi*`MEM` in literature, if it is both left and right maximal. If only two sequences are given in input, then *multi*`MEM`s are referred as *maximal exact matches*, and they are abbreviated as `MEM`s. Moreover, if the substrings composing a fragment are restricted to be unique in the input sequence they belong, then they compose a *maximal multiple unique match*, abbreviated by *multi*`MUM`. In the case of two input sequences, then *multi*`MUM`s are referred to as *maximal unique matches*, and they are abbreviated as `MUM`s.

If the size of the substrings is fixed to a certain length L, then the substrings are called *L-grams.*

Examples of exact fragments are the exact L-grams as used in `Blastz` [168], maximal exact matches (`MEM`s) in `Avid` [35], the maximal multiple exact matches (*multi*`MEM`s) that are used in `Mga` [97], the maximal unique matches (`MUM`s) that are used in `Mummer` [52, 53], and the maximal multiple unique matches (*multi*`MUM`s) that are used in `Emagen` [54] and `Mauve` [51].

A fragment is *non-exact*, if it is composed of non-exact matches that allow character substitutions, insertions and deletions. Examples of non-exact fragments are the non-exact L-grams as used in the recent version of `Blastz` [168], and the extended seeds of `Glass` [18].

A comprehensive dissertation of all the techniques that are used by existing tools to detect fragments in the input sequences is beyond the scope of this section (the interested reader can refer to [42]). However, all the techniques for computing fragments in comparative genomics falls in one of the following three categories: hashing techniques, automata-based techniques, and suffix trees/suffix arrays techniques.

In the hashing techniques, a hash table is constructed to store the locations of all the L-grams in one of the input sequences. Then, all the other sequences are streamed against the table to locate common L-grams, by looking-up in the table each L-gram. The hashing technique, due to its simplicity, and practical efficiency, is widely used in software tools such as `Blastz` [168], `PipMaker` [169], `multiPipMaker` [167], `Blat` [111], `Glass` [18].

The hash table can also be used to compute non-exact fixed size L-grams and *multi*`MEM`. If we allow the mismatching characters to occur only at some fixed position of the L-gram, we can represent the non-exact fragment by a *binary mask* of ones and zeros, specifying the positions where mismatches are not allowed, ones, and where they are (zeros). For example, if we are interested in the 8-grams such that the third and the fifth characters do not care, the corresponding mask is `11010111`. These fragments can be easily computed by modifying the streaming phase of the hashing technique. This kind of non-exact L-grams has been first used in `PatternHunter` [131, 129], and later in a recent version of `Blastz` [168]. The hashing technique has also been used to generate *multi*`MEM` and *multi*`MUM` [127, 51], when the suffix trees/suffix arrays were not so popular in bioinformatics, but nowadays they are no more employed.

Also automata can be used to compute exact and non-exact L-grams. Instead of constructing a hash table for the first sequence, we can construct the Aho-Corasick automaton [6] for its L-grams, matching each L-grams of the sequences against the automaton. A modified version of the automaton is used, for example, in `Chaos` [39].

Although hashing technique is still used when fixed size L-grams are required due to its simplicity and practical efficiency, if *multi*`MEM` or *multi*`MUM` are required, suffix tree [186, 88] is the reference choice. It allows to compute *multi*`MEM` and *multi*`MUMin` linear time and space. Although suffix tree has been used in tools as `Mummer` [52, 53], its large space consumption (20 bytes per character), makes it impractical for very large sequences. Enhanced suffix array [1, 2] is a more space efficient data structure, that can compute *multi*`MEM` or *multi*`MUM` with the same time complexity as suffix tree. Recent versions of `Mga` [97] and `Emagen` [54] use this approach to compute fragments.

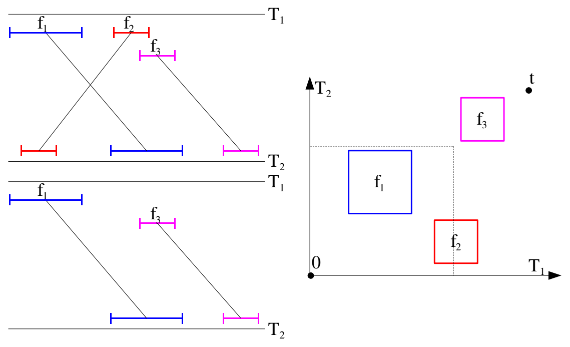


Figure 4.3: On top, the input set of fragments. The optimal global chain of fragments is reported on bottom. On the right, the geometric representation of the input set of fragments.

4.5.2 Anchor selection

After that fragments have been generated, the second step of the anchor-based alignment approach consists in selecting a subset of them, the so called anchors, which will be the "pivots" of the returned alignment.

A fragment f of k genomes can be represented by a hyper-rectangle in $\mathbb{R}^k$ with the two extreme corner points $\texttt{beg}(f)$ and $\texttt{end}(f)$. For any point $p \in \mathbb{R}^k$, let $p.x_1, p.x_2, \ldots, p.x_k$ its coordinates. A hyper-rectangle [161] is the Cartesian product of intervals on distinct coordinate axes. Let $p = (l_1, \ldots, l_k)$ and $q = (r_1, \ldots, r_k)$, we denote by $R(p, q)$ the hyper-rectangle $[l_1 \ldots r_1] \times \ldots \times [l_k \ldots r_k]$ with $l_i \leq r_i$ for all $1 \leq i \leq k$. Figure 4.3 shows an example in $\mathbb{R}^2$. The upper left figure shows a set of fragments in two input sequences, while on the right we reported their representation as rectangles.

We associate a non-negative weight, $f.\texttt{weight} \in \mathbb{R}$, with each fragment. The weight can be the length of the fragment in the case of exact fragments, but different choices based on the statistical significance of the fragments are also possible. In the following sections, when clear from the context, we identify the point $\texttt{beg}(f)$ or $\texttt{end}(f)$ with the fragment f, also assuming that the length of the fragment is selected as its weight.

Given a set of fragments, as showed in Figure 4.3, we add the points $\mathbf{0} = (0, \ldots, 0)$ and $\mathbf{t} = (|S_1|, \ldots, |S_k|)$ as the *origin* and the *terminus* fragment, having null weight. For these fragments, we define $\texttt{beg}(\mathbf{0}) = -\infty$, $\texttt{end}(\mathbf{0}) = 0$, $\texttt{beg}(\mathbf{t}) = \mathbf{t}$, $\texttt{end}(\mathbf{t}) = \infty$. Two fragments are *colinear* if the order of their respective segments is the same in all the input sequences. Two fragments *overlap* if their segments overlap in one of the input sequences. More formally, we say that f *precedes* f', $f \ll f'$, if and only if $\texttt{end}(f).x_i < \texttt{beg}(f').x_i$ for all $1 \leq i \leq k$. If $f \ll f'$ then they are colinear and non-overlapping. For example, in Figure 4.3 f_1 and f_3 are colinear and non-overlapping, while f_1 and f_2 are not, since the occurrence of f_1 precedes the occurrence of f_2 in the first sequence, but follows the occurrence of f_2 in the second sequence.

A *chain* of colinear and non-overlapping fragments (chain for short), is a sequence of fragments $C = (f_1, \ldots, f_l)$ such that $f_i \ll f_{i+1}$ for all $1 \leq i < l$. The *score* of the chain C is defined as the sum of the weights of its fragments minus the cost of connecting pairs of consecutive fragments. More precisely, $\mathtt{score}(C) = \sum_{i=1}^{l} f_i.\mathtt{weight} - \sum_{i=1}^{l-1} g(f_i, f_{i+1})$, where $g(f_i, f_{i+1})$ is the *gap cost* of connecting fragment f_i to f_{i+1} in the chain. At this point we have all the ingredients to define the global chaining problem:

Definition 5 (Global Chaining Problem). *Given m weighted fragments and a gap cost function, the* global chaining problem *is to determine an optimal global chain having maximum score, starting at the origin fragment* $\mathbf{0}$ *and ending at the terminus fragment* $\mathtt{t}$.

The global chaining problem is also called *fragment alignment problem* [64, 188], since the final alignments of certain programs are sometimes delivered in terms of these fragments, as in the case of `Dialign` [141, 142], or just *chaining* [88].

Several apparently unreleated problems reduce or are strongly correlated to the chaining problem. In the 1-*dimensional chaining problem* discussed in [88], the fragments are 1-dimensional intervals, and the optimal global chain is a chain of non-overlapping intervals. If we ignore gap cost, and set the weight of each fragment to its length, the 1-dimensional chaining problem is nothing more than the chaining problem in 1 dimension. In this case colinearity does not matter, because any two non-overlapping 1-dimensional fragments are also colinear.

Also the *single coverage problem* introduced in [169] reduces to the chaining problem. Given a set of 2-dimensional fragments, the single coverage problem consists in finding a set of fragments of maximum coverage with respect to one sequence without overlaps in the other sequence. The resulting fragments are *not* necessary colinear and their segments can overlap in the second sequence, but not in the first one. It can be be proved that the optimal solution of the *single coverage problem* can be found by solving the 1-dimensional chaining problem for the set of intervals that represent the projections of the 2-dimensional fragments on the x-axis [149].

In [183] the *transformation distance* is introduced to address the question of how a sequence T_2 can be derived from the sequence T_1 with the minimum number of segment-based genome rearrangement operations (segment-copy, reverse-copy, and insertion). As shown in [183], computing the transformation distance reduces to solve the 1-dimensional global chaining problem, where the gap between the fragments are properly penalized.

Two other problems, apparently unreleated with the global chaining problem are the *longest increasing subsequence problem*, LIS for short, and the *heaviest increasing subsequence problem*, HIS for short. In this context “subsequence” is not synonymous of “substring”, because while the characters of a substring are required to occur *contiguously* in the input string, the characters of a subsequence can be interspersed with characters that are not in the subsequence. Let T a sequence of n integers, not necessarily distinct. An *increasing subsequence* of T is a subsequence of T whose values *strictly* increase from left to right [88]. Given T, the longest increasing subsequence problem consists in finding the increasing subsequence of the longest size. If every number $T[i]$ has a weight $T[i]$.`weight`, and the weight of a subsequence is the sum of the weights of its character, then the heaviest increasing subsequence problem consists in finding the increasing subsequence having maximal weight [101]. Clearly, the LIS problem is a special case of the HIS problem where the weight of all the integers is 1. The HIS problem is a special case of the 2-dimensional global chaining problem. In fact, given the sequence T, if we construct the set of fragments $F = \{(1, T[1]), \ldots, (n-1, T[n-1])\}$ where the i-th fragment has weight $T[i]$.`weight`, each optimal 2-dimensional global chain of F corresponds to a heaviest increasing subsequence in T.

Given two (possibly incomplete) genomic sequences of two related organisms, one frequent question is that of computing how much of one sequence is not represented in the other sequence, under the restriction that positions in each sequence can be involved in at most one selected match [90]. In other words, we want to find a set of non-overlapping fragments such that the amount of sequence covered by the fragment is maximized. The fragments are not required to be colinear.

In the terminology of graph theory, this problem is to find a *maximum weight independent set*, MWIS for short, in a particular type of intersection graph where for each input fragment f_i there is a vertex having weight f_i.`weight`. The vertices corresponding to f_i and f_j are connected by an edge, if f_i and f_j overlap. If in the input graph a weight is associated with each vertex, the *maximum weight independent set problem* is to find an independent set of maximum weight. In [12] it has been showed that MWIS problem for fragments is $\mathcal{NP}$-complete. Even worse, this problem was recently shown to be $\mathcal{APX}$-hard in [13], although some idea for approximation algorithms are also discussed in the same paper.

An interesting variation of the MWIS, that is called *maximal matched sequence problem* (MMSP), has been studied in [90]. Given a set of fragments, the MMSP consists in computing a set non-overlapping set of sub-fragments such that the amount of covered sequence is maximized. The authors of [90] showed that this problem can be optimally solved in polynomial time.

Two are the main computational approaches to solve the global chaining problem: the graph-based approach and the geometric approach.

A comprehensive description of these algorithms is beyond the scope of the current section, and we refer the interested reader to [46, 126] for the description of the graph-based approach, and to [149, 88, 102] for the geometric approach. However, in the rest of this section we briefly sketch the ideas behind these computational approaches.

Given the input set of weighted fragments $S = \{f_1, \ldots, f_m\}$, in the graph-based approach we construct a weighted directed acyclic graph $G = (V, E)$, where the set V of vertices consists of all fragments, including $\mathbf{0}$ and $\mathtt{t}$, and there is an edge $(i, j) \in E$ with weight $w_{ij} = f_j.\mathtt{weight} - g(f_i, f_j)$ if $f_i \ll f_j$. An optimal global chain of fragments corresponds to a path of maximum score from $\mathbf{0}$ to $\mathbf{t}$ in the graph. Let $f_i.\texttt{score}$ the maximum score of all the chains starting at $\mathbf{0}$ that end at f_i. Given the graph acyclicity, $f_i.\texttt{score}$ can be computed by the formula $f_i.\texttt{score} = f_i.\texttt{weight} + \max\{f_j.\texttt{score} - g(f_i, f_j) : f_j \ll f_i\}$ (we also set $\mathbf{0}.\texttt{score} = 0$).

If computing gap costs takes constant time, a dynamic programming algorithm based on the above recurrence takes $\mathcal{O}(m^2)$ time and $\mathcal{O}(m)$ space. This graph-based solution can work for multiple input sequences, and for any kind of gap cost.

In the geometric approach the $\mathcal{O}(m^2)$ time bound is improved by exploiting the geometric nature of the problem. As explained above a fragment f of k genomes can be represented by a hyper-rectangle in $\mathbb{R}^k$ with the two extreme corner points $\texttt{beg}(f)$ and $\texttt{end}(f)$. For example, in Figure 4.3 there are two input sequences, hence the fragments are represented as 2-dimensional rectangles.

The main ingredients of the geometric approach to global chaining, are *range maximum queries*. Given a set S of points in $\mathbb{R}^2$, with a score associated with each point, a *range maximum query* (RMQ) asks for a point of maximum score in a rectangle $R(p, q)$.

