University of Verona

DEPARTMENT OF INFORMATICS
Master degree in Bioinformatics and Medical Biotechnology

Master's Thesis

On a new distance measure for genomic repeat discovery

Candidate:
Alessio Milanese

Thesis advisor:
Dr. Zsuzsanna Lipták

Thesis submitted in February 2015

Abstract

In this thesis we study a new approach to compare and investigate two genomic sequences. We study the *threshold* q-gram distance which measures the similarity between two sequences using the concept of q-grams, and which captures the hapax (uniquely occurring substring) and repeat content in the sequences.

A q-gram is a substring of length q. The q-gram profile of a string is an array that contains the multiplicity of each q-gram. The q-gram distance is the L_1-distance of the q-gram profiles of two strings (i.e. the sum of the absolute differences of the components), introduced by Ukkonen in 1992. Another important notion we use are *de Bruijn graphs*: the de Bruijn graph of a string is a directed multigraph whose nodes are the $(q-1)$-grams of the string and whose edges are its q-grams. De Bruijn graphs are useful to represent the q-gram profile, and hence to study the q-gram distance.

Based on the q-gram profile and q-gram distance, we introduce the *threshold* q-gram profile of a string and the *threshold* q-gram distance between two strings. These distinguish between the q-grams that appear only once in the string, which we call *hapax*, and those that appear at least twice, which we call *repeats*. We introduce a modified version of de Bruijn graphs that we call *protographs*. A protograph behaves for the *threshold* q-gram distance as does the de Bruijn graph for the q-gram distance.

We study three main questions: First, under which circumstances does the protograph, or, equivalently, the *threshold* q-gram profile, uniquely identify the string's q-gram profile? Second, which are the *string transformations* that do not change the *threshold* q-gram profile? Third, we study the relationship between the protograph of a string and its de Bruijn graph. We present partial results towards solving these questions.

As a proof of concept, we also implemented an algorithm to compute the *threshold* q-gram profile and distance. We describe the algorithms and data structures we used, analyse time and space complexity, as well as discussing some alternatives. We discuss questions of efficient implementation and the advantages and drawbacks of each solution.

As an initial study we applied the *threshold* q-gram distance to some real genomes. We first describe how to deal with sequencing errors in the genomes. Afterwards we analyse the results of the *threshold* q-gram distance applied to two strains of *E.coli*. We also analyzed the effects of randomly shuffling sequences in terms of this distance measure. Moreover, we study how the repeat and hapax contents change with the value of q. We ran experiments on four bacterial genomes with different degrees of relatedness to each other and study how the *threshold* q-gram distance changes.

Contents

Chapter 1

Introduction

In this thesis we study the *threshold* q-gram distance, a string distance measure recently introduced by G. Franco [Fra13], and its application to genomes.

We have two biological motivations. The first one is related to the importance of understanding the similarity between biological sequences. During the 80s Russel F. Doolittle was collecting all biological sequences that he could find. At that time there were not databases for biological sequences. He stored them on a computer and he compared these sequences to each other in order to find an evolutionary or biological relationship. In 1983, when a new sequence of a cancer-causing gene was reported, Doolittle compared it to all sequences in his database and he found that it was surprisingly similar to a sequence of a normal gene involved in growth and development [D$^+$83]. This led to the idea that cancer may be caused by normal genes that are activated at the wrong time. Since then the search for similar sequences in databases became increasingly more important. The reference measure in bioinformatics is the edit distance [Lev66], its time and space complexity is quadratic in the length of the input strings. If we have to evaluate the similarity between two genomes or if we have to search a protein sequence in a database, then the edit distance is too time-consuming.

Our second motivation, is related to a property of the genomes. In the genomes there are many repeated regions: in *Homo sapiens* more than half of the genome is repetitive or repeat-derived [dKGC$^+$11]. Let s be a string, we refer to a substring of s that appears only once in s as *hapax*, and a substring that appears at least twice as *repeat*. If a substring is repeated more then a fixed threshold times in the two genomes which we are comparing, then this should not change the distance. As an example, during the replication of the DNA a replication slippage can happen, otherwise known as slipped-strand mispairing: a form of mutation that leads to either a trinucleotide or dinucleotide expansion or contraction during DNA replication. A slippage event normally occurs when a sequence of repetitive nucleotides

(tandem repeats) are found at the site of the replication. When calculating the similarity between two sequences, we want to disregard this type of error introduced in the replication process.

To solve the first problem, several other string distances that can be computed in linear time have been introduced. Often, these solutions rely on properties of substrings of the two strings.

The q-gram distance between two strings, which can be computed in linear time, was developed by E. Ukkonen in 1992 [Ukk92]. This distance evaluates the difference in the number of occurrences of substrings of a fixed length (the q-grams) in the two strings. Ukkonen introduced also the q-gram profiles: the q-gram profile of a string is an array that contains the multiplicity of each q-gram. The q-gram distance is the L_1-distance of the q-gram profiles of two strings (i.e. the sum of the absolute differences of the components).

The *threshold* q-gram distance is based on the q-gram distance and on the concept of repeat that was recently studied in several papers [FM13, CFM12, CFM14] about the hapax and repeat content in genomes.

The *threshold* q-gram distance can be computed in linear time, this addresses the first problem. The second problem is solved by the properties of the *threshold* q-gram distance: given two strings s and t, this distance is zero when s and t have the same set of hapax and the same set of repeats. Based on the q-gram profile, we define the *threshold* q-gram profile of a string. This distinguishes between the q-grams that are hapax, and those that are repeats.

Pevzner [Pev95] used a graph representation to study the q-gram distance. The de Bruijn graph of a string is a directed multigraph whose nodes are the $(q-1)$-grams of the string and whose edges are its q-grams. De Bruijn graphs are useful to represent the q-gram profile, and hence to study the q-gram distance.

We introduce a modified version of de Bruijn graphs that we call *protographs*. A protograph represents the *threshold* q-gram profile the same way as the de Bruijn graph represents the q-gram profile.

We study three main questions: First, under which circumstances does the protograph, or, equivalently, the *threshold* q-gram profile, uniquely identify the string's q-gram profile? Second, which are the *string transformations* that do not change the *threshold* q-gram profile? Third, we study the relationship between the protograph of a string and its de Bruijn graph. We present partial results towards solving these questions.

As a proof of concept, we also implemented an algorithm to compute the *threshold* q-gram profile and distance. As an initial study we applied the *threshold* q-gram distance to some real genomes.