If the gap cost function is the constant function 0, the recurrence $f_i.\texttt{score} = f_i.\texttt{weight} + \max\{f_j.\texttt{score} - g(f_i, f_j) : f_j \ll f_i\}$ can be rewritten as $f_i.\texttt{score} = f_i.\texttt{weight} + f_j.\texttt{weight}$ where f_j is the fragment whose end point is returned by $RMQ(R(\mathbf{0}, \texttt{beg}(f_i) - \overrightarrow{1}))$. For example, consider Figure 4.3. The score of the top-scoring chain ending at f_3, is computed by adding $f_3.\texttt{weight}$ to the score of the top-scoring chain that is contained in the rectangle $R(\mathbf{0}, \texttt{beg}(f_3)) - \overrightarrow{1})$ (highlighted in dashed lines). As discussed in [149], given m input fragments, this algorithm requires $\mathcal{O}(m \log m)$ time and $\mathcal{O}(m)$ space to compute the optimal global chain of fragments.

An ad-hoc variant of the above algorithm, having the same complexity bounds of the above algorithm, which only works with two input sequences, is described in [88].

The geometric approach to global chaining can be generalized to k dimensions by using *range trees* and *kd-tree* geometric data structures to answer to the RMQ queries, and can also be generalized to handle different gap cost functions [149]. However, the $\mathcal{O}(m \log^{k-2} m \log\log m)$ time complexity of the geometric approach in the general case of k input sequences, even without gap costs, makes it impractical, even for a moderate number of input sequences.

4.6 Regender

Differently than previous tools cited in Section 4.5, which are general-purpose, in the sense that they do not perform any assumption about the input data, we would like to exploit the structural properties of the sequences of our dataset, described in Section 4.2.

As we know from Section 4.2 our dataset contains 38 low-coverage strains, each one with 16 chromosomes. While the coverage of 38 strains in [130] is low (one-to-fourfold), and the fraction of unresolved bases in these chromosomes is relevant (see Section 4.2 for details), the `RefSeq` chromosomes, that are available at the SGD database [170] have been sequenced at high coverage, and contain no unresolved bases at all. It follows that, while the alignment between two (or more) of the 38 low coverage strains can be very difficult, if at least one of the input sequences to be compared does not contain unresolved bases, we can handle the unresolved segments in the other strain. Consider the example in Figure 4.2. Although, the occurrence of `YARCTy1-1` is masked in the `s288c` strain by three unresolved segments, the fact that the regions preceding and following the `YARCTy1-1` transposon in `RefSeq` are conserved, the two conserved segments inside the transposon, and the “parallelogram shape” of the three non-conserved white regions inside the transposon suggesting that no insertion or deletion occurred, are all evidences of the conservation of the transposons in the `s288c` strain. In some sense, the global alignment suggests that the content of the unresolved segments in `s288c` is the same as that of the corresponding segments in `RefSeq`.

Due to their exponential time dependency on the number of input sequences, multiple alignments/chaining algorithms cannot be applied to dataset containing more than few sequences (our dataset contains 38 strains, plus `RefSeq`!). Although Section 4.4 discusses the limit of the global alignment even in the case of two sequences, Section 4.5 shows how we can overcome the above limitations through chaining algorithms, which can be used to efficiently compare a pair of sequences. Hence, we chose a “star” comparison strategy. We performed a pairwise comparison between the chromosomes of the `RefSeq` strain and the homologous chromosomes of the other 38 low-coverage strains.

When comparing a `RefSeq` chromosome with the homologous chromosome of another strain (e.g. `Chr` 1 of `RefSeq` with `Chr` 1 of `Y55`, `Chr` 2 with `Chr` 2, etc), the assumption that the number of substitutions, insertions, deletions of one or few bases is small sounds realistic, due to the high similarity of the compared strains. Under this assumption, the choice of exact L-grams, instead of more flexible motifs as basic building blocks of the output alignment is a reasonable tradeoff between computational efficiency, and accuracy of the output alignment. Moreover, if few large scale rearrangement events occurred in the input pair of chromosomes, we can aggregate consecutive L-grams in a linear time greedy fashion, instead of using a chaining algorithm requiring $\mathcal{O}(r \log r)$ time, where r is the number of common L-grams detected during the first step of the algorithm.

Although the above assumptions seem to be realistic (we are not comparing mouse and human chromosomes, but we are comparing different strains of *S. cerevisiæ*), only an empirical data analysis of the input dataset can prove the validity of these assumptions.

4.6.1 Preliminary data analysis

Our approach is driven by data analysis. We performed a preliminary study to understand how to grasp the high similarity in our dataset. Consider the pair of homologous chromosomes ($\mathtt{Chr}N_{\mathtt{A}}, \mathtt{Chr}N_{\mathtt{B}}$) where `A` is `RefSeq B` is one of the other 38 low-coverge strains, and $1 \leq N \leq 16$ ranges over one of the 16 chromosomes of the *S. cerevisiæ* dataset. Examine all the possible (overlapping) L-grams of $\mathtt{Chr}N_{\mathtt{B}}$ as candidates, where an L-gram is a segment of L consecutive bases.

Assuming that $\mathtt{Chr}N_{\mathtt{B}}$ contains m bases, there are $m-L+1$ L-grams, accounting for possible duplicates. Call *valid* the L-grams that do not contain any symbol `N`. The *common* L-grams are the valid L-grams that occur *exactly* (i.e. fully conserved with no mutation) both in $\mathtt{Chr}N_{\mathtt{A}}$ and $\mathtt{Chr}N_{\mathtt{B}}$.

N	bases	(a)	(b)	(c)	N	bases	(a)	(b)	(c)
1	248 261	81.32	59.92	0.43	9	467 776	89.53	74.02	0.29
2	800 992	98.54	82.55	0.14	10	770 597	94.60	76.43	0.62
3	321 691	93.82	83.74	2.13	11	693 726	97.64	78.98	0.11
4	1 522 688	96.24	77.38	0.89	12	1 067 059	95.12	78.66	2.15
5	577 152	96.33	77.66	0.39	13	923 317	96.96	84.35	0.52
6	273 660	97.56	76.05	0.19	14	781 629	98.20	82.46	0.15
7	1 113 452	95.05	78.63	0.73	15	1 105 914	95.67	81.07	0.33
8	566 494	95.47	78.70	0.84	16	946 183	96.53	83.55	0.65

Table 4.4: Statistics (%) for the L-grams ($L = 32$) satisfying properties (a)–(c). The length of each chromosome in `Y55` is also given.

In our experiments, $L = 32$ resulted to be a good choice, leading to the following empirical facts that were observed for chromosomes $N = 2, 3, \ldots, 16$, with chromosome $N = 1$ (whose percentages are shown inside parentheses below) being an outlier. The reported percentages are absolute, as they are obtained by dividing the number of wanted L-grams by $m - L + 1$.

(a) The *valid* L-grams are numerous: they are in the range 89.53%–98.54% in `Y55` (81.32% for `Chr` 1).

(b) The *common* L-grams are also numerous: they are between 74.02%–84.35% in `Y55` (59.92% for `Chr` 1).

(c) The common L-grams that occurs *once* in each genome are the vast majority: indeed, those occurring two or more times are very few, between 0.11%–2.15% in `Y55`.

A summary reporting the above percentages for the L-grams in the 16 chromosomes of the `Y55` strain is shown in Table 4.4 (the results for the other strains are almost identical). This high similarity can be biologically explained recalling that we are not comparing chromosomes of different species, but instead we are comparing homologous chromosomes of different strains of the same species.

The implication of (a)–(c) is that due to the high similarity between `RefSeq` chromosomes and the homologous ones of the other strains, the localization of the conserved regions can be performed by using the common L-grams instead of the more flexible and inefficient approaches based on approximate L-grams described in Section 4.5.

4.6.2 Two phases approach

Our algorithm for the rapid detection of large highly-conserved segments, called `Regender` (REsident GENome DEtectoR), is driven by the above data analysis. It performs a two-phase processing of all the possible pair of chromosomes ($\mathtt{Chr}N_{\mathtt{A}}, \mathtt{Chr}N_{\mathtt{B}}$) in our dataset (we recall that the dataset contains 39 strains, and each strain has 16 chromosomes).

In the first phase, `Regender` finds the common L-grams between $\mathtt{Chr}N_{\mathtt{A}}$ and $\mathtt{Chr}N_{\mathtt{B}}$. In the second phase, `Regender` aggregates consecutive L-grams in a greedy fashion using some user-defined parameters that control when the next conserved region begins in both $\mathtt{Chr}N_{\mathtt{A}}$ and $\mathtt{Chr}N_{\mathtt{B}}$. We provide the details of the algorithm in Section 4.6.3.

`Regender` is related to the *anchor-based* algorithms [149] that circumvent the quadratic costs (time and space) of the traditional algorithms for sequence alignment [88]). As discussed in Section 4.5, these algorithms after detecting the fragments that are common to both $\mathtt{Chr}N_{\mathtt{A}}$ and $\mathtt{Chr}N_{\mathtt{B}}$, select a subset of colinear fragments that will be the matching regions of the output alignment, and finally apply an expensive dynamic programming scheme to the regions of $\mathtt{Chr}N_{\mathtt{A}}$ and $\mathtt{Chr}N_{\mathtt{B}}$ that are left uncovered by the anchors.

Driven by our data analysis, `Regender` can go simpler. First, the L-grams of $\mathtt{Chr}N_{\mathtt{A}}$ are stored in a hash table, and those of $\mathtt{Chr}N_{\mathtt{B}}$ are searched in the table during a scan of $\mathtt{Chr}N_{\mathtt{B}}$. The high similarity of $\mathtt{Chr}N_{\mathtt{A}}$ and $\mathtt{Chr}N_{\mathtt{B}}$ justifies our choice of exact L-grams as fragments. Second, our dataset gives almost surprisingly a natural set of anchors: contrarily to the anchor-based algorithms, we do not need any dynamic programming or chaining techniques to enforce the colinearity and the non-overlapping property, since there is almost a one-to-one mapping between the occurrences of the L-grams (see Section 4.6.1). Actually, we take advantage of the fact the L-grams overlap and, if they are not colinear, we get a hint for a possible translocation. As a result, `Regender` performs just a scan of $\mathtt{Chr}N_{\mathtt{A}}$ and $\mathtt{Chr}N_{\mathtt{B}}$. One execution of `Regender` takes less than a second on a standard PC with limited amount of memory. This is a major requirement, since we need to execute `Regender` for all pairs of homologous chromosomes of $\mathtt{Chr}N_{\mathtt{A}}$ and $\mathtt{Chr}N_{\mathtt{B}}$. Third, we remark that we do not need a complete alignment of $\mathtt{Chr}N_{\mathtt{A}}$ or $\mathtt{Chr}N_{\mathtt{B}}$ for the purposes of the analysis performed in this chapter. A high-quality alignment of the conserved regions in $\mathtt{Chr}N_{\mathtt{A}}$ or $\mathtt{Chr}N_{\mathtt{B}}$ is unnecessary in our case, as illustrated by the clear patterns emerging in Figure 4.4, where although the occurrence of the transposon `T` is masked by unresolved bases in the `Y55` strain (on bottom), the transposition of the transposon `T` is detected by the insertion and the deletion events in the `Y55` strain. What we really care about is the description of the dynamics of the mobilome, identifying and locating all the mobile elements in the input sequences, together with the genomic rearrangements they are involved into. A merit of our approach is that of being able to select a small set of candidates for the latter investigation, as discussed next.

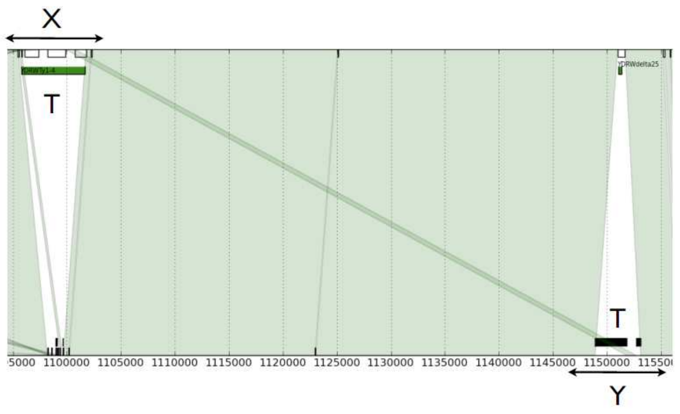


Figure 4.4: A plot of the common L-grams for `Chr` 4 (1,095,000–1,155,000) of `RefSeq` (top sequence), and `Y55` (bottom sequence), where $L = 32$. Each line connects the starting positions of a common L-gram. Thus, the empty triangles or trapezoids represent non-conserved regions. Annotated mobile elements are represented by the green rectangles just below the top line. Unresolved sequences are the black rectangles just above the bottom line. High resolution plots are available at `www.di.unipi.it/~gbattag/regender`.

4.6.3 Algorithm and implementation

As previously mentioned, we exploit the high similarity between genomes of different strains by running a massive computation involving all the chromosome pairs $(\texttt{Chr}N_{\texttt{A}}, \texttt{Chr}N_{\texttt{B}})$, where the first is a `RefSeq` chromosome, while the second is one of the other 38 low-coverage strains of the dataset.

We follow a two-phase approach for `Regender`, whose inputs are two chromosomes $\texttt{Chr}N_{\texttt{A}}$ and $\texttt{Chr}N_{\texttt{B}}$, the length L of the grams, and two user-defined parameters δ_1 and δ_2 to be used in the second phase. First, we find all the common L-grams between $\texttt{Chr}N_{\texttt{A}}$ and $\texttt{Chr}N_{\texttt{B}}$. Second, we detect highly conserved regions by aggregating consecutive L-grams. Finally, we inspect the non-conserved regions that are found by `Regender`, so as to infer mobilome elements.

Phase 1 of `Regender`: common L-grams. We aim at finding which L-grams of $\texttt{Chr}N_{\texttt{B}}$ occur inside $\texttt{Chr}N_{\texttt{A}}$, where an L-gram is any sequence of L consecutive bases. First, we construct a dictionary for all the L-grams in $\texttt{Chr}N_{\texttt{A}}$ and, then, we search for the L-grams of $\texttt{Chr}N_{\texttt{B}}$ inside the dictionary. This task can be performed in expected linear time by employing a rolling hash approach based on cyclic polynomial, as described in [44]. Note that using a general purpose hash function would be more expensive by a multiplicative factor of L. Also, using a trie-based dictionary instead of hashing would guarantee a linear-time worst-case performance, but hashing is faster in practice.

A detailed description of the rolling hashing is beyond the scope of the current section. However, the main idea behind this approach is simple. Let assume that each of the four bases, say c, is mapped into a 32-bit integer h_c. Moreover, let us denote the bit-wise exclusive or by $\oplus$. Let $s(-)$ be the cyclic binary rotation function, which shifts the input bit string to the left, moving the leftmost bit in the rightmost position. For example, $s(\texttt{10110}) = \texttt{01101}$. We use $s^i(-)$ to indicate $s(-)$ iterated i times on the input value. For example, $s^2(\texttt{10110}) = s(\texttt{01101}) = \texttt{11010}$.

Given the input L-gram $t = t[1]t[2]\cdots t[L]$, its hash value is $h(t) = s^{L-1}(h_{t[1]}) \oplus s^{L-2}(h_{t[2]}) \oplus \ldots \oplus s(h_{t[L-1]}) \oplus h_{t[L]}$. The resulting value is represented by a 32-bits integer. Computing the hash values in a rolling fashion is done as follows. Suppose $t' = t[2] \ldots t[L+1]$ is the L-gram following t. To quickly compute $h(t')$ from $h(t)$, we only need to remove the base $t[1]$ and add the new base $t[L+1]$. First, the previous hash value is rotated one position to the left, obtaining $h'' = s(h(t))$. Then, the new hash value is $h(t') = h'' \oplus s^L(h_{t[1]}) \oplus h_{t[L+1]}$.

Some care is required in handling "unresolved" bases, denoted by `N`, in the input chromosomes. Since the rolling hash approach cannot handle them, when moving the sliding window of length L from left to right, we consider the maximal runs of consecutive bases different from `N`, provided that they are of length at least L (otherwise, they cannot contain any valid L-gram inside). In this way, we can amortize the $\mathcal{O}(L)$ initialization cost for the rolling hash, with the run length. The linear average-case cost justifies our choice of the rolling hash approach. In fact, assuming that the lookup operation takes constant time, the cost to create the hash table becomes predominant in the time complexity.

Lemma 33. *The first phase of the algorithm* ***`Regender`*** *requires* $\mathcal{O}(|\texttt{Chr}N_{\texttt{A}}|+|\texttt{Chr}N_{\texttt{B}}|)$ *time on average.*

The output of the first phase is a mapping M, associating each L-gram s_2 of $\texttt{Chr}N_{\texttt{B}}$, with its occurrence list $occs(s_2)$ in $\texttt{Chr}N_{\texttt{A}}$. If s_2 does not occur in $\texttt{Chr}N_{\texttt{A}}$, $occs(s_2)$ is empty. Although not optimal in the worst case, our hash based approach turned out to be effective on our datasets, yielding few collisions, and allowing us to compare two entire chromosomes in few seconds.

Phase 2 of `Regender`: conserved regions. During the second phase, the information about the L-gram occurrences, stored in the mapping M computed in the first phase, is used to establish a correspondence between segments of consecutive bases in $\texttt{Chr}N_{\texttt{B}}$ and $\texttt{Chr}N_{\texttt{A}}$, mapping a segment $I_2 = \texttt{Chr}N_{\texttt{B}}[l_2, r_2]$ into a corresponding segment $I_1 = \texttt{Chr}N_{\texttt{A}}[l_1, r_1]$. This information is represented by the mapping M_2, and it is graphically shown with green lines in Fig. 4.4.

We perform a left-to-right scan of $\texttt{Chr}N_{\texttt{A}}$ and $\texttt{Chr}N_{\texttt{B}}$, according to the following greedy rule. Initially, I_1 and I_2 are empty. During the scan, the current segments I_1 and I_2 are extended when the following conditions are met:

- There exists a common L-gram s, which occurs both to the right of I_1 and I_2, and no other L-gram with this property can be found between I_1 and s, and I_2 and s;
- Letting d_1 be the number of bases between I_1 and s, and d_2 be the number of bases between I_2 and s, it is $|d_1 - d_2| \leq \delta_2$ and $d_2 \leq \delta_1$ (hence, $d_1 \leq \delta_1 + \delta_2$).

To describe the main steps, assume that the first $j - 1$ bases of $\texttt{Chr}N_{\texttt{B}}$ have already been processed, and that M_2' is the mapping constructed so far. To add the next pair of intervals to M_2', the main steps are as follows:

(1) *Starting point search.* The starting point of the next segment is set to the coordinate of the leftmost L-gram (say j_1) that does not belong to any previously mapped interval in M_2', and that occurs at least once in $\texttt{Chr}N_{\texttt{A}}$ (i.e. $M(\texttt{Chr}N_{\texttt{B}}[j_1 \ldots j_1 + L - 1]) \neq \varnothing$). Let $L_1 = \{i_1, \ldots, i_p\}$ be the nonempty occurrence list $occs(s_2)$ in $\texttt{Chr}N_{\texttt{A}}$, where $s_2 = \texttt{Chr}N_{\texttt{B}}[j_1 \ldots j_1 + L - 1]$. Among all the identical L-grams in L_1, we map s_2 into the nearest one. Namely, we select $i^* = argmin_{i \in L_1}\{|j_1 - i|\}$. Note that L_1 is a singleton list in the majority of cases in our dataset. In the rest of the current section, i^* will be referred as the *image* of j_1. If s_2 and its corresponding occurrence at coordinate i^* of $\texttt{Chr}N_{\texttt{A}}$ cannot be found, all the segments have been already reported, and the mapping M_2' is returned.

(2) *Segment extension.* Once a starting point j_1 together with its image i^* has been selected, the first L-gram $s_2 = \texttt{Chr}N_{\texttt{B}}[j_1 \ldots j_1 + L - 1]$ is added to the new segment. At this point, the next L-gram $s_2' = \texttt{Chr}N_{\texttt{B}}[j_2 \ldots j_2 + L - 1]$ is examined, along with its occurrence list $L_2 = \{k_1, \ldots, k_l\}$ mapped by M. An occurrence k^* that satisfies the following conditions is selected from L_2. First, the maximum number of bases between s_2 and s_2', must be less than or equal to the user-defined threshold δ_1. In other words, it must be $d_2 \leq \delta_1$ where $d_2 = j_2 - j_1 - L$. Second, since s_2 precedes s_2' in $\texttt{Chr}N_{\texttt{B}}$, we require that the image of s_2 in $\texttt{Chr}N_{\texttt{A}}$, namely $s_1 = \texttt{Chr}N_{\texttt{A}}[i^* \ldots i^* + L - 1]$, precedes the image of s_2' in $\texttt{Chr}N_{\texttt{A}}$, $s_1' = \texttt{Chr}N_{\texttt{A}}[k^* \ldots k^* + L - 1]$. Hence, we require that $i^* < k^*$. Finally, we aim at mapping two L-grams that occur closely into $\texttt{Chr}N_{\texttt{B}}$, into L-grams occurring closely in $\texttt{Chr}N_{\texttt{A}}$. We constraint the difference of their distance to be within the user-defined threshold δ_2: it must be $|d_1 - d_2| \leq \delta_2$, where $d_2 = j_2 - j_1 - L$, and $d_1 = k^* - i^* - L$.

If an occurrence of s_2' satisfying the above conditions is found, the L-gram s_2' is added as an extension to the current segment. The above steps are repeated to find a new L-gram following s'_2 in $\texttt{Chr}N_{\texttt{B}}$, and satisfying the above conditions. On the other hand, if s'_2 does not satisfy the above conditions, then the next L-gram, s''_2, mapped by M into a nonempty occurrence list is selected, and an occurrence satisfying the above conditions is looked for. If such an L-gram cannot be found, the extension phase terminates.

(3) *Mapping update.* Let $s_2 = \texttt{Chr}N_{\texttt{B}}[j_1 \ldots j_1 + L - 1]$ and $s'_2 = \texttt{Chr}N_{\texttt{B}}[j_2 \ldots j_2 + L - 1]$ be the first and the last L-gram of the current segment, and $s_1 = \texttt{Chr}N_{\texttt{B}}[j^* \ldots j^* + L - 1]$ and $s'_1 = \texttt{Chr}N_{\texttt{B}}[k^* \ldots k^* + L - 1]$ be their corresponding occurrences selected in the previous two steps (where it can be $s_1 = s_2$). The current mapping M'_2 is updated by adding the correspondence between segments $\texttt{Chr}N_{\texttt{B}}[j_1, j_2 + L - 1]$ and $\texttt{Chr}N_{\texttt{A}}[i^*, k^* + L - 1]$.

Steps (1)–(3) are repeated until a new segment is found. At the end, the whole mapping M_2 for the conserved regions (anchors) is returned.

To compute the time complexity of the second phase of `Regender` algorithm, we observe that the sum of the sizes of the occurrence lists in M is upper bounded by $|\texttt{Chr}N_{\texttt{A}}| - L + 1$. In other words, the size of the mapping M is $\mathcal{O}(|\texttt{Chr}N_{\texttt{A}}| + |\texttt{Chr}N_{\texttt{B}}|)$. Steps (1)–(3) can be implemented by a left-to-right scan of the chromosomes.

Lemma 34. *The second phase of the algorithm* `Regender` *requires* $\mathcal{O}(|\texttt{Chr}N_{\texttt{A}}| + |\texttt{Chr}N_{\texttt{B}}|)$ *time.*

Theorem 35. *Algorithm* `Regender` *requires* $\mathcal{O}(|\texttt{Chr}N_{\texttt{A}}| + |\texttt{Chr}N_{\texttt{B}}|)$ *time on average.*

4.7 Experimental Results

To show the correctness of the idea of identifying the transposons by detecting non-conserved mobile elements we implemented a prototype of the `Regender` tool, and we analyze the dataset of 38 strains of *S. cerevisiæ* described in [130].

Due to the size of the dataset (the genome of each strain is long about 12 Mb, while the whole dataset is about 500 Mb), the selected technique is required to be efficient. We already discussed in Section 4.6 the time complexity of `Regender`. Section 4.7.1 empirically compares `Regender` with the state of the art tools that are reported in Table 4.5.

The second issue that we investigated is the output quality. It is worth nothing that `Regender` is fast in returning a poor quality output where few conserved segments are found, or non conserved segments are tagged as conserved. Section 4.7.2 compares the output of `Regender` with the output of the other tools, to investigate how much the choice of exact L-grams penalizes the quality of the output.

The remaining section is intended to validate our methodology for transposon detection by using the known transposons annotations that are available at the SGD database [170]. In Section 4.7.3, we discuss what is the relationship between the non-conserved (mobile) part of each chromosome, and its transposons.

Tool	Frag. Detection	Anchor Selection	Align. Completion
Avid [35]	MEM/Suff. Tree	Variant of Smith-Waterman	Dyn. Progr.
GS-Aligner [173]	Hashing	Heuristic	Dyn. Progr.
Lagan [38]	Aho-Corasick	Longest Incr. Subseq	Dyn. Progr.
Lastz [168]	Hashing	See [168]	Dyn. Progr.
Mga [97]	MUM/Suff. Tree	Longest Incr. Subseq	Dyn. Progr.
Mummer [53]	MUM/Suff. Tree	Longest Incr. Subseq	Dyn. Progr.
Ssaha [147]	Hashing	–	–
Murasaki [160]	Hashing	Heuristic	–
Blastn [7]	Hashing	–	–
Regender	Hashing	Heuristic	–

Table 4.5: Summary of the tools `Regender` is compared with. The second column reports the type of fragments generated by the tool, and the technique used to compute them. The third column reports the techniques used to select the anchors, while the fourth column reports the techniques used to close the gaps in between the selected anchors. Not all the tools perform the anchor selection step, and the final step.

4.7.1 `Regender` performance

To asses the efficiency of `Regender`, we compared our tool with the state of the art genome comparison tools reported in Table 4.5.

The tools reported in Table 4.5 are a small part of all the tool/libraries available for whole genome alignment/comparison. However, the tools that we selected for our experimental comparison have to satisfy some requirements:

- The tool must be available online, at the site referred in the paper describing the tools (in the case of broken links, we contacted the authors).
- Free of charge: no proprietary software is took into account.
- If available, we used the executable. Otherwise, we also accepted the source code. However, in case of non trivial compile-time errors the tool is not taken into account.
- The tools must be well documented. We adopted a black-box approach, so we are not interested in the tool internals, but the input parameters and the output format must be properly described.
- Software security is a fundamental aspect in software development. We aim at comparing mature and robust software tools, hence we do not consider tools crashing at runtime, or running out of memory due to the size of the input dataset.

Table 4.5 lists the tools that are part of our experimental investigation. Not all the tools reported in Table 4.5, perform all the three computational steps described in Section 4.5. More precisely, while `Avid` [35], `GS-Aligner` [173], `Lagan` [38], `Lastz` [168], `Mga` [97], `Mummer` [52] performs all the three steps, `Murasaki` [160] and

`Regender` skip the final step. `Blastn` [7] and `Ssaha` [147] only compute the set of fragments of the input sequences. However, we included these tools in our comparison because of their relevance to the bioinformatics community. To compare the output of these tools with the other, we implemented a 2-dimensional global chaining algorithm, that is a modified version of that in [88], computing an optimal global chain given an input set of 2-dimensional fragments.