1.1 Organization of the thesis

The chapters are organized as follows:

Chapter 2

In this chapter we will explain the basic definitions that we need and the background informations useful for the thesis. Hence we will define the basic definitions on strings and graphs. We will describe three distances between strings and the balanced de Bruijn subgraphs.

Chapter 3

This chapter contains the main theoretical part of the thesis. We will first describe the *threshold* q-gram distance and introduce the protographs. Using the protograph, in Section 3.3 and 3.4 we study how we can transform one string into another without changing the *threshold* q-gram distance. We describe three transformations that are present only with respect to the *threshold* q-gram distance.

Chapter 4

We describe the algorithms and data structures we used, analyse time and space complexity, as well as discussing some alternatives. We discuss questions of efficient implementation and the advantages and drawbacks of each solution.

Chapter 5

We first describe how to deal with sequencing errors in the genomes. Afterwards we analyse the results of the *threshold* q-gram distance applied to two strains of *E.coli*. We also analyzed the effects of randomly shuffling sequences in terms of this distance measure. Moreover, we study how the repeat and hapax contents change with the value of q. We ran experiments on four bacterial genomes with different degrees of relatedness to each other and study how the *threshold* q-gram distance changes.

Chapter 6

Conclusions of the thesis. We discuss the open problems and we give an overview of the future research possibilities.

Chapter 2

String distances and Graph representation

Contents

In this chapter we will give a brief introduction to the string notation used in this thesis and then we will analyse three different string distances: *Hamming distance*, the simplest distance between two strings; *edit distance*, the reference distance measure in bioinformatics; and the *q-gram distance*, which provides a lower bound for the edit distance but can be computed more efficiently.

Afterwards we will give an overview of the basic definitions and concepts for dealing with graphs. Finally, in Section 2.2.1 we will describe balanced de Bruijn subgraphs.

2.1 String definitions

An *alphabet* Σ is a finite non-empty set, whose elements are called characters, symbols, or letters. We denote with σ the size of the alphabet. A *string* over Σ is a finite sequence $s = s_1 s_2 ... s_n$ s.t. for all i, $s_i \in s$. We denote by $|s|$ its length n. The unique string of length 0 is called the empty string and denoted by λ.

Let s and t be strings over an alphabet Σ, where $|s| = n$, $|t| = m$. Then the *concatenation* $u = st$ is the string $u = u_1 ... u_{n+m}$ s.t.

$$u_i = \begin{cases} s_i, & \text{if } i \leq n \\ t_{i-n}, & \text{if } i > n \end{cases}$$

Let s and t be two strings over an alphabet Σ:

- s is a *substring* of t, if there exist two strings u and v such that $t = usv$;
- s is a *prefix* of t, if there exists a string v such that $t = sv$;
- s is a *suffix* of t, if there exists a string v such that $t = vs$;

Let $s = s_1 s_2 s_3 ... s_n$ be a string of length n, let $i, j \in \{1, ..., n\}$, we denote the substring $s_i s_{i+1} ... s_j$ by $s[i, j]$. By convention if $i > j$, then $s[i, j] = \lambda$. Furthermore we denote the i-th symbol by $s[i]$ or s_i.

We denote by $\mathbb{N}$ the set of natural numbers composed of positive integers and zero.

$$\mathbb{N} = \{0, 1, 2, ...\}$$

To denote the set of natural numbers without the zero, we use $\mathbb{N}^*$.

$$\mathbb{N}^* = \{1, 2, 3, ...\}$$

Let Σ be an alphabet, we denote with Σ^* the set of all strings, of any length, over Σ. We denote with Σ^q all strings of length q over Σ, for $q \in \mathbb{N}^*$.

We will now introduce a new distance function on the Euclidean space.

Definition 1. Let $X = (x_1, x_2, ..., x_n)$, $Y = (y_1, y_2, ..., y_n) \in \mathbb{R}^n$ be two points, choose $p > 1$, then the $p-$*norm distance* between x and y is defined as:

$$L_p(X, Y) = \left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{1/p}$$

L_1 and L_2 are also referred to *Manhattan distance* and *Euclidean distance*, respectively.

2.1.1 Hamming distance

The Hamming distance simply counts the number of positions where two strings differ from each other.

Definition 2. Let $s = s_1 s_2 ... s_n$ and $t = t_1 t_2 ... t_n$ be two strings of the same length n. We define the *Hamming distance* $d_H(s, t)$ of s and t as the number of positions $1 \leq i \leq n$ where $s_i \neq t_i$.

Example 1. Example of the Hamming distance between two strings:

```
s = AAACBDDBCABBCDABCA
t = AABBBDDBCABBDDBBCA
      XX         X X
```

$$d_H(s, t) = 4$$

2.1.2 Edit distance

One of the most basic questions about a gene or protein is related to the function it performs, in fact half of the proteins in eukaryotic genomes have unknown function [AAJdCLV09]. One way to solve this problem is to find some other genes or proteins that are similar to the one with unknown function. Similarity of two genes at the sequence level suggests that they are homologous and hence that they may have similar functions.

When we are working with biological data, which are usually obtained from experiments, we also have to deal with errors introduced by the sequencing process. The Hamming distance is the simplest measure of distance between strings. But it has two drawbacks: *1)* it can be calculated only on strings of the same size and *2)* it accounts only for one type of error, that is a substitution of a character in the two strings. Even if the two sequences do not contain errors, when we study two biological sequences that were affected by the evolution there are many types of modifications.

The *edit distance* is a way of quantifying how two strings s and t are different from each other by counting the minimum number of operations required to transform s in t, or vice versa.

There are several definitions of edit distance, depending on the edit operations we take into account. The most used variant was introduced in 1965 by Vladimir Levenshtein [Lev66] and uses three basic edit operations: *substitutions*, *insertions* and *deletions*.

To compute the edit distance between two strings we use dynamic programming. The algorithm for computing the edit distance is similar to the NeedlemanWunsch algorithm [NW70] for computing optimal alignments.

The main idea is to compute the distance for each pairs of prefixes of the given strings. Considering the empty string, we obtain $m+1$ prefixes for s and $n+1$ prefixes for t. We construct an $((m+1)\times(n+1))$-matrix M that contains the edit distance for each pair of prefixes. In this way, in position $M(i,j)$ we have the edit distance between $s[1,i]$ and $t[1,j]$. In particular, in position $M[m,n]$ we have the edit distance of s and t. We compute the edit distance between prefixes using the information of the edit distance of shorter prefixes.

The algorithm to fill in the matrix M and calculate the edit distance between s and t requires at least $m \cdot n$ steps: one for each element of the matrix. Therefore the time complexity of the algorithm to calculate the edit distance is $O(m \cdot n)$. The space complexity is $O(m \cdot n)$ to store the matrix and the strings.