Chrm.	Avid	GS-Aligner	Lagan	Lastz	Mga	Mummer	Ssaha	Murasaki	Blastn	Regender
1	6.32	3.51	20.69	3.40	0.68	0.69	1.05	8.14	0.55	0.72
2	24.27	21.71	35.94	11.31	4.01	1.55	2.27	14.02	1.39	2.31
3	9.21	5.26	12.11	4.65	1.91	0.71	1.12	9.18	0.70	1.02
4	58.85	60.64	73.44	36.36	5.41	3.47	4.69	22.88	2.93	4.88
5	28.76	12.73	28.31	9.73	3.98	1.14	1.67	11.65	1.14	1.60
6	7.42	4.64	10.77	3.52	0.99	0.65	1.01	8.48	0.57	0.89
7	35.60	34.99	55.76	21.76	4.44	2.27	3.04	17.35	2.08	3.32
8	26.43	11.65	24.78	8.52	1.88	1.15	1.65	11.32	1.03	1.57
9	15.76	8.21	21.70	5.77	3.21	0.88	1.47	10.22	0.81	1.24
10	71.79	18.46	43.59	9.51	4.18	1.46	2.01	13.08	1.33	2.20
11	53.92	15.86	26.44	9.92	1.70	1.38	1.93	12.66	1.22	2.07
12	34.37	32.43	67.83	16.62	3.74	2.42	2.97	17.22	2.04	3.23
13	28.20	26.12	46	16.16	8.50	1.81	2.42	15.14	1.68	2.96
14	35.48	18.91	36.88	7.21	5.65	1.53	2.10	13.63	1.37	2.28
15	35.34	33.65	59.48	18.88	2.82	2.16	2.95	17.05	2.01	3.31
16	28.76	26	70.61	16.29	3.20	1.87	2.45	15.38	1.70	2.95

Table 4.6: Time comparison between all the tools reported in Table 4.5. Each value is the average running time (in seconds) of all the 38 pairwise comparisons between a `RefSeq` chromosome and the homologous chromosomes of the other 38 low coverage strains. The experiments have been performed on an Intel Core 2 Duo P8400 notebook, with 4GB of RAM. The maximum amount of RAM available for the first phase of `Regender` has been set to 200MB. The size of the L-grams has been set to 32, while δ_1 and δ_2 have been set to 100.

Table 4.6 reports the results obtained by running the tools in Table 4.5 on all the chromosomes of the *S. cerevisiæ* dataset in [130]. More precisely, we adopted a star comparison strategy, by performing a pairwise comparison between each one of the chromosome of the `RefSeq` strain with the homologous chromosomes of the other 38 low-coverage strains. In other words, we compared `Chr` 1 of `RefSeq` strain, with the `Chr` 1 of the `Y55` strain, with the `Chr` 1 of the `YPS128` strain, etc.

Table 4.6 shows the results aggregated by chromosome. Each value is the average running time (in seconds) of all the 38 pairwise comparisons between a `RefSeq` chromosome and the homologous chromosomes of the other 38 low-coverage strains. As we can see from the table, although the `Regender` prototype has been implemented in Python and Java, and the code has not been fine tuned or engineered at all, it is competitive with the other tools. Two tools are only faster than `Regender`: `Mummer`, who on average needs the 70% of the time required by `Regender`, and `Blastn` which is the faster tool and requires 63% of `Regender` time. Among the remaining tools only `Ssaha` can compete with `Regender` requiring almost the same amount of time (100.35%). The other tools shows poorer performances, and are five (`Lastz`) to seventeen (`Lagan`) times slower than `Regender`. This is not surprising, since `Regender` does not require to perform a high-quality alignment of the input chromosomes as the other tools do (as explained in Section 4.6.1 this is not necessary in identifying and locating the mobile elements in the input sequences).

4.7.2 `Regender` output quality

Chrm.	`Avid`	`GS-Aligner`	`Lagan`	`Lastz`	`Mga`	`Mummer`	`Ssaha`	`Murasaki`	`Blastn`	`Regender`
1	81.49	68.86	81.32	75.15	24.56	75.89	77.29	68.72	77.10	78.33
2	96.81	77.50	96.78	72.76	31.58	73.29	95.56	70.61	95.27	96.40
3	95.47	74.48	95.23	75.44	40.03	74.07	90.14	69.90	90.25	94.66
4	94.98	77.67	94.64	78.24	35.55	78.99	93.77	69.27	92.96	94.31
5	94.59	75.81	94.32	76	36.08	66.24	92.71	68.35	90.55	93.97
6	93.71	81.85	93.82	64.89	26.25	63.88	91.17	66.86	89.25	92.25
7	94.94	79.86	94.93	84.05	33.78	84.45	93.41	69.74	92.44	94.34
8	93.85	78.67	94.08	79.78	34.49	78.72	91.56	66.17	92.02	92.52
9	93.07	80.07	93.52	83.79	28.98	83.34	91.15	70.75	90.02	92.02
10	93.16	75.93	92.94	86.43	30.52	81.12	91.97	69.04	89.67	92.45
11	98.26	81.45	98.35	77.25	25.72	76.58	94.73	71.73	93.72	96.96
12	93.42	75.07	93.37	70.30	30.28	73.05	92.16	69.51	91.25	92.69
13	96.04	75.61	95.37	82.41	37.81	80.34	94.77	69.18	94.30	95.57
14	96.31	72.68	96.37	87.64	40.82	85.36	94.75	72.46	94.03	95.76
15	95.81	77.85	95.74	81.66	27.66	79.01	94.36	69.99	94.40	95.28
16	95.35	74.43	94.75	81.79	38.72	74.17	93.16	68.26	92.79	94.99

Table 4.7: Percentages of bases tagged as conserved in each `RefSeq` chromosome. Each value is the average of all the 38 pairwise comparisons between a chromosome of `RefSeq` and the homologous chromosome of the other 38 low-coverage strains.

In Section 4.7.1, we showed that `Regender` is competitive with the other tool with respect to the execution time. What about the output quality?

For our purposes, we are not interested in comparing the alignment quality, but since we have to distinguish between conserved regions and non-conserved regions to detect the mobile elements, we are primarily interested in which bases are tagged as conserved and which are not. In other words, we are only interested in the *matching* character of each global alignment, which are the conserved bases, while all the other characters (insertions, deletions, mismatches) are considered as non-conserved characters.

In the current section, we compare the set of bases that are tagged as conserved by `Regender` with the set of conserved bases identified by the other tools.

First, we investigate how many bases are tagged as conserved by each tool. Table 4.7 report the percentages of bases tagged as conserved in each chromosome. Each value is the average on all the 38 pairwise comparisons between a chromosome of `RefSeq` and the homologous chromosome of the other 38 low-coverage strains. Values are aggregated by chromosome.

As we can see, on average, all the tools but `Mga` and `Murasaki` report a percentage of conserved bases that is comparable with that of `Regender`. Only `Avid` and `Lagan`, on average, identify a higher number of conserved bases than `Regender`, 101% and 100.9% respectively. `Ssaha` and `Blastn` report almost the same number of conserved bases as `Regender` (99% and 98% respectively), while the number of conserved bases reported by `GS-Aligner`, `Lastz`, `Mummer` is about the 83% of that reported by `Regender`.

While `Murasaki` tagging 75% of bases as conserved, it seems to show a lower sensitivity than the other tools, but still agrees on the high percentage of conserved bases, `Mga` completely disagrees with the other tools tagging on average only the 35% of the bases as conserved. Probably, this phenomenon is caused by the inadequacy of the default `Mga` parameters to our computational tasks (we recall that in all of our experiments we used the default parameters of each tool). However, the concordance of all the other tools in tagging more than 83% of bases as conserved suggests a scenario where the conserved bases are the vast majority of the bases, and the large-scale genomic rearrangement events are not frequent. (Recall that about 8% of bases are unresolved bases, so the true percentage of conserved bases could be even higher than 83%.)

From above, it follows that the number of conserved bases identified as conserved is almost the same for all the tools but `Mga`. But, do the tools identify the same set of conserved bases, or not? The answer to this question comes from Table 4.8 and Table 4.9, where we reported for each tool the percentage of bases that are tagged as conserved by `Regender` and also by the considered tool, the percentage of bases that are tagged as conserved by the tool but not by `Regender`, and the percentage of bases that are tagged as conserved by `Regender` but not by the tool. Also in this case the results are aggregated by chromosome.

As we can see the situation is similar to that in Table 4.7. Also Table 4.8 and Table 4.9 shows an almost perfect agreement between the set of conserved bases computed by `Regender` and `Avid` (99%), `Lagan` (98%), `Ssaha` (97%), and `Blastn` (97%). A good agreement is also shown when comparing `Regender` with `GS-Aligner`, `Lastz`, and `Mummer`. In all of these cases, the percentage of bases that is tagged as conserved by both tools is above 80%, on average. A poor agreement is shown with `Murasaki` (70%), but in particular with `Mga` (35%). This is a consequence of the small number of conserved bases detected by `Murasaki` and `Mga`.

These experimental results shows that `Regender` is not only fast, but also accurate. The `Regender` choice of exact L-gram, and the greedy heuristic used in the second step, suggested by the high-similarity of the input sequences, do not decrease the output quality of the tool.

4.7.3 Transposons and mobile segments

The results discussed in Section 4.7.2 shows a scenario where the large majority of bases is conserved (on average, more than 93%), and few large-scale rearrangement events occur in the chromosomes of the 38 strains that we compared with `RefSeq`. In the previous sections, we conjectured that the non-conserved segments of `RefSeq` are good candidates to be transposons. In the current section, we investigate this issue by comparing the set of bases that are tagged as non-conserved by `Regender` with the set of bases that are annoted as transposons in the SGD database [170].

Chrm	`Avid`			`GS-Aligner`			`Lagan`			`Lastz`			`Mga`		
1	95.67	4.10	0.22	79.80	4.27	15.94	94.90	4.39	0.71	86.46	4.84	8.70	30.63	0.10	69.27
2	99.47	0.48	0.05	79.60	0.45	19.95	99.33	0.53	0.14	74.95	0.29	24.75	32.65	0	67.35
3	98.67	1.09	0.24	76.77	1.10	22.13	98.32	1.14	0.54	78.32	0.79	20.89	41.96	0.02	58.02
4	98.78	0.97	0.26	80.71	0.91	18.38	98.12	1.11	0.76	81.33	0.90	17.78	37.59	0.06	62.35
5	98.97	0.84	0.18	79.44	0.72	19.83	98.56	0.90	0.54	79.05	0.97	19.98	38.14	0.04	61.82
6	98.04	1.76	0.20	85.60	1.65	12.75	98.10	1.78	0.12	68.32	1.28	30.40	28.36	0.01	71.63
7	98.97	0.84	0.20	83.39	0.69	15.92	98.60	1.01	0.39	87.97	0.63	11.41	35.74	0.03	64.23
8	98.22	1.60	0.18	82.49	1.40	16.11	98.16	1.75	0.09	83.97	1.24	14.79	37.19	0.05	62.76
9	98.30	1.41	0.28	84.72	1.21	14.07	98.24	1.68	0.08	89.04	1.08	9.89	31.30	0.03	68.67
10	98.94	0.91	0.15	80.90	0.69	18.41	98.28	1.12	0.60	92.27	0.62	7.10	32.80	0.04	67.16
11	98.55	1.39	0.06	81.68	1.26	17.06	98.50	1.46	0.04	78.80	0.52	20.68	26.39	0.11	73.50
12	98.65	1.06	0.28	79.56	0.80	19.64	98.36	1.19	0.46	74.21	0.90	24.88	32.70	0.01	67.29
13	99.31	0.59	0.10	78.30	0.47	21.22	98.49	0.65	0.86	85.51	0.37	14.11	39.38	0.02	60.60
14	99.23	0.67	0.10	74.83	0.60	24.57	99.21	0.71	0.08	90.50	0.52	8.97	42.50	0.02	57.48
15	99.14	0.71	0.16	80.59	0.62	18.79	98.95	0.77	0.29	84.63	0.58	14.80	29.04	0.01	70.95
16	99.33	0.52	0.15	77.59	0.43	21.99	98.64	0.55	0.80	85.25	0.44	14.31	40.77	0.01	59.22

Table 4.8: Comparison between `Regender` and `Avid`, `GS-Aligner`, `Lagan`, `Lastz`, `Mga`. For each tool, the percentage of bases that are tagged as conserved by `Regender` and also by the tool (the first column in each group of columns), the percentage of bases that are tagged as conserved by the tool but not by `Regender` (the second row in each group), and the percentage of bases that are tagged as conserved by `Regender` but not by the other tool (the third row in each group). Each value is the average on all the 38 pairwise comparisons between a chromosome of `RefSeq` and the homologous chromosomes of the other 38 low-coverage strains.

Chrm	`Mummer`			`Ssaha`			`Murasaki`			`Blastn`		
1	92.66	2.16	5.18	95.18	1.76	3.05	66.58	11.20	22.23	93.82	2.34	3.85
2	75.60	0.25	24.15	98.48	0.33	1.19	72.19	0.61	27.20	98.15	0.34	1.51
3	77.53	0.46	22.01	94.01	0.64	5.34	69.45	2.52	28.03	94.03	0.68	5.28
4	82.82	0.50	16.68	98.06	0.69	1.25	70.31	1.81	27.88	97.11	0.74	2.15
5	69.66	0.45	29.89	97.60	0.53	1.87	69.71	1.74	28.55	95.23	0.56	4.20
6	67.66	1.01	31.32	97.12	0.87	2.01	68.29	2.43	29.27	94.59	1.11	4.30
7	88.72	0.45	10.83	98.07	0.48	1.45	71.23	1.55	27.22	97.04	0.48	2.48
8	83.30	0.96	15.74	97.06	0.95	1.99	67.81	2.16	30.03	97.42	1.02	1.56
9	89.21	0.70	10.09	97.49	0.80	1.70	71.04	3.33	25.63	96.17	0.83	2.99
10	86.93	0.41	12.66	98.64	0.42	0.93	68.88	3.30	27.82	96.07	0.46	3.47
11	78.27	0.42	21.32	95.71	1.01	3.28	71.31	1.54	27.15	95.29	0.69	4.02
12	77.91	0.48	21.60	98.16	0.64	1.20	70.34	2.66	27	97.16	0.65	2.19
13	83.55	0.27	16.17	98.49	0.34	1.17	70.94	0.85	28.22	98	0.34	1.66
14	88.29	0.44	11.27	98.02	0.46	1.52	72.93	1.55	25.52	97.20	0.50	2.30
15	82.18	0.40	17.42	98.09	0.48	1.44	71.11	1.35	27.54	98.13	0.47	1.40
16	77.62	0.23	22.14	97.46	0.31	2.23	69.77	1.21	29.02	97.07	0.31	2.63

Table 4.9: Comparison between `Regender` and `Mummer`, `Ssaha`, `Murasaki`, `Blastn`. The statistics are the same as that in Table 4.8.