2.1.3 q-gram distance

The edit distance can model the evolutionary events that act on a nucleotide sequence, but it has two drawbacks: *1)* it requires quadratic time to be computed, which it is not feasible when we are working with genomes of length of several millions, and *2)* it emphasizes the correct order of the characters, i.e it does not take into account the possible movements of blocks of text. For example if we exchange two paragraphs of this thesis, then the edit distance will be high; but from a certain point of view, the thesis has not changed so much. These types of events have occurred several times during the evolution of genomes, and are called *translocations*.

The q-gram distance was introduced by Ukkonen [Ukk92] in 1992, and it is a possible solution to the two problems above. This section closely follows [Ukk92], [SGK$^+$12] and [Pev95].

First of all, what is a q-gram? A *q-gram* is any string $t = a_1a_2...a_q$ in Σ^q. The notion of q-gram dates back to Shannon [Sha48]. In the literature the terms *q-mer* and *q-word* are also used.

Let s be a string of length n and let $q \in \mathbb{N}^*$ be a number s.t. $q < n$. The string s contains $n - q + 1$ q-grams (counted with multiplicity). For example, let $s = ATCGAAGT$ and $q = 5$, the 5-grams of s are $ATCGA, TCGAA, CGAAG, GAAGT$.

Let $s = s_1s_2...s_n$ be a string over Σ, let $\sigma = |\Sigma|$, and choose $q \in [1, n]$. The *occurrence count* of a q-gram $x \in \Sigma^q$ in s is the number:

$$N(s, x) = |\{i \in [1, n - q + 1] \mid s[i, i + q - 1] = x\}|$$

The *q-gram profile* of s is a vector P with σ^q elements defined as $P_q(s) = (N(s, x))_{x \in \Sigma^q}$.

To calculate the q-gram distance we evaluate the difference between the profiles of the two strings.

Definition 3. Let $s \in \Sigma^n$ and $t \in \Sigma^m$ be strings over an alphabet Σ, and let q be a number s.t. $1 \leq q \leq min\{n, m\}$. The *$q$-gram distance* between s and t is

$$d_q(s, t) = \sum_{x \in \Sigma^q} |N(s, x) - N(t, x)|$$

As is easy to see, the q-gram distance is the Manhattan distance, or L_1 distance, between the profiles of the two strings.

Example 2. Let $\Sigma = \{0, 1\}$ be an alphabet, let $s = 0110$, $t = 11100$, and $q = 2$. Using lexicographic order $(00, 01, 10, 11)$ among the q-grams, the two profiles are:

$$P_2(s) = (0, 1, 1, 1),$$

$$P_2(t) = (1, 0, 1, 2),$$

and the 2-gram distance is equal to

$$d_2(s, t) = \sum_{x \in \Sigma^q} |N(s, x) - N(t, x)| = 1 + 1 + 0 + 1 = 3.$$

Example 3. Let $\Sigma = \{0, 1\}$ be an alphabet, let $s = 011000$, $t = 000110$, and $q = 2$. Using lexicographic order, the profiles are $P_2(s) = P_2(t) = (2, 1, 1, 1)$, so $d_2(s, t) = 0$.

Note that in Example 3 the q-gram distance is zero though $s \neq t$.

The q-gram distance is a pseudometric, in fact for all $s, t, z \in \Sigma^*$:

1. $d_q(s, t) \geq 0$ (non-negativity),
2. $d_q(s, t) = d_q(t, s)$ (symmetry),
3. $d_q(s, t) \leq d_q(s, z) + d_q(z, t)$ (triangle inequality).

It is not a metric since it does not fulfil the identity of indiscernibles: $d_q(s, t)$ can be 0 even if $s \neq t$. This is the case if s and t have the same q-gram profile.

Let s be a string of length n and $q \leq n$. Let $Z_q(s) = \{t \in \Sigma^* \mid d_q(s, t) = 0\}$ be the set of strings with the same q-gram profile as s. As can be seen easily, all strings in $Z_q(s)$ are of length $|s|$.

A new string in $Z_q(s)$ can be found applying one of the following transformations introduced in [Ukk92]:

1. (transposition) Let z_1 and z_2 be $(q-1)$-grams and y_i be strings, for $i \in \{1, ..., 5\}$. If s can be written as $s = y_1 z_1 y_2 z_2 y_3 z_1 y_4 z_2 y_5$, then the string $t = y_1 z_1 y_4 z_2 y_3 z_1 y_2 z_2 y_5$, where y_2 and y_4 are exchange, is also in $Z_q(s)$. If z_1 and z_2 coincides we have a similar transformation. Let z be a $(q-1)$-gram, if s can be written as $s = y_1 z y_2 z y_3 z y_4$, then the string $t = y_1 z y_3 z y_2 z y_4$ is in $Z_q(s)$.

2. (rotation) Let z_1 and z_2 be $(q-1)$-grams and y_1 and y_2 be strings. If s can be written as $s = z_1 y_1 z_2 y_2 z_1$, then also $t = z_2 y_2 z_1 y_1 z_2$ is in $Z_q(s)$.

We refer to these as *Ukkonen transformations*.

Ukkonen conjectured [Ukk92] and Pevzner proved [Pev95] that repeatedly applying those two rules it is possible to generate the whole set $Z_q(s)$.

Example 4. Let $\Sigma = \{a, b, c\}$ be an alphabet, let $s = baaaacbbbccaaabcbbbaa$ and let $q = 4$. From s we can obtain s' applying a transposition:

	y_1	z_1	y_2	z_2	y_3	z_1	y_4	z_2	y_5
s =	b	aaa	ac	bbb	cc	aaa	bc	bbb	aa
s' =	b	aaa	bc	bbb	cc	aaa	ac	bbb	aa
	y_1	z_1	y_4	z_2	y_3	z_1	y_2	z_2	y_5

From s we can obtain s'' applying a rotation:

	z_1	y_1	z_2	y_2	z_1
s =	baa	aacbbb	cca	aabcbb	baa
s'' =	caa	aabcbb	baa	aacbbb	caa
	z_2	y_2	z_1	y_1	z_2

Note that $s', s'' \in Z_q(s)$.

There is only a weak connection between the edit distance and the q-gram distance.

Lemma 1 (q-gram lemma, [Ukk92, Thm 5.1])**.** *Let s and t be strings, let d_E denote the edit distance. Then*

$$d_q(s, t)/(2q) \leq d_E(s, t)$$

Proof.

An operation of the edit distance can destroy up to q q-grams. For example, let $s = aaaaaaaa$ be a string and let $q = 3$. In s we have six times the q-gram aaa. If we substitute the fourth a for a t we get $s' = aaataaaa$: the q-gram aaa compares only three times. As we can see, a substitution can affect 3 3-gram; we can prove the same for insertion and deletion.

Each changed q-gram can cause a change of up to 2 in the q-gram distance, as the occurrence of one q-gram increases and the other decreases: in the previous example three q-grams aaa disappear and appears aat, ata, taa. Thus each edit operation can increase the q-gram distance by $2q$ in the worst case. □

We now discuss how to implement the computation of the q-gram distance and what its space and time complexities are.

Let $\Sigma = \{a_0, a_1, ..., a_{\sigma-1}\}$ be an alphabet whose size is σ. Let s and t be two strings over Σ. To calculate the q-gram distance between s and t we create two profiles that we store in two arrays G_s and G_t of size σ^q. We cannot consider a q-gram v as an index, rather we have to turn it into an

integer. We can consider v as a σ-ary integer. Let $v = b_1 b_2 ... b_q$ be a q-gram on Σ, let us denote $\tilde{b}_i = j$ if $b_i = a_j$. Then the *integer code* of v is

$$\tilde{v} = \tilde{b}_1 \sigma^{q-1} + \tilde{b}_2 \sigma^{q-2} + ... + \tilde{b}_q \sigma^0.$$

We can use a strategy introduced for the Rabin-Karp algorithm [KR87] to efficiently calculate the integer codes for all q-grams of a string. Let $s = s_1 ... s_n$ be a string over Σ and let $v_i = s_i ... s_{i-q+1}$, $1 \leq i \leq n - q + 1$ be the i-th q-gram of s. Then

$$\tilde{v}_{i+1} = (\tilde{v}_i - \tilde{s}_i \cdot \sigma^{q-1}) \cdot \sigma + \tilde{s}_{i+q}. \tag{2.1}$$

Example 5. Let $\Sigma = \{A, T, C, G\}$, $\sigma = 4$. Let $s = CAGTGATT$ and $q = 4$. The first 4-gram is $v_1 = CAGT$ and the second 4-gram is $AGTG$. The integer code of the first and the second 4-grams are

$$\tilde{v}_1 = 2 \cdot 4^3 + 0 \cdot 4^2 + 3 \cdot 4^1 + 1 \cdot 4^0 = 141$$

$$\tilde{v}_2 = 0 \cdot 4^3 + 3 \cdot 4^2 + 1 \cdot 4^1 + 3 \cdot 4^0 = 55$$

Alternatively we can calculate v_2 using Equation 2.1:

$$\tilde{v}_2 = (141 - 2 \cdot 4^3) \cdot 4 + 3 = 55.$$

Note that when we have to implement the computation of the integer code of a q-gram using the methods in Example 5, they require two different times. The first method requires q steps, thus the time complexity is $O(q)$, while the method that uses Equation (2.1) requires constant time, i.e. $O(1)$.

Given the string s, we set the integer code of the first q-gram as $\tilde{v}_1 = \sum_{i=1}^{q} \tilde{s}_i \sigma^{q-1}$ and applying Equation (2.1) for $1 \leq i \leq n-q$ we get the integer codes of all q-grams in s. Simultaneously we count the occurrences of each q-gram in G_s by setting $G_s[\tilde{v}_i] \leftarrow G_s[\tilde{v}_i] + 1$, for all $1 \leq i \leq n - q$. The total time to compute G_s is $O(q + |s| - q) = O(|s|)$, composed of $O(q)$ to calculate $\tilde{v}_1$ and $O(|s|-q)$ to calculate the integer codes of all other q-grams. Similarly we compute G_t in time $O(|t|)$.

Now we can compute the difference between G_s and G_t: we scan simultaneously G_s and G_t and we take the absolute value of the difference of the i-th elements, for all $1 \leq i \leq \sigma^q$. We add it to the distance, i.e. $distance = distance + |G_s[i] - G_t[i]|$. The time complexity for this operation is $O(\sigma^q)$.

Alternatively it is also possible to archive complexity time $O(|s| + |t|)$ by applying the following strategy: we use two additional arrays in order to access only the q-grams that really appear in s and t. Thus we obtain the q-gram distance between s and t in $O(|s| + |t|)$. For details see Chapter 4.

We require σ^q integers for the array that represents a q-gram profile. Since all numbers are smaller or equal $|s| + |t|$, we need an integer with $\lceil \log_2(|s| + |t|) \rceil$ bits.

The space complexity for G_s and G_t is $O(2 \cdot \sigma^q)$. Therefore the total space for the strings and the profiles is $O(\sigma^q + |s| + |t|)$. An exponential space complexity is a limit for real applications, and it is also a waste of memory. In fact, for high values of q, at most $|t| + |s| - 2(q-1)$ elements of G_s and G_t are different from zero, resulting in a sparse vector. We can reduce the space requirements using any standard hashing technique, without increasing the time complexity. In Chapter 4 there are more details on both questions (for which value of q the vector is sparse and the hashing technique).

The most important model for sequence comparison in biology is the edit distance. However, when comparing biological sequences, the edit distance, which requires quadratic time, is too time-consuming. On the other hand, the q-gram distance is computable in linear time. We can exploit the q-gram distance as a filter for the edit distance. In fact, by the q-gram lemma, the q-gram distance provides a lower bound for the edit distance.

2.2 Graph representation

The first graph ever drawn can be traced back to almost 300 years ago, when the mathematician Leonhard Euler used a graph to represent the 'Bridges of Königsberg problem'.

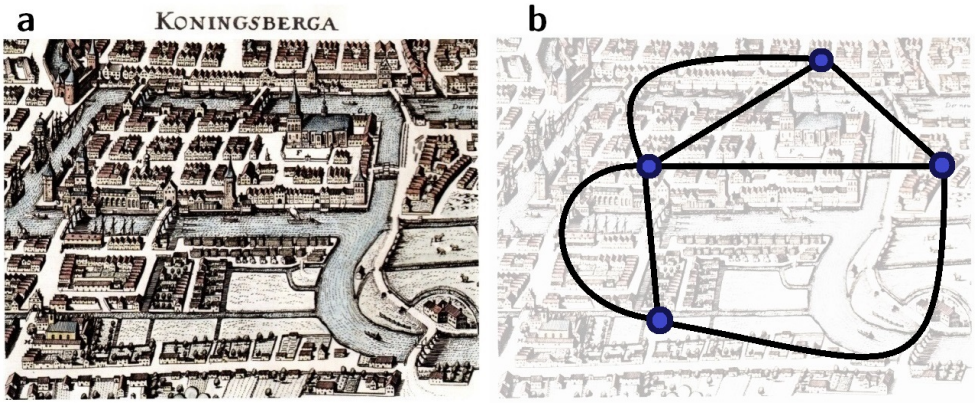


Figure 2.1: Bridges of Königsberg problem. (a) A map of Königsberg, engraving of Matthüs Merian, 1652. (b) The Königsberg Bridge graph, in which the nodes represent the four land areas of the city and the edges represent the seven bridges that connect them.