Chrm	Ann. Bases	Cons. Bases	Ann. and Cons.	Cons. and Ann.
1	6,220	130,089	4.66	0.22
2	21,216	752,938	11.65	0.33
3	10,553	273,337	25.95	1.00
4	53,289	1,399,072	8.57	0.33
5	19,415	477,475	49.14	2.00
6	8,256	104,934	17.45	1.37
7	43,622	945,080	12.57	0.58
8	10,674	452,179	28.86	0.68
9	7,893	352,702	12.50	0.28
10	22,473	588,669	63.81	2.44
11	3,975	510,221	46.94	0.37
12	35,712	987,072	30.02	1.09
13	28,246	860,357	11.32	0.37
14	20,465	699,772	5.09	0.15
15	29,428	933,192	14.23	0.45
16	35,560	877,694	28.43	1.15

Table 4.10: Overlaps between the transposons and the conserved regions in the `RefSeq` chromosomes. The second and the third column report the number of annoted `RefSeq` bases and the number of `RefSeq` conserved bases. The last two columns of the table, report the percentage of bases that are annoted as transposons, but that are also tagged as conserved, and, viceversa, the percentage of conserved bases that are also annoted as transposons. Each base is tagged as conserved, if it is conserved in at least 20 strains.

Chrm	Ann. Bases	Non-Cons. Bases	Ann. and Non-Cons.	Non-Cons. and Ann.
1	6,220	6,071	95.31	97.64
2	21,216	19,158	88.35	97.84
3	10,553	9,032	67.46	78.82
4	53,289	54,033	90.41	89.17
5	19,415	10,892	44.52	79.35
6	8,256	6,637	74.26	92.38
7	43,622	42,592	86.04	88.12
8	10,674	8,507	64.93	81.47
9	7,893	9,927	83.92	66.73
10	22,473	8,886	34.78	87.95
11	3,975	2,997	46.62	61.83
12	35,712	31,898	68.41	76.58
13	28,246	26,009	87.88	95.44
14	20,465	19,791	91.00	94.10
15	29,428	26,574	85.44	94.61
16	35,560	25,306	69.61	97.81

Table 4.11: Overlaps between transposons and non-conserved regions in the `RefSeq` chromosomes. The second and the third column report the number of annoted bases and the number of non-conserved bases. The last two columns of the table, report the percentage of bases that are annoted as transposons, but that are also tagged as non-conserved, and, viceversa, the percentage of non-conserved bases that are also annoted as transposons. Each base is tagged as non-conserved, if it is non-conserved in at least 20 strains.

In the rest of this section, we refer to the bases annoted as transposons, simply as *annoted* bases.

In Table 4.10 each row refers to one of the 16 chromosomes of the dataset. The second and the third column report the number of annoted `RefSeq` bases (we do not distinguish in this context between `TY` and `LTR` elements), and the number of

RefSeq conserved bases. A RefSeq base is tagged as *conserved* if it is conserved in at least 20 of the 38 pairwise comparisons between a chromosome of the RefSeq strain and the homologous chromosomes of the other strains. The last two columns of the table report the percentage of bases that are annoted as transposons, but also tagged as conserved, and, viceversa, the percentage of conserved bases that are also annoted as transposons. Table 4.11 reports the same statistics, but instead of the conserved RefSeq bases, the non-conserved RefSeq bases are considered.

Table 4.10 shows that both the percentage of annoted bases that are also tagged as conserved, and the percentage of conserved bases that are also annoted are very low, on average: 23% and 0.8% respectively. These percentages, clearly show that the overlap between annoted bases and conserved bases is small. In other words, few transposons are contained in the conserved segments of the RefSeq chromosomes.

Completely different is the situation depicted in Table 4.11. In this case, on average, the 86% of the RefSeq annoted bases are also non-conserved bases, while the percentage of the non-conserved bases that are also annoted is above 74%, on average. This fact suggests that the correlation between the non-conserved bases detected by Regender and the bases that are annoted as transposons is strong, and that our idea of detecting transposable elements by non-conserved mobile segments works.

Ann. Name	Chr	Ann. Name	Chr
YBLCdelta7	2	YHRWdelta9	8
YBRWdelta17	2	YILCdelta5	9
YBRWdelta16	2	YJLWdelta8	10
YBRCdelta14	2	YLRCtau1	12
YBLWdelta8	2	YLRWdelta20	12
YBRWdelta15	2	YMRWdelta17	13
YCRWdelta11	3	YMRCdelta11	13
YCRWdelta9	3	YMRCtau1	13
YCRCtau1	3	YMRWdelta16	13
YDRWdelta10	4	YNLWsigma2	14
YDRCdelta2	4	YORCdelta11	15
YDRWdelta12	4	YOLCdelta7	15
YGLWdelta4	7	YOLCdelta8	15
YGRWsigma7	7	YORCdelta9	15
YGRCdelta12	7	YPRWdelta16	16
YGRWdelta26	7	YPLWdelta7	16
YGRWdelta31	7	YPRWdelta14	16
YHRCtau4	8	YPRCdelta15	16

Table 4.12: List of transposons conserved in all the 39 strains of the dataset.

However, the fact that not all the annoted bases have been tagged as conserved in at least one strain poses an intriguing question. Is it possible that some of the TY/LTR elements are conserved in all the strains, not being subject to any genomic rearrangements?

Given the results of the pairwise comparisons between RefSeq chromosomes and the homologous chromosomes of the other 38 low-coverage strains of the dataset, it is easy to answer to the above question. Table 4.12 reports all the transposons that are conserved in all the strains. It can be observed that no TY is listed in the table, while the 13% of LTR elements are conserved in all the 38 strains of the dataset. This surprising phenomenon clearly shows that some of the transposable elements that have been identified on the `RefSeq` strain due to their structure, are not subject to any rearrangement event in any of the strain of the dataset. In other words, they look like transposable elements which at some point of the *S. cerevisiæ* evolution have settled on a specific position of the genome. In some sense they preserved their structure, but they lost their mobility.

What about the 26% of bases that have been tagged as non-conserved, but that are not annoted as `TY` or `LTR` in the `RefSeq` strain?

Even more interesting is the size distribution of the above non-conserved segments. About 35% of them is very small (less than 300 bases), 30% range over 300 and 600 bases, while the remaining 35% is very large (more than 6000 bases long).

Some of these segments can be part of mobile and repetitive DNA sequences that are not related with transposable elements, others could be transposons that have not been annoted in `RefSeq`(in particular the short segments can be `LTR` elements that have not been identified by traditional methods due to their unusual structure). However, only a structural analysis of these segments based on traditional techniques [25] can distinguish between true `TY` and `LTR` elements and false positive. The interested reader can download the precise coordinates of all the not-conserved segments at `www.di.unipi.it/~gbattag/regender`.

4.8 Conclusions and Future Work

In the previous sections we have discussed a concrete example of pattern discovery, showing how pattern discovery can overcome the limitations of the traditional pattern search approaches in detecting transposons. Although in our case study we analyzed a *S. cerevisiæ* dataset, our approach is not restricted to yeast genomes, since it does not rely on the structure of the transposons but on their behavior (being mobile). However, our approach is not intended to be an alternative to the traditional approaches. The identification of new classes of transposons is a multistep process and only by detecting the presence of specific repeats and specific promoters that are associated with known transposon classes inside a non-conserved segment, we can declare a DNA segment to be a transposon. Our alignment-based pattern discovery methodology is intended to help the user to filter a whole input genome that can be several Mb long, focusing on few non-conserved DNA segments that, as we discussed in Section 4.7.3, are good transposon candidates.

The analysis that we performed in the previous sections is only the first step of a more ambitious project that aim at identifying all the transposable elements in a given population genomic dataset, tracking their movements, and forecasting their future rearrangements. In the rest of this section we discuss some interesting lines of research that can be the subject of further investigations.

Improving the fragment generation phase. So far, the emphasis has been on the methodological aspect of transposons detection rather than on the efficiency of the proposed approach. We have shown the advantages of a transposon structure agnostic approach, based on alignment, over the traditional pattern search approaches.

Although the `Regender` prototype has not been designed with the efficiency in mind, as discussed in Section 4.7.1, its simplicity and its ability of exploiting the high-similarity of the compared sequences makes it competitive with other state of the art tools, and even with `Blast`.

Moreover, in Section 4.7.2 we have shown through a massive experimentation comparing `RefSeq` chromosomes with the homologous chromosomes of the other 38 low-coverage *S. cerevisiæ* strains of [130], that our choice of exact L-grams does not decrease the output quality. To our knowledge, this is the first time that such a massive experimental analysis is performed on a large population genomic dataset.

The rolling hash technique used by `Regender` is not the only available technique that can be used to detect exact L-grams, shared by a set of input sequences. Suffix tree [186, 88] can accomplish this task in time and space linear in the aggregate size of the input sequences. However, its high memory consumption that in practice can range from 9 to 11 times the size of the string to be indexed [77], makes it inapplicable to a large dataset.

Several approaches have been proposed in literature to overcome this limitation. Suffix arrays provide a more space-efficient alternative to suffix trees [133]. In [106, 2] it has been showed that every algorithm that uses a suffix tree can systematically be replaced with an algorithm, which uses an *enhanced* version of suffix arrays, having the same time complexity as the original algorithm. In practice, enhanced suffix arrays requires about 8 bytes per character [2, 98].

Since `Regender` has been designed with modularity in mind, the rolling hash fragment generation component can easily be replaced by a different implementation based on suffix array. It would be of interest to compare the efficiency of our rolling hash approach, with different approaches based on suffix trees/arrays.

Massive Datasets. In our experimental investigation we compared each chromosome of the `RefSeq` strain, with the homologous chromosome of the other 38 *S. cerevisiæ* strains of the dataset. In this way we can detect large scale *intra-chromosomic* rearrangements where a transposons of chromosome 4 has transposed into a different location of the same chromosome.

However, [128] shows that transposons are not only involved in intra-chromosomic rearrangements, but they can also be involved in *inter-chromosomic* rearrangements, transposing from one chromosome to another. To detect these inter-chromosomic rearrangements, the comparison of pairs of homologous chromosome is not enough, but the whole `RefSeq` genome must be compared with that of the other strains.

If this is feasible in the case of *S. cerevisiæ*, whose complete genome is about

12 Mbases, the analysis of different organisms can be much more challenging (the human genome, for example, is about 2.9 Gbases long).

As discussed in Section 4.7.1, we used a pairwise star comparison strategy, where a pairwise comparison is performed between the chromosome of the `RefSeq` strain, and the homologous chromosome of the other 38 strains.

However, we would like to go beyond, by *simultaneously* comparing all the 39 strain genomes. This task is challenging with the *S. cerevisiæ* dataset that we analyzed, which is about 468 Mb, but it is impossible with other datasets as that in `www.1000genomes.org`, that contains 1000 human genomes.

However, our experiments clearly show that such population genomic datasets are a clear example of *repetitive sequence collection*, because the difference between the `RefSeq` genome and the genome of the other strains can be expressed by a short lists of basic edit operations, together with few large scale rearrangement operations. Data analysis on such massive dataset is unfeasible with traditional suffix tree or suffix array approaches, due to the memory consumption.

Recent advances in full-text indexing reduce the space occupation of the suffix tree to that of the compressed sequences, while maintaining the same functionality with only a polylogarithmic slowdown [144, 85, 69]. However, the underlying compression model considers only the predictability of the next sequence symbol given the k previous ones, where k is a small integer, hence it is unable to represent longer-term repetitiveness.

In [132], the authors develop new static and dynamic full-text indexes that can capture the high repetitiveness of the input sequences, requiring a space that is proportional to the length of one sequence, plus the total number of edit operations, that represents the other sequences. We believe that this approach can be refined also to take into account large-scale rearrangement events together with single base modifications.

Multiple Alignment. As discussed in Section 4.7.1, in our experiments we used a pairwise star comparison strategy, where a pairwise comparison is performed between the chromosomes of the `RefSeq` strain, and the homologous chromosomes of the other 38 strains.

A natural extension of this method, that is also discussed in [25], is multiple alignment. If pairwise global alignment, as discussed in Section 4.4, is impractical on large sequences, multiple sequence alignment is referred in [88] as the "holy grail" of the bioinformatics. Although several efficient multiple sequence alignment tools [124, 59, 148, 107, 125] are publicly available, they are only applicable if the input sequences are a few Kb long.

In Section 4.5.2 we presented the global chaining problem. As discussed in [149], the generalization of the geometric approach to k input sequences of average size n, is feasible but the resulting algorithm, whose precise time complexity depends on the selected gap cost function, has a $\log^k(n)$ factor that makes it impractical on a dataset containing more than few sequences.

Differently from the geometric approach, the graph-based approach, referred in Section 4.5.2, works for any number of input sequences and for any kind of gap cost function. Its quadratic time complexity in the number of fragments identified in the input sequences make it the reference choice for multiple sequence global chaining [97, 54].

We followed an alternative approach, which follows the spirit of the well known progressive alignment technique that is used for multiple sequence alignment [88]. We started by performing a pairwise comparison between one chromosome of the `RefSeq` strain with its homologous in a different strain, by `Regender`. Then, a third homologous chromosome is selected. It is compared with the `RefSeq` chromosome, and the resulting alignment is "merged" to the first one. This process is iterated until all the homologous chromosome have been aligned. The output multiple alignment, together with the high resolution plots for all the 16 chromosomes are available online at `www.di.unipi.it/~gbattag/regender`.

Also in the case of these multiple alignments we plan to perform a complete experimental investigation, by comparing our approach with other state of the art tools.