The city of Königsberg has seven bridges that connect the four parts of the city (Figure 2.1 a), the problem that plagued every resident was whether it was possible to visit every part of the city walking across each of the seven bridges exactly once and returning to one's starting location.

To solve the problem Euler represented each landmass as a point (called

a node) and each bridge as a line segment (called an edge) connecting two points, and thus created a graph (Figure 2.1 b).

After creating this stylized representation of the problem, Euler found a procedure for determining when an arbitrary graph contains a path that visits every edge exactly once and returns to where it began. We now call such a path an Euler circuit.

The graph drawn in Figure 2.1 b is an example of an *undirected graph*, in which the edges do not have direction. In this thesis we use only *directed graphs*, which have directed edges, drawn as an arrow from one node to another. We will now introduce the notion of directed graph and several other concepts, after that we will introduce the multigraphs. In a multigraph we can have more than one directed edge from one node to another. A simple graph is a directed graph having no loops or multiple edges. We will now make these notions precise.

The definitions mainly follow the book [Die12] by R. Diestel, and the lecture notes [Dic06] by A. Dickson.

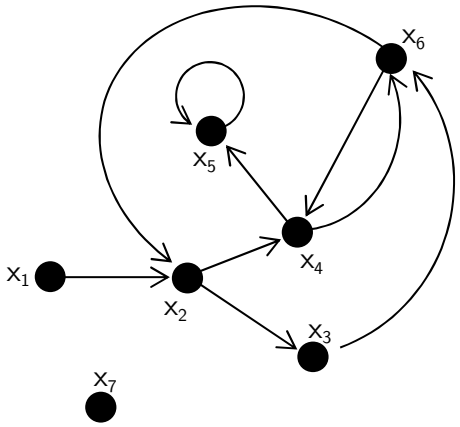


Figure 2.2: Example of a digraph on $V = \{x_1, x_2, x_3, x_4, x_5, x_6, x_7\}$ with edge set $E = \{(x_1, x_2), (x_2, x_3), (x_2, x_4), (x_3, x_6), (x_4, x_5), (x_4, x_6), (x_6, x_4), (x_5, x_5), (x_6, x_2)\}$

A *directed graph* (or *digraph*) is a pair $G = (V, E)$, where V is a finite set (called vertices or nodes) and E a set of ordered pairs of vertices (called arcs or edges), i.e. $E \in V \times V$. We refer to the set of vertices of a graph G as $V(G)$, and to its edge set as $E(G)$. Associated with a digraph, we have two maps, tail: $E \to V$ and head: $E \to V$ which assign to every edge e an *initial vertex* $tail(e)$ and a *terminal vertex* $head(e)$. The edge e is directed from $tail(e)$ to $head(e)$. The two vertices $tail(e)$ and $head(e)$ are also called *endpoints* of the edge e. Note that in our definition we do not exclude the possibility that $head(e) = tail(e)$, this is called a *loop* or *self-loop* (you can see an example in Figure 2.2 in the node x_5).

A *walk* in a graph G is a sequence $(v_0, e_1, v_1, e_2, v_2, ...e_k, v_k)$ of vertices and edges in G such that for all $i < k, tail(e_i) = v_{i-1}$ and $head(e_i) = v_i$. Since we can uniquely identify an edge in a walk from the node on its left and from the node on its right, therefore we can write a walk as a sequence of nodes: $(v_0, v_1, v_2, ...v_k)$ such that for all $1 < i < k, (v_{i-1}, v_i) \in E(G)$. The vertex v_0 is called the *beginning* of the walk, and v_k is the *end* of the walk. A *trail* is a walk with no repeated edges. If $v_0 = v_k$ then the trail is called *closed*, and we call it a *circuit*. A *path* is a trail with no repeated nodes. A *cycle* is a circuit with no repeated vertex, except for the first and last node.

Example 6. In Figure 2.2 we can see an example of:

- a walk, not a trail, not a path: $(x_1, x_2, x_4, x_6, x_4, x_6, x_4)$
- a trail, not a path: $(x_1, x_2, x_4, x_6, x_4, x_5)$
- a path: (x_1, x_2, x_4, x_6)
- a circuit, not a cycle: $(x_2, x_3, x_6, x_4, x_6, x_2)$
- a cycle: (x_2, x_3, x_6, x_2)

In Figure 2.3 we can see the relationship between walks, trails, circuits, paths and cycles.

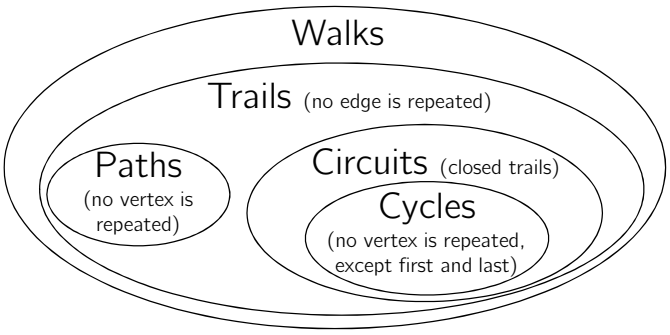


Figure 2.3: Representation of the relationship between walks and the derived definitions, following [Dic06].

A directed graph G is connected if the underlying undirected graph (take away direction from edges in G) is connected; an undirected graph is connected if any two of its vertices can be joined by a path. A *subgraph* of a graph $G = (V, E)$ is a graph $G' = (V', E')$ s.t. $V' \subseteq V$ and $E' \subseteq E$; note that when we say that G' is a graph we are implicitly saying that $E' \subseteq V' \times V'$.

If two vertices u, v in $V(G)$ are linked by an edge e, i.e. $\text{tail}(e) = u$ and $\text{head}(e) = v$, or $\text{tail}(e) = v$ and $\text{head}(e) = u$, then we say u and v are *adjacent*. Let $e = (u, v)$, we say that e is an *outgoing edge* of u and an

incoming edge of v. Given a node v belonging to a graph G, we define the *indegree* of v as the number of incoming edges in v, and we represent it as $d^-(v)$; the same way we define the *outdegree* of v as the number of outgoing edges: $d^+(v)$. We define the *balance* of a node v as $bal(v) = d^+(v) - d^-(v)$. Let G be a graph, we refer to the balance of the node v in G as $bal_G(v)$.

An *Euler trail*[1] is a trail in a graph which visits every edge exactly once. Similarly, an *Euler circuit*[2] is an Euler trail which starts and ends in the same vertex. We say that a graph G is *Eulerian* if it has an Euler circuit.

A (directed) *multigraph* is a triple (V, E, Ψ), where (V, E) is a graph and Ψ is a function $\Psi : E \to \mathbb{N}^*$, which assigns to every edge its multiplicity. In Figure 2.4 we can see an example of a multigraph where $\Psi(e_3) = 1$ and $\Psi(e_4) = 2$. Let G be a multigraph, we refer to the multiplicity of an edge $e \in E(G)$ as $\Psi_G(e)$. When drawing a graph G, usually we represent an edge e as $\Psi(e)$ different edges.

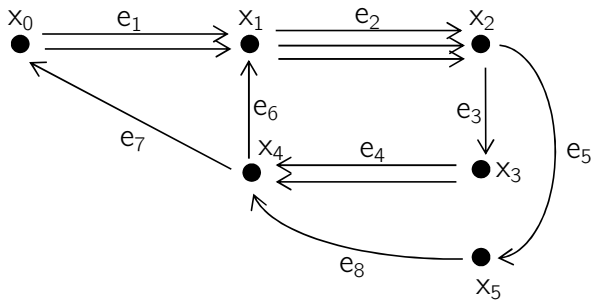


Figure 2.4: Example of a multigraph on $V = \{x_0, x_1, x_2, x_3, x_4, x_5\}$ with edge set $E = \{(x_0, x_1), (x_1, x_2), (x_2, x_3), (x_3, x_4), (x_2, x_5), (x_4, x_1), (x_4, x_0), (x_5, x_4)\}$ and $\Psi(e_1) = 2$, $\Psi(e_2) = 3$, $\Psi(e_3) = 1$, $\Psi(e_4) = 2$, $\Psi(e_5) = 1$, $\Psi(e_6) = 1$, $\Psi(e_7) = 1$, $\Psi(e_8) = 1$.

A *simple graph* is a graph having no loops or multiple edges. Thus, a simple graph is a multigraph without loops and where $\Psi(e) = 1$ for all $e \in E$.

All definitions of walk, trail, path, circuit and cycle carry over from simple graphs to multigraphs, respecting the multiplicity of edges. This means that in a trail and in a circuit each edge e is used at most $\Psi(e)$ times.

Example 7. In Figure 2.4 we can see an example of:

- a walk, not a trail, not a path: $(x_2, x_5, x_4, x_1, x_2, x_5)$
- a trail, not a path: $(x_1, x_2, x_3, x_4, x_1, x_2, x_5)$
- a path: $(x_0, x_1, x_2, x_5, x_4)$

[1] often also called an Euler path.
[2] often also called an Euler cycle or Euler tour.

- a circuit, not a cycle: $(x_0, x_1, x_2, x_3, x_4, x_1, x_2, x_5, x_4, x_0)$
- a cycle: $(x_1, x_2, x_3, x_4, x_1)$

All definitions of indegree, outdegree, balance and the definitions of Euler trail and Euler circuit, carry over respecting the multiplicity of edges.

A fundamental theorem of graph theory combines the definition of Euler trail and Euler circuit with the information of the balance of the nodes in the graph. The original version of the theorem for undirected multigraphs had been given, without proof, by Leonhard Euler in 1736. The theorem was rigorously proved in 1873 by Carl Hierholzer [Hie73, Bar09].

Theorem 1. *A connected multigraph G contains an Euler circuit if and only if $bal(v) = 0$ for all $v \in V(G)$. We call such a graph Eulerian.*

Theorem 2. *A graph G has an Euler trail if and only if it is a connected multigraph G such that $bal(v) = 0$ for all $v \in V(G)$, or exists two vertices u, v s.t. $bal(v) = 0$ for all $v \in V(G) \setminus \{u, w\}$, $bal(u) = -1$ and $bal(w) = 1$. The node w is the* beginning *and the node u is the* end *of the Euler trail.*

2.2.1 Balanced de Bruijn subgraph

Nicolaas de Bruijn introduced, in 1946 [dB46], a new type of graph to solve the 'superstring problem': find a shortest string that contains all possible q-grams. A balanced de Bruijn subgraph is a subgraph of the de Bruijn graph defined by N. de Bruijn.

Definition 4. Let s be a string of length n and $2 \leq q \leq n$. The *balanced de Bruijn subgraph* is a directed multigraph $DBG(s, q) = (V, E, \Psi, label)$ defined as follows:

- The vertex set V corresponds to the set of all the $(q-1)-$grams of s;
- There is a directed edge from x to y if there exists a q-gram in s whose prefix is x and whose suffix is y:

  $E = \{(x, y) \mid x, y \in V,$ such that there exists a q-gram $z = xy_q = x_1 y$ in $s\}$

- At each edge $(x, y) \in E$ is associated a label equal to the $q-$gram that connects x and y. We label each edge $e = (x, y)$ by the q-gram xy_q;
- A multiplicity function $\Psi : E \to \mathbb{N}^*$. For every edge e, $\Psi(e) = N(s, label(e))$.

Note that a balanced de Bruijn subgraph is constructed from overlapping q-grams and hence is connected. It is 'balanced' in the sense that the vertices

are balanced and thus that the graph contains an Euler trail or an Euler circuit.

To illustrate Definition 4, we show the balanced de Bruijn subgraph for the string $s = ABCAACABCAABC$ over the alphabet $\Sigma = \{A, B, C\}$ where $q = 4$, in Figure 2.5.

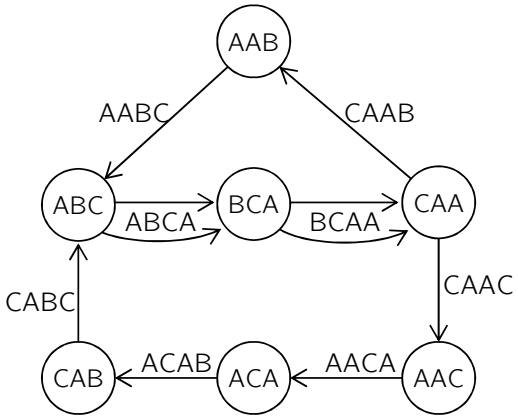


Figure 2.5: The de Bruijn graph DBG(s, 4) for the string $s = ABCAACABCAABC$.