Analysis of mobile elements. In Section 4.7.2 we have empirically shown the correctness of our idea: transposons can be identified by looking at the non-conserved elements of the `RefSeq` strain who are involved in large scale insertion or deletion events. More precisely, we discovered that, on average, more than 86% of the bases that are annoted as transposons are non-conserved, while more than 74% of non-conserved bases is annoted as transposable elements.

Moreover, by investigating why a relevant part of the `RefSeq` bases that are tagged as conserved by `Regender` are annoted as transposons, we discovered that there exists a non negligible number of inactive `LTR` elements, that are conserved in all the strains of the dataset (see Table 4.12 for details). A traditional pattern search approach would make this kind of analysis impossible, since the above `LTR`s are masked by unresolved bases in some strains.

However, if only the 74% of non-conserved bases is annoted as transposable elements, what about the remaining 26% of bases that have been tagged as non-conserved, but that are not annoted as `TY` or `LTR` in the `RefSeq` strain? Some of these bases can be part of mobile and repetitive DNA sequences who are not related with transposable elements. Others could be transposons that have not been annoted in `RefSeq`. However, only a structural analysis of these segments based on traditional techniques [25] can match the structure of these "candidate" transposons against the known transposon families that are reported in literature (see [34] for a survey), distinguishing between true transposable elements and false positive.

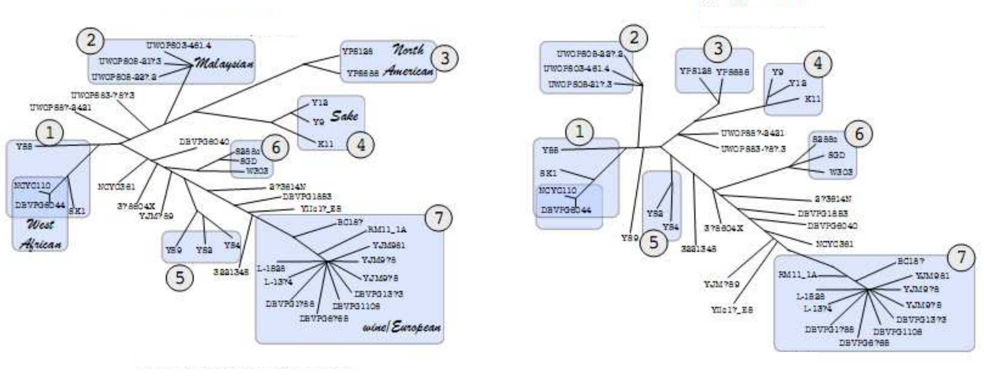


Figure 4.5: Redrawn of the phylogenetic tree constructed in [130], based on the SNPs detected in the *S. cerevisiæ* dataset (on the left). On the right, the *mobilome tree* constructed by using the information about the non-conserved segments detected in the *multiple alignment* of all the 38 strains of the same dataset by `Regender`. Groups of strains that are clustered together in both trees are highlighted in rectangles.

From Mobilome to Mobilome Tree. A *phylogenetic tree* (sometimes referred as *evolutionary tree*), is a tree showing the evolutionary relationships among various species or other entities that are believed to have a common ancestor. Phylogenetic trees may be either rooted or unrooted. In rooted trees, there is a unique node, called the root, representing a common ancestor, from which a unique path leads to any other node. An unrooted tree only specifies the relationship among species, without identifying a common ancestor, or evolutionary path (the interested reader can refer to [88] for a formal definition of phylogenetic tree). In our case, since we compare different strains of *S. cerevisiæ* the resulting phylogenetic tree has one strain in each leaf, and it reveals the relationships among these strains.

The basic problem in constructing a phylogenetic tree is the construction of a matrix, where each row represents the characteristics being used to measure the distance between two different entities being compared (two strains in our case).

In [130] the phylogenetic tree representing the population structure of the *S. cerevisiæ* strains has already been constructed by considering the *SNPs* occurring in the strains. *Single Nucleotide Polymorphism* (SNP for short) are mutations in the DNA sequence that occur when a single nucleotide (`A`,`T`,`C`, or `G`) in the genome sequence is mutated. For example, a SNP can transform the DNA sequence `AACGCTAG` into `ATCGCTAG`, substituting the second `A` with `T`.

SNPs are a widely used class of features in detecting the plylogenetic structure of a population. However, as discussed in [67, 31], transposons are important generators of variation upon which natural selection act. Hence, it would be interesting to perform a phylogenetic study, not based on single base mutations only, as in the case of SNPs, but on large scale rearrangement events.

A preliminary result of this line of investigation is shown in Figure 4.5, which qualitatively compares the phylogenetic tree constructed by SNPs in [130] (on the left), with a second tree, that we call *mobilome tree*, because it has been constructed by using the information about the non-conserved segments detected in the *multiple alignment* of the 38 strains in the *S. cerevisiæ* dataset. The set of strains "clustered together" in both trees are highlighted in rectangles and numbered. As we can see, although the two trees have been constructed by completely different methods, they show a very similar structure. This first result, empirically validates our method by showing that the similarity of different strain genomes can be detected by looking at the mobile segments (the mobilome), without any annotations about the genes that are found in the genome, and the SNP locations. However, only an extensive and well designed experimental analysis, involving other population datasets, can asses the validity of this method.

Conclusions

In this thesis we discussed three incarnations of the pattern discovery task, exploring three types of patterns that can model different regularities of the input dataset. While mask patterns have been designed to model short repeated biological sequences, showing a high conservation of their content at some specific positions, π-patterns have been designed to detect repeated patterns whose parts maintain their physical adjacency but not their ordering in all the pattern occurrences. Transposons, instead, model mobile sequences in the input dataset, which can be discovered by comparing different copies of the same input string, detecting large insertions and deletions in their alignment.

Each of the above class of patterns represents a novel solution to tackle an existing problem. In the case of mask patterns we described a new class of repetitions in a sequence, using a succinct description (mask) that gives rise to a smaller set of output patterns with respect to the set of frequent patterns with don't cares (2^L instead of $(|\Sigma| + 1)^L$ in the worst case). While in the traditional approaches, frequent don't care patterns having a different number of occurrences in the input text T were considered as different patterns, in our approach, if they share the same structure in terms of positions that are conserved across their occurrences, they are represented by the same mask pattern. In Chapter 2 we have shown how to build the set of all frequent maximal masks of length L in $\mathcal{O}(2^L n)$ time and space in the worst case, using a variant of the Karp-Miller-Rosenberg approach, and a proper visit of the mask lattice.

In Chapter 3 we explored the problem of detecting conserved gene clusters in genomes by using fixed length π-patterns. Differently than previous approaches that have been proposed in the literature, we do not sacrifice the information about the symbol multiplicities during the pattern discovery task. Moreover, we do not rely on any notion of maximality to reduce the number of output patterns, instead we rank the π-patterns according to the number of conserved common intervals detected in their occurrence lists. In the aforementioned chapter a two-phase approach has been proposed to detect the top-k frequent π-patterns occurring in the input text T. While the first phase can be efficiently accomplished (we proposed a novel $\mathcal{O}(L \log |\Sigma| \, n)$ time, and $\mathcal{O}(Ln)$ space algorithm, that improves the state of the art algorithm by a $\log(n)$ time factor), the complexity of the second ranking phase heavily depends on the presence of repeated symbols in the π-pattern p to rank.

In fact, in Section 3.4 we have shown that given a π-pattern p, together with its occurrence list $\mathcal{L}(p)$, if p is a set of symbols then the number of frontiers represented by its minimum PQ-tree can be computed in $\mathcal{O}(|\mathcal{L}(p)||p|)$ time and $\mathcal{O}(|p|)$ space. In the case of multisets of symbols, we have proved in Section 3.5.4 that even the computation of the number of the frontiers of a given PQ-tree where the labels of the leaves are not necessarily distinct is $\#\mathcal{P}$-complete, also providing a negative answer to a long standing problem that has been left as an open issue in [155].

Finally, in Chapter 4 we have shown as even a well settled field of investigation in bioinformatics, as that of transposon detection, can benefit from a change of perspective. In fact, in this chapter, to overcome the limitations of the existing approaches, we recast the transposon detection task as a pattern discovery task. While traditional pattern search approaches fail in handling real world low-coverage population genomic datasets that are rich in unresolved bases, our pattern discovery approach can exploit the availability of multiple copies of the input genomic sequences to overcome this limitation. Transposons are detected by focusing on the non-conserved regions of the coarse-grained global alignment of the input genomic sequences computed by the `Regender` tool. The efficiency of the `Regender` approach (that runs, on average, in time linear in the sum of the sizes of the two input sequences) has been empirically assessed in the case study discussed in Chapter 4, where we experimentally compared `Regender` with other 9 state of the art global alignment tools, on the dataset containing 38 strains of the *S. cerevisiæ* yeast that has been recently released in [130]. To our knowledge, this is the first time that such a massive experimental analysis is performed on a large population genomic dataset. This experimental analysis has shown that `Regender` outperforms existing tools in terms of execution time, without compromising the quality of the output alignment. Moreover, by comparing the non-conserved segments that have been identified by `Regender`, with the the transposon annotations that are available at SGD database [170], we have shown the correctness of our idea of identifying transposons by detecting the non-conserved segments in the sequence alignment.

Future work. In the previous chapters we already presented some specific open issues that can be subjects of further investigations (see Section 2.6, Section 3.6, and Section 4.8 for details).

Bibliography

[1] M. Abouelhoda, S. Kurtz, and E. Ohlebusch. The enhanced suffix array and its applications to genome analysis. *Algorithms in Bioinformatics*, pages 449–463, 2002.

[2] M.I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.

[3] A. Aggarwal and J. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[4] R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. In *ACM SIGMOD Record*, volume 22, pages 207–216. ACM, 1993.

[5] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. 20th Int. Conf. Very Large Data Bases, VLDB*, volume 1215, pages 487–499. Citeseer, 1994.

[6] A.V. Aho and M.J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[7] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, 1990.

[8] S.F. Altschul, T.L. Madden, A.A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D.J. Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic acids research*, 25(17):3389, 1997.

[9] A. Amir, A. Apostolico, G.M. Landau, and G. Satta. Efficient text fingerprinting via Parikh mapping. *J. Discrete Algorithms*, 1(5-6):409–421, 2003.

[10] H. Arimura and T. Uno. A polynomial space and polynomial delay algorithm for enumeration of maximal motifs in a sequence. In *Algorithms and Computation, 16th International Symposium (ISAAC'05)*, volume 3827 of *Lecture Notes in Computer Science*, pages 724–737, Hainan, China, 2005. Springer.

[11] S. Arora and B. Barak. *Computational Complexity A Modern Approach*. Cambridge University Press, Cambridge, 2009.

[12] V. Bafna, B. Narayanan, and R. Ravi. Nonoverlapping local alignments (weighted independent sets of axis-parallel rectangles)* 1. *Discrete Applied Mathematics*, 71(1-3):41–53, 1996.

[13] R. Bar-Yehuda, M.M. Halldórsson, J.S. Naor, H. Shachnai, and I. Shapira. Scheduling split intervals. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 732–741. Society for Industrial and Applied Mathematics, 2002.

[14] G. Battaglia, D. Cangelosi, R. Grossi, and N. Pisanti. Masking patterns in sequences: A new class of motif discovery with don't cares. *Theoretical Computer Science*, 410(43):4327–4340, 2009.

[15] G. Battaglia, R. Grossi, R. Marangoni, and N. Pisanti. Mining biological sequences with masks. In *Biological Knowledge Discovery from Databases (BIOKDD'09/DEXA'09)*, Linz, Austria, 2009.

[16] G. Battaglia, R. Grossi, and N. Scutellà. Consecutive ones property and pq-trees for multisets: Hardness of counting their orderings. *Arxiv preprint arXiv:1102.0041*, 2011.

[17] G. Battaglia, R. Grossi, and N. Scutellà. Counting the orderings for multisets in consecutive ones property and PQ-trees. In *Proceedings of the fifteenth annual International Conference on Developments in Language Theory (DLT'11)*, volume 6795 of *LNCS*, Milan, Italy, 2011. Springer.

[18] S. Batzoglou, L. Pachter, J.P. Mesirov, B. Berger, and E.S. Lander. Human and mouse gene structure: comparative analysis and application to exon prediction. *Genome Research*, 10(7):950, 2000.

[19] M.P. Béal, A. Bergeron, S. Corteel, and M. Raffinot. An algorithmic view of gene teams. *Theoretical Computer Science*, 320(2-3):395–418, 2004.

[20] G. Bejerano, M. Pheasant, I. Makunin, S. Stephen, W.J. Kent, J.S. Mattick, and D. Haussler. Ultraconserved elements in the human genome. *Science*, 304(5675):1321, 2004.

[21] C. Berge. *Hypergraphs: Combinatorics of Finite Sets*, volume 45. Elsevier, 1989.

[22] A. Bergeron, C. Chauve, F. De Montgolfier, and M. Raffinot. Computing common intervals of K permutations, with applications to modular decomposition of graphs. *Algorithms–ESA 2005*, pages 779–790, 2005.

[23] A. Bergeron, C. Chauve, and Y. Gingras. Formal Models of Gene Clusters. *Bioinformatics algorithms: techniques and applications*, page 177, 2008.

[24] A. Bergeron, S. Corteel, and M. Raffinot. The algorithmic of gene teams. *Algorithms in Bioinformatics*, pages 464–476, 2002.

[25] C.M. Bergman and H. Quesneville. Discovering and detecting transposable elements in genome sequences. *Briefings in bioinformatics*, 2007.

[26] E. Biscardi. Strumenti computazionali per la mobilomica, una nuova branca della bioinformatica. Master's thesis, University of Pisa, 2011.

[27] M. Blanchette, T. Kunisawa, and D. Sankoff. Gene order breakpoint evidence in animal mitochondrial phylogeny. *Journal of Molecular Evolution*, 49(2):193–203, 1999.

[28] G. Blin, C. Chauve, and G. Fertin. The breakpoint distance for signed sequences. In *Proc. 1st Algorithms and Computational Methods for Biochemical and Evolutionary Networks (CompBioNets)*, pages 3–16, 2004.