We can see in the example in Figure 2.5 that if we start from the node ABC, which is the first $(q-1)$-gram of s, and we move through the graph following the $q-$grams that we find in the string as edges, we find a circuit. In particular, we find an Euler circuit, since we use every edge a number of times equivalent to its multiplicity.

The following theorem is easy to see.

Theorem 3 ([Pev89])**.** *Let s be a string and choose q. There is a one-to-one correspondence between Eulerian trails in $DBG(s, q)$ and strings with the same q-gram profile as s.*

Let s be a string of length n, choose q s.t. $1 \leq q \leq n$. Let $G = DBG(s, q)$ and let α be the Euler trail in G corresponding to s. Then by Theorem 3, there exists a string $t \neq s$ s.t. $d_q(s, t) = 0$ if and only if there exists an Euler trail $\beta \neq \alpha$ in G.

Pevzner [Pev95] used the balanced de Bruijn subgraph to represent and study the q-gram distance, and the *transformations* introduced by Ukkonen [Ukk92].

When studying the q-gram distance, we saw that there are two transformations on a string s that do not change the q-gram profile of s, which are rotation and transposition. We can visualize these two transformations on a balanced de Bruijn subgraph. In Figure 2.6 we can see the rotation transformation, and in Figure 2.7 there is an example of transposition. Note

that in order to have a rotation in a string s, $DBG(s, q)$ must contain an Euler circuit.

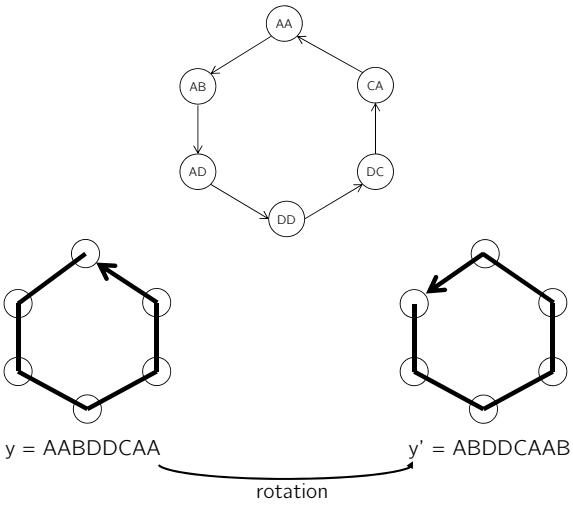


Figure 2.6: The representation of the rotation transformation, as defined by Ukkonen, on a balanced de Bruijn subgraph. Under the graph we represent two Eulerian circuits and the corresponding strings. This graph representation follows [Pev95].

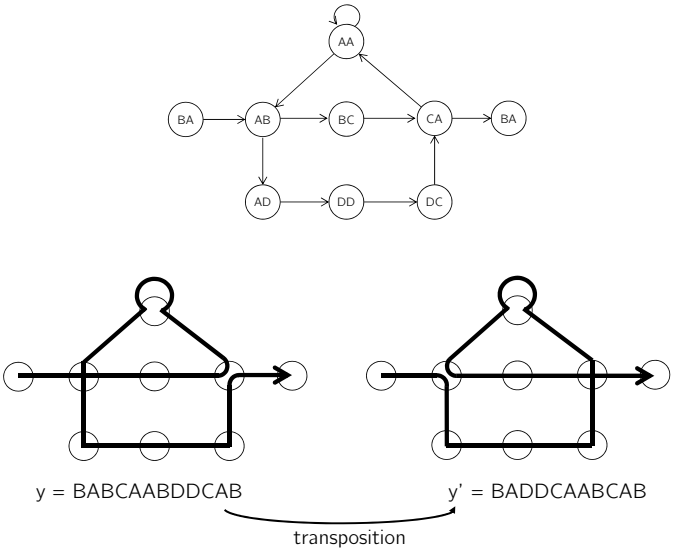


Figure 2.7: Representation of the transposition transformation, as defined by Ukkonen, on a balanced de Bruijn subgraph. Under the graph we represent two Eulerian trails and the corresponding strings. This graph representation follows [Pev95].

Imprecision in Ukkonen transformations

Let s be a string and choose q. We recall the two Ukkonen transformations:

1. (transposition) Let z_1 and z_2 be $(q-1)$-grams and y_i be strings, for $i \in \{1, ..., 5\}$. If s can be written as $s = y_1 z_1 y_2 z_2 y_3 z_1 y_4 z_2 y_5$, then the string $t = y_1 z_1 y_4 z_2 y_3 z_1 y_2 z_2 y_5$, where y_2 and y_4 are exchange, is also in $Z_q(s)$. If z_1 and z_2 coincides we have a similar transformation. Let z be a $(q-1)$-gram, if s can be written as $s = y_1 z y_2 z y_3 z y_4$, then the string $t = y_1 z y_3 z y_2 z y_4$ is in $Z_q(s)$.

2. (rotation) Let z_1 and z_2 be $(q-1)$-grams and y_1 and y_2 be strings. If s can be written as $s = z_1 y_1 z_2 y_2 z_1$, then also $t = z_2 y_2 z_1 y_1 z_2$ is in $Z_q(s)$.

Ukkonen conjectured [Ukk92] and Pevzner proved [Pev95] that repeatedly applying the two Ukkonen transformations is possible to generate the whole set $Z_q(s)$. We found an imprecision in this classic theorem of Ukkonen/Pevzner. We present it in the following example, where two strings s and t have the same q-gram profile but it is not possible to transform s in t using the Ukkonen transformations.

Example 8. Let $\Sigma = \{a, b, c\}$ be an alphabet and let $q = 4$. The two strings:
$s = abcbbcabcccab$,
$t = abcccabcbbcab$,
have the same 4-gram profile and the same balanced de Bruijn subgraph, as we can see in Figure 2.8.