[29] G. Blin, C. Chauve, and G. Fertin. Genes order and phylogenetic reconstruction: Application to γ-proteobacteria. *Comparative Genomics*, pages 11–20, 2005.

[30] T. Blumenthal. Operons in eukaryotes. *Briefings in functional genomics & proteomics*, 3(3):199, 2004.

[31] A. Bohne, F. Brunet, D. Galiana-Arnoux, C. Schultheis, and J.N. Volff. Transposable elements as drivers of genomic and biological diversity in vertebrates. *Chromosome Research*, 16(1):203–215, 2008.

[32] K.S. Booth. *PQ-tree algorithms.* PhD thesis, Univ. of California, Dec., 1975.

[33] K.S. Booth and G.S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. *J. Comput. Syst. Sci.*, 13(3):335–379, 1976.

[34] N.J. Bowen and I.K. Jordan. Transposable elements and the evolution of eukaryotic complexity. *Current issues in molecular biology*, 4:65–76, 2002.

[35] N. Bray, I. Dubchak, and L. Pachter. AVID: A global alignment program. *Genome Research*, 13(1):97, 2003.

[36] B. Brejová, D. Brown, and T. Vinar. Vector seeds: an extension to spaced seeds allows substantial improvements in sensitivity and specificity. In *WABI*, volume 2812 of *LNCS*, pages 39–54, Budapest, Hungary, 2003. Springer.

[37] G.S. Brodal and R. Fagerberg. Cache oblivious distribution sweeping. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming, ICALP 2002*, volume 2380 of *Lecture Notes in Computer Science*, pages 426–438, Malaga, Spain, 2002. Springer-Verlag.

[38] M. Brudno, C.B. Do, G.M. Cooper, M.F. Kim, E. Davydov, et al. LAGAN and Multi-LAGAN: efficient tools for large-scale multiple alignment of genomic DNA. *Genome research*, 13(4):721, 2003.

[39] M. Brudno and B. Morgenstern. Fast and sensitive alignment of large genomic sequences. In *CSB*, pages 138–, 2002.

[40] S. Burkhardt and J. Kärkkäinen. Better filtering with gapped q-grams. In *CPM*, volume 2089 of *LNCS*, pages 73–85, Jerusalem, Israel, 2001. Springer.

[41] S.B. Carroll. Evolution at two levels: on genes and form. *PLoS Biology*, 3(7):1159, 2005.

[42] P. Chain, S. Kurtz, E. Ohlebusch, and T. Slezak. An applications-focused review of comparative genomics tools: Capabilities, limitations and future challenges. *Briefings in Bioinformatics*, 4(2):105, 2003.

[43] K.P. Choi, F. Zeng, and L. Zhang. Good spaced seeds for homology search. *Bioinformatics*, 20(7):1053–1059, 2004.

[44] J.D. Cohen. Recursive hashing functions for n-grams. *ACM Trans. Inf. Syst.*, 15(3):291–320, 1997.

[45] Human Genome Sequencing Consortium. Finishing the euchromatic sequence of the human genome. *Nature*, 431:931–945, 2006.

[46] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition.* The MIT Press, September 2001.

[47] M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on strings.* Cambridge Univ Pr, 2007.

[48] M. Crochemore and W. Rytter. Usefulness of the karp-miller-rosenberg algorithm in parallel computations on strings and arrays. *Theoretical computer science*, 88(1):59–82, 1991.

[49] M. Crochemore and W. Rytter. *Jewels of stringology.* World Scientific, 2002.

[50] Z.J. Czech, G. Havas, and B.S. Majewski. Perfect hashing. *Theoret. Comput. Sci.*, 182(1-2):1–43, 1997.

[51] A.C.E. Darling, B. Mau, F.R. Blattner, and N.T. Perna. Mauve: multiple alignment of conserved genomic sequence with rearrangements. *Genome research*, 14(7):1394, 2004.

[52] A.L. Delcher, S. Kasif, R.D. Fleischmann, J. Peterson, O. White, and S.L. Salzberg. Alignment of whole genomes. *Nucleic Acids Research*, 27(11):2369, 1999.

[53] A.L. Delcher, A. Phillippy, J. Carlton, and S.L. Salzberg. Fast algorithms for large-scale genome alignment and comparison. *Nucleic Acids Research*, 30(11):2478, 2002.

[54] J.S. Deogun, J. Yang, and F. Ma. Emagen: An efficient approach to multiple whole genome alignment. In *Proceedings of the second conference on Asia-Pacific bioinformatics-Volume 29*, page 122. Australian Computer Society, Inc., 2004.

[55] G. Didier. Common intervals of two sequences. *Algorithms in Bioinformatics*, pages 17–24, 2003.

[56] G. Didier, T. Schmidt, J. Stoye, and D. Tsur. Character sets of strings. *Journal of Discrete Algorithms*, 5(2):330–340, 2007.

[57] J.-E. Duchesne, M. Giraud, and N. El-Mabrouk. Seed-based exclusion method for non-coding RNA gene search. In *COCOON*, volume 4598 of *LNCS*, pages 27–39, Banff, Canada, 2007. Springer.

[58] R.M. Durbin, D.L. Altshuler, G.R. Abecasis, D.R. Bentley, A. Chakravarti, A.G. Clark, F.S. Collins, F.M. De La Vega, P. Donnelly, M. Egholm, et al. A map of human genome variation from population-scale sequencing. *Nature*, 467(7319):1061–1073, 2010.

[59] R.C. Edgar. MUSCLE: multiple sequence alignment with high accuracy and high throughput. *Nucleic acids research*, 32(5):1792, 2004.

[60] E.E. Eichler and D. Sankoff. Structural dynamics of eukaryotic chromosome evolution. *Science*, 301(5634):793, 2003.

[61] T.H. Eickbush and A.V. Furano. Fruit flies and humans respond differently to retrotransposons. *Current opinion in genetics & development*, 12(6):669–674, 2002.

[62] T. Eiter, K. Makino, and G. Gottlob. Computational aspects of monotone dualization: A brief survey. *Discrete Applied Mathematics*, 156(11):2035–2049, 2008.

[63] K. Engel. *Sperner theory.* Cambridge University Press, New York, NY, USA, 1997.

[64] D. Eppstein, Z. Galil, R. Giancarlo, and G.F. Italiano. Sparse dynamic programming I: Linear cost functions. *Journal of the ACM (JACM)*, 39(3):519–545, 1992.

[65] R. Eres, G.M. Landau, and L. Parida. A combinatorial approach to automatic discovery of cluster-patterns. In *WABI*, volume 2812 of *LNCS*, pages 139–150, Budapest, Hungary, 2003. Springer.

[66] E. Eskin. From profiles to patterns and back again: a branch and bound algorithm for finding near optimal motif profiles. In *RECOMB '04: Proceedings of the eighth annual international conference on Resaerch in computational molecular biology*, pages 115–124, New York, NY, USA, 2004. ACM.

[67] N.V. Fedoroff. Transposable elements as a molecular evolutionary force. *Annals of the New York Academy of Sciences*, 870(MOLECULAR STRATEGIES IN BIOLOGICAL EVOLUTION):251–264, 1999.

[68] S. Feng and E. Tillier. A fast and flexible approach to oligonucleotide probe design for genomes and gene families. *Bioinformatics*, 23(10):1195–1202, 2007.

[69] P. Ferragina and G. Manzini. Indexing compressed text. *Journal of the ACM (JACM)*, 52(4):552–581, 2005.

[70] D.J. Finnegan. Eukaryotic transposable elements and genome evolution. *Trends in Genetics*, 5:103–107, 1989.

[71] G. Franceschini. Proximity mergesort: Optimal in-place sorting in the cache-oblivious model. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 291–299. Society for Industrial and Applied Mathematics, 2004.

[72] M.L. Fredman and L. Khachiyan. On the complexity of dualization of monotone disjunctive normal forms. *J. Algorithms*, 21(3):618–628, 1996.

[73] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 285–297, New York, NY, USA, 1999.

[74] D.R. Fulkerson and D.A. Gross. Incidence matrices and interval graphs. *Pacific J. Math*, 15(3):835–855, 1965.

[75] M.R. Garey and D.S. Johnson. *Computers and Intractability.* W.H. Freeman and Co., New York, 1979.

[76] S.P. Ghosh. File organization: the consecutive retrieval property. *Commun. ACM*, 15(9):802–808, 1972.

[77] R. Giegerich, S. Kurtz, and J. Stoye. Efficient implementation of lazy suffix trees. *Algorithm Engineering*, pages 30–42, 1999.

[78] M. Giulia, G. Battaglia, N. Pisanti, R. Grossi, and R. Marangoni. Towards mobilome inference in yeast genomes. Poster at the seventh annual meeting of the Bioinformatics Italian Society (BITS'10), 2010.

[79] M. Giulia, G. Battaglia, N. Pisanti, R. Grossi, and R. Marangoni. Inferring mobile elements in s. cerevisiae strains. In *International Conference on Bioinformatics Models, Methods and Algorithms (BIOINFORMATICS'11)*, pages 131–136, Rome, Italy, 2011. SciTePress.

[80] P. Godfrey, R. Shipley, and J. Gryz. Algorithms and analyses for maximal vector computation. *VLDB Journal: Very Large Data Bases*, 16(1):5–28, October 2006.

[81] B. Goethals. Survey on frequent pattern mining. *Manuscript*, pages 1–43, 2003.

[82] L. Goodstadt and C.P. Ponting. Phylogenetic reconstruction of orthology, paralogy, and conserved synteny for dog and human. *PLoS Computational Biology*, 2(9):e133, 2006.

[83] O. Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162(3):705–708, 1982.

[84] T.R. Gregory. *The Evolution of the Genome*. Academic Press, December 2004.

[85] R. Grossi and J.S. Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. *SIAM Journal on Computing*, 35:378, 2005.

[86] D. Gunopulos, R. Khardon, H. Mannila, S. Saluja, H. Toivonen, and R. Sewak Sharma. Discovering all most specific sentences. *ACM Trans. Database Syst.*, 28(2):140–174, 2003.

[87] D. Gunopulos, H. Mannila, R. Khardon, and H. Toivonen. Data mining, hypergraph transversals, and machine learning (extended abstract). In *PODS '97: Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 209–216, New York, NY, USA, 1997. ACM.

[88] D. Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology.* Cambridge University Press, 1997.

[89] S. Haddadi and Z. Layouni. Consecutive block minimization is 1.5-approximable. *Inf. Process. Lett.*, 108(3):132–135, 2008.

[90] A. Halpern, D. Huson, and K. Reinert. Segment match refinement and applications. *Algorithms in Bioinformatics*, pages 126–139, 2002.

[91] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2001.

[92] X. He and M.H. Goldwasser. Identifying conserved gene clusters in the presence of homology families. *Journal of Computational Biology*, 12(6):638–656, 2005.

[93] S. Heber and J. Stoye. Finding all common intervals of k permutations. In *CPM*, volume 2089 of *Lecture Notes in Computer Science*, pages 207–218, Jerusalem, Israel, 2001. Springer.

[94] D.A. Hickey. Selfish DNA: a sexually-transmitted nuclear parasite. *Genetics*, 101(3-4):519, 1982.

[95] D.S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, 18(6):341–343, 1975.

[96] R. Hoberman and D. Durand. The incompatible desiderata of gene cluster properties. *Comparative Genomics*, pages 73–87, 2005.

[97] M. Höhl, S. Kurtz, and E. Ohlebusch. Efficient multiple genome alignment. In *ISMB*, pages 312–320, 2002.

[98] R. Homann, D. Fleer, R. Giegerich, and M. Rehmsmeier. mkESA: enhanced suffix array construction tool. *Bioinformatics*, 25(8):1084, 2009.

[99] W.L. Hsu. PC-trees vs. PQ-trees. In *COCOON*, volume 2108 of *Lecture Notes in Computer Science*, pages 207–217, Guilin, China, 2001. Springer.

[100] L. Ilie and S. Ilie. Fast computation of good multiple spaced seeds. In *WABI*, volume 4645 of *LNBI*, pages 346–358, Philadelphia, PA, USA, 2007. Springer.

[101] G. Jacobson and K.P. Vo. Heaviest increasing/common subsequence problems. In *Combinatorial Pattern Matching*, pages 52–66. Springer, 1992.

[102] D. Joseph, J. Meidanis, and P. Tiwari. Determining DNA sequence similarity using maximum independent set algorithms for interval graphs. *Algorithm TheorySWAT'92*, pages 326–337, 1992.

[103] K.J. Kalafus, A.R. Jackson, and A. Milosavljevic. Pash: Efficient genome-scale sequence anchoring by positional hashing. *Genome research*, 14(4):672, 2004.

[104] V.V. Kapitonov and J. Jurka. A universal classification of eukaryotic transposable elements implemented in Repbase. *Nature Reviews Genetics*, 9(5):411–412, 2008.

[105] R.M. Karp, R.E. Miller, and A.L. Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. In *STOC*, pages 125–136, Denver, Colorado, USA, 1972. ACM.

[106] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Combinatorial Pattern Matching*, pages 181–192. Springer, 2001.

[107] K. Katoh, K. Kuma, H. Toh, and T. Miyata. MAFFT version 5: improvement in accuracy of multiple sequence alignment. *Nucleic acids research*, 33(2):511, 2005.

[108] D.J. Kavvadias and E.C. Stavropoulos. An efficient algorithm for the transversal hypergraph generation. *J. Graph Algorithms and Applications*, 9(2):239–264, 2005.

[109] H.H. Kazazian et al. Mobile elements and disease. *Current opinion in genetics & development*, 8(3):343–350, 1998.

[110] U. Keich, M. Li, B. Ma, and J. Tromp. On spaced seeds for similarity search. *Discr. Appl. Math.*, 138(3):253–263, 2004.

[111] W.J. Kent. BLAT the BLAST-like alignment tool. *Genome research*, 12(4):656, 2002.

[112] W.J. Kent and A.M. Zahler. Conservation, regulation, synteny, and introns in a large-scale C. briggsae–C. elegans genomic alignment. *Genome research*, 10(8):1115, 2000.

[113] L. Khachiyan, E. Boros, K. Elbassioni, and V. Gurvich. An efficient implementation of a quasi-polynomial algorithm for generating hypergraph transversals and its application in joint generation. *Discr. Appl. Math.*, 154(16):2350–2372, 2006.

[114] J.M. Kim, S. Vanguri, J.D. Boeke, A. Gabriel, and D.F. Voytas. Transposable elements and genome organization: a comprehensive survey of retrotransposons revealed by the complete Saccharomyces cerevisiae genome sequence. *Genome research*, 8(5):464, 1998.

[115] D.E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison Wesley, 2nd edition, 1997.

[116] R. Kolpakov and M. Raffinot. New algorithms for text fingerprinting. In *Combinatorial Pattern Matching*, pages 342–353. Springer, 2006.

[117] L.T. Kou. Polynomial complete consecutive information retrieval problems. *SIAM J. Comput.*, 6(1):67–75, 1977.

[118] G. Kucherov, L. Noé, and M. A. Roytberg. Subset seed automaton. In *CIAA*, volume 4783 of *LNCS*, pages 180–191, Praque, Czech Republic, 2007. Springer.

[119] G. Kucherov, L. Noé, and M.A. Roytberg. Multiseed lossless filtration. *IEEE/ACM Trans. Comput. Biology Bioinform.*, 2(1):51–61, 2005.

[120] G. Kucherov, L. Noé, and M.A. Roytberg. A unifying framework for seed sensitivity and its application to subset seeds. *J. Bioinform. and Comput. Biology*, 4(2):553–570, 2006.

[121] C. Kuratowski. Sur les problèmes des courbes gauches en Topologie. *Fund. Math.*, 15:271–283, 1930.

[122] G.M. Landau, L. Parida, and O. Weimann. Gene proximity analysis across whole genomes via PQ trees. *Journal of Computational Biology*, 12(10):1289–1306, 2005.

[123] E.S. Lander, L.M. Linton, B. Birren, C. Nusbaum, M.C. Zody, J. Baldwin, K. Devon, K. Dewar, M. Doyle, W. FitzHugh, et al. Initial sequencing and analysis of the human genome. *Nature*, 409(6822):860–921, 2001.

[124] M. A. Larkin, G. Blackshields, NP Brown, R. Chenna, PA McGettigan, H. McWilliam, F. Valentin, IM Wallace, A. Wilm, R. Lopez, et al. Clustal W and Clustal X version 2.0. *Bioinformatics*, 23(21):2947, 2007.

[125] T. Lassmann and E.L.L. Sonnhammer. Kalign – an accurate and fast multiple sequence alignment algorithm. *BMC bioinformatics*, 6(1):298, 2005.

[126] E.L. Lawler. *Combinatorial optimization: networks and matroids.* Dover Pubns, 2001.

[127] M.Y. Leung, B.E. Blaisdell, C. Burge, and S. Karlin. An efficient algorithm for identifying matches with errors in multiple long molecular sequences. *Journal of Molecular Biology*, 221(4):1367–1378, 1991.

[128] B. Lewin. *Genes (IX ed.).* Jones and Bartlett, 2007.

[129] M. Li, B. Ma, D. Kisman, and J. Tromp. Patternhunter ii: Highly sensitive and fast homology search. *J. Bioinformatics and Computational Biology*, 2(3):417–440, 2004.

[130] G. Liti, D.M. Carter, A.M. Moses, J. Warringer, L. Parts, S.A. James, R.P. Davey, I.N. Roberts, A. Burt, V. Koufopanou, et al. Population genomics of domestic and wild yeasts. *Nature*, 458(7236):337–341, 2009.

[131] B. Ma, J. Tromp, and M. Li. PatternHunter: faster and more sensitive homology search. *Bioinformatics*, 18(3):440, 2002.

[132] V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki. Storage and Retrieval of Highly Repetitive Sequence Collections. *Journal of Computational Biology*, 17(3):281–308, 2010.

[133] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 319–327. Society for Industrial and Applied Mathematics, 1990.

[134] E.M. Marcotte, M. Pellegrini, H.L. Ng, D.W. Rice, T.O. Yeates, and D. Eisenberg. Detecting protein function and protein-protein interactions from genome sequences. *Science*, 285(5428):751, 1999.

[135] E.R. Mardis. The impact of next-generation sequencing technology on genetics. *Trends in Genetics*, 24(3):133–141, 2008.

[136] B. McClintock. The origin and behavior of mutable loci in maize. *Proceedings of the National Academy of Sciences of the United States of America*, 36(6):344, 1950.

[137] B. McClintock. Controlling elements and the gene. In *Cold Spring Harbor symposia on quantitative biology*, volume 21, page 197. Cold Spring Harbor Laboratory Press, 1956.

[138] J. Meidanis, O. Porto, and G.P. Telles. On the consecutive ones property. *Discrete Applied Mathematics*, 88(1-3):325–354, 1998.

[139] M. Michael, F. Nicolas, and E. Ukkonen. On the complexity of finding gapped motifs. *CoRR*, abs/0802.0314, 2008.

[140] P.A. Mieczkowski, F.J. Lemoine, and T.D. Petes. Recombination between retrotransposons as a source of chromosome rearrangements in the yeast Saccharomyces cerevisiae. *DNA repair*, 5(9-10):1010–1020, 2006.

[141] B. Morgenstern. DIALIGN 2: improvement of the segment-to-segment approach to multiple sequence alignment. *Bioinformatics*, 15(3):211, 1999.

[142] B. Morgenstern. A space-efficient algorithm for aligning large genomic sequences. *Bioinformatics*, 16(10):948, 2000.

[143] E.W. Myers and W. Miller. Optimal alignments in linear space. *Computer Applications in the Biosciences*, 4(1):11–17, 1988.

[144] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys (CSUR)*, 39(1):2, 2007.

[145] G. Navarro and M. Raffinot. *Flexible pattern matching in strings*. Cambridge University Press Cambridge;, 2007.

[146] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino-acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443453, 1970.

[147] Z. Ning, A.J. Cox, and J.C. Mullikin. SSAHA: a fast search method for large DNA databases. *Genome research*, 11(10):1725, 2001.

[148] C. Notredame, D.G. Higgins, and J. Heringa. T-coffee: a novel method for fast and accurate multiple sequence alignment1. *Journal of molecular biology*, 302(1):205–217, 2000.

[149] E. Ohlebusch and M.I. Abouelhoda. Chaining algorithms and applications in comparative genomics. *Handbook of Computational Molecular Biology*, 2006.

[150] S. Ohno. Evolution by gene duplication (1970) New York.

[151] K.R. Oliver and W.K. Greene. Transposable elements: powerful facilitators of evolution. *Bioessays*, 31(7):703–714, 2009.

[152] L.E. Orgel and FHC Crick. Selfish DNA: the ultimate parasite. *Nature*, 284(5757):604–607, 1980.

[153] C.M. Papadimitriou. *Computational complexity*. Addison-Wesley, Reading, Massachusetts, 1994.

[154] L. Parida. *Pattern Discovery in Bioinformatics / Theory & Algorithms*. Taylor & Francis; Chapman & Hall, September 2007.

[155] L. Parida. Statistical significance of large gene clusters. *Journal of Computational Biology*, 14(9):1145–1159, 2007.

[156] L. Parida, I. Rigoutsos, A. Floratos, D. Platt, and Y. Gao. Pattern discovery on character sets and real-valued data: linear bound on irredundant motifs and an efficient polynomial time algorithm. In *SODA '00: Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, pages 297–308, Philadelphia, PA, USA, 2000. Society for Industrial and Applied Mathematics.

[157] S. Pasek, A. Bergeron, J.L. Risler, A. Louis, E. Ollivier, and M. Raffinot. Identification of genomic features using microsyntenies of domains: domain teams. *Genome research*, 15(6):867, 2005.

[158] A.E. Peaston, A.V. Evsikov, J.H. Graber, W.N. de Vries, A.E. Holbrook, D. Solter, and B.B. Knowles. Retrotransposons regulate host genes in mouse oocytes and preimplantation embryos. *Developmental Cell*, 7(4):597–606, 2004.

[159] N. Pisanti, M. Crochemore, R. Grossi, and M.-F. Sagot. Bases of motifs for generating repeated patterns with wild cards. *IEEE/ACM Trans. Comput. Biology Bioinform.*, 2(1):40–50, 2005.

[160] K. Popendorf, H. Tsuyoshi, Y. Osana, Y. Sakakibara, and D.P. Martin. Murasaki: A Fast, Parallelizable Algorithm to Find Anchors from Multiple Genomes. *PLoS ONE*, 5(9):195–197, 2010.

[161] F.P. Preparata and M.I. Shamos. *Computational geometry: an introduction*. Springer, 1985.

[162] M.N. Price, A.P. Arkin, and E.J. Alm. The life-cycle of operons. *PLoS Genet*, 2(6):e96, 2006.

[163] P. Rice, I. Longden, and A. Bleasby. EMBOSS: The European Molecular Biology Open Software Suite. *Trends in Genetics*, 16(6):276–277, June 2000.

[164] D. Sankoff. Genome rearrangement with gene families. *Bioinformatics*, 15(11):909, 1999.

[165] D. Sankoff. Rearrangements and chromosomal evolution. *Current opinion in genetics & development*, 13(6):583–587, 2003.

[166] T. Schmidt and J. Stoye. Quadratic time algorithms for finding common intervals in two and more sequences. In *Combinatorial Pattern Matching*, pages 347–358. Springer, 2004.

[167] S. Schwartz, L. Elnitski, M. Li, M. Weirauch, C. Riemer, A. Smit, et al. MultiPipMaker and supporting tools: Alignments and analysis of multiple genomic DNA sequences. *Nucleic Acids Research*, 31(13):3518, 2003.

[168] S. Schwartz, W.J. Kent, A. Smit, Z. Zhang, R. Baertsch, R.C. Hardison, D. Haussler, and W. Miller. Human–mouse alignments with BLASTZ. *Genome Research*, 13(1):103, 2003.

[169] S. Schwartz, Z. Zhang, K.A. Frazer, A. Smit, C. Riemer, J. Bouck, R. Gibbs, R. Hardison, and W. Miller. PipMaker a web server for aligning two genomic DNA sequences. *Genome Research*, 10(4):577, 2000.

[170] SGD. SGD project. Saccharomyces Genome Database, 2010. `http://www.yeastgenome.org`.

[171] X. She, G. Liu, M. Ventura, S. Zhao, D. Misceo, R. Roberto, M.F. Cardone, M. Rocchi, E.D. Green, N. Archidiacano, et al. A preliminary comparative analysis of primate segmental duplications shows elevated substitution rates and a great-ape expansion of intrachromosomal duplications. *Genome research*, 16(5):576, 2006.

[172] J. Shendure and H. Ji. Next-generation DNA sequencing. *Nature biotechnology*, 26(10):1135–1145, 2008.

[173] A.C.C. Shih and W.H. Li. GS-Aligner: a novel tool for aligning genomic sequences using bit-level operations. *Molecular biology and evolution*, 20(8):1299, 2003.

[174] W.K. Shih and W.L. Hsu. A new planarity test. *Theor. Comput. Sci.*, 223(1-2):179–191, 1999.

[175] R.G. Snell, J.C. MacMillan, J.P. Cheadle, I. Fenton, L.P. Lazarou, P. Davies, M.E. MacDonald, J.F. Gusella, P.S. Harper, and D.J. Shaw. Relationship between trinucleotide repeat expansion and phenotypic variation in huntington's disease. *Nature genetics*, 4(4):393–397, 1993.

[176] Y. Sun and J. Buhler. Designing multiple simultaneous seeds for dna similarity search. *J. Comput. Biology*, 12(6):847–861, 2005.

[177] Y. Sun and J. Buhler. Designing patterns for profile hmm search. *Bioinformatics*, 23(2):42–43, 2007.

[178] R.J. Taft and J.S. Mattick. Increasing biological complexity is positively correlated with the relative genome-wide expansion of non-protein-coding DNA sequences. *Genome Biology*, 4:P1, 2003.

[179] J. Tamames et al. Evolution of gene order conservation in prokaryotes. *Genome Biology*, 2(6):1–0020, 2001.

[180] E. Ukkonen. Structural analysis of gapped motifs of a string. In *MFCS*, volume 4708 of *LNCS*, pages 681–690, Ceský Krumlov, Czech Republic, 2007. Springer.

[181] T. Uno and M. Yagiura. Fast algorithms to enumerate all common intervals of two permutations. *Algorithmica*, 26(2):290–309, 2000.

[182] L.G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8(2):189–201, April 1979.

[183] J.S. Varre, J.P. Delahaye, and E. Rivals. Transformation distances: a family of dissimilarity measures based on movements of segments. *Bioinformatics*, 15(3):194, 1999.

[184] J.S. Vitter. Algorithms and data structures for external memory. *Foundations and Trends in Theoretical Computer Science*, 2(4):305–474, 2008.

[185] J. Vuillemin. A data structure for manipulating priority queues. *CACM*, 21(4):309–315, 1978.

[186] P. Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th Annual Symposium on Switching and Automata Theory (swat 1973)-Volume 00*, pages 1–11. IEEE Computer Society, 1973.

[187] T. Wicker, F. Sabot, A. Hua-Van, J.L. Bennetzen, P. Capy, B. Chalhoub, A. Flavell, P. Leroy, M. Morgante, O. Panaud, et al. A unified classification system for eukaryotic transposable elements. *Nature Reviews Genetics*, 8(12):973–982, 2007.

[188] W.J. Wilbur and D.J. Lipman. Rapid similarity searches of nucleic acid and protein data banks. *Proceedings of the National Academy of Sciences of the United States of America*, 80(3):726, 1983.

[189] D.B. Wilson and D.S. Hogness. The enzymes of the galactose operon in Escherichia coli. *Journal of Biological Chemistry*, 239(8):2469, 1964.

[190] C. Yanofsky, T. Platt, IP Crawford, BP Nichols, GE Christie, H. Horowitz, M. VanCleemput, and AM Wu. The complete nucleotide sequence of the tryptophan operon of Escherichia coli. *Nucleic Acids Research*, 9(24):6647, 1981.