	$P_4(s)$	$P_4(t)$
abcb	1	1
bcbb	1	1
cbbc	1	1
bbca	1	1
bcab	1	1
cabc	1	1
abcc	1	1
bccc	1	1
ccca	1	1
ccab	1	1

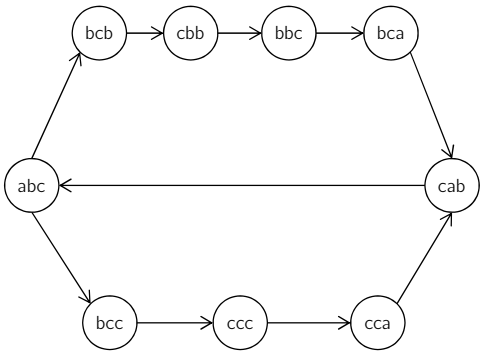


Figure 2.8: On the left there are the q-gram profiles of the strings $s = abcbbcabcccab$ and $t = abcccabcbbcab$. On the right the de Bruijn graph $DBG(s, 4)$, which is the same as $DBG(t, 4)$.

Hence there must be a sequence of Ukkonen transformations that transform s in t. But we cannot apply any Ukkonen transformation to s:

1. cannot apply transposition of the first type to s: the only $(q-1)$-grams which occurs at least twice are *abc* and *cab*:

   $$s = \underline{abc}bb\underline{\underline{cab}}c\underline{ccab}$$

   I cannot write s in the desired form.

2. we cannot apply transposition of type 2 because no $x \in \Sigma^{q-1}$ occurs 3 times.

3. cannot apply rotation to s because the first $(q-1)$-gram is different from the last $(q-1)$-gram.

Basically we have to apply a transposition like in point 1 in the example, but the problem is that we have an overlap between the $(q-1)$-grams z_1 and z_2. If we look at $DBG(s, 4)$ in Figure 2.8, it is possible to find two Eulerian trails that start in *abc* and end in *cab*: one that uses the upper path first (i.e. through *bcb*), which corresponds to the string s; and one that uses the lower path first, which corresponds to the string t.

Therefore, the imprecision is in the formulation of the Ukkonen transformations: we have to include the possibility of having an overlap between the $(q-1)$-grams z, z_1, z_2.

If we look at $DBG(s, 4)$ in Figure 2.8, we can see that the path that comes back from *cab* to *abc* is composed only of one edge, and this corresponds exactly to the overlapping of z_1 and z_2 that prevents the application of the transposition. If the path from *cab* to *abc* would have been composed of at least $q-1$ edges, 3 in the example, then we would not have the overlapping between z_1 and z_2.

Chapter 3

Threshold q-gram distance

Contents

In this chapter we study a modified version of the q-gram distance which uses a $threshold \in \mathbb{N}$. We modify the way we count the occurrences of a q-gram x to calculate the q-gram profile: when we reach a value equal to $threshold + 1$ we stop increasing the occurrence count of x. We can imagine this as if a member of the tribe of the Pirahã is counting the number of q-grams in a string. Daniel Leonard Everett studied the language of the Pirahã people in Amazonia and found that the words for numbers appear limited to 'one', 'two' and 'many' [Eve05]. Following our definitions the Pirahã people are counting using a $threshold$ equal to 2.

In this thesis we consider only $threshold = 1$. In this way, given a string s and a q-gram x, the value of the occurrence count of x in s, which is $N(s, x)$, is 0 if x does not appear in s, 1 if x occurs exactly once and 2 if it occurs at least two times.

The idea is that in this way we identify the q-grams that appears only once in the string, which we call *hapax*, and those that appear at least twice, which we call *repeats*.

This chapter is organized as follows: first we define the *threshold* q-gram distance, then we introduce a modified version of the balanced de Bruijn subgraph, which we call protograph. A protograph can represent the *threshold* q-gram profile, as the balanced de Bruijn subgraph represents the q-gram profile. Using the protograph, in Section 3.3 and 3.4 we study how we can transform one string into another without changing the *threshold* q-gram distance. In addition to the two transformations described by Ukkonen, we describe three transformations that are present only using the *threshold* q-gram distance.

In Section 3.5 we introduce a way to represent a protograph using a system of linear equations and a system of linear inequalities.

3.1 The threshold q-gram distance

Let Σ be an alphabet, let x in Σ^q be a q-gram, let s be a string in Σ^*. Let $S(s, x)$ denote the *status* of x in s, which is defined as:

$$S(s,x) = \begin{cases} 0 & \text{if } x \text{ does not occur in } s \\ 1 & \text{if } x \text{ occurs exactly once in } s \\ \infty & \text{if } x \text{ occurs at least two times in } s \end{cases}$$

Let s and t be strings in Σ^*, let x in Σ^q be a q-gram. We denote by $PS(s, t, x) = (S(s, x), S(t, x))$ the *pair-status* of s and t respect to x.

Given a q-gram x, a pair-status equal to $(1, 1)$ means that x appears as hapax in both strings, a pair status equal to $(0, 2)$ means that x does not appear in the first string and is a repeat in the second string. The same way we can interpret all 9 possible pairs.

Let $s \in \Sigma^n$, let $\sigma = |\Sigma|$ and let $1 \leq q \leq n$. The *threshold q-gram profile* of s is σ^q-vector $T_q(s) = (S(s, x))_{x \in \Sigma^q}$.

Definition 5. Let s and t be strings of length n and m respectively, and let q be a number $1 \leq q \leq \min\{n, m\}$. The *threshold q-gram distance* between s and t is

$$TQD_q(s,t) = |\{x \in \Sigma^q \mid S(s,x) \neq S(t,x)\}|.$$

In other words, the *threshold* q-gram distance is the Hamming distance between $T_q(s)$ and $T_q(t)$. Here we use the Hamming distance (for the q-gram distance we used the Manhattan distance) because we want to measure the number of q-grams in the two strings that have different status.

Example 9. Let $\Sigma = \{A, B\}$ be an alphabet, let $s = ABAABBBB$, of length 8, and $t = AABBBBBBBBAB$, of length 12, be strings over Σ and let $q = 2$. Using lexicographic order (AA, AB, BA, BB) among the q-grams, the two 2-gram profiles are:

$$P_2(s) = (1, 2, 1, 3)$$

$$P_2(t) = (1, 2, 1, 7)$$

The two *threshold* 2-gram profiles are:

$$T_2(s) = (1, \infty, 1, \infty)$$

$$T_2(t) = (1, \infty, 1, \infty)$$

The 2-gram distance is equal to

$$d_2(s, t) = 4$$

The threshold 2-gram distance is equal to

$$TQD_2(s, t) = 0$$

From the Example 9 we observe the following:

1. the *threshold* q-gram distance between two strings s and t can be 0 even if the $s \neq t$ as we saw for the q-gram distance;
2. the *threshold* q-gram distance between two strings s and t can be 0 even if the q-gram distance between s and t is greater than 0;
3. two strings with the same *threshold* q-gram profile can have different length, while we saw that two strings with the same q-gram profile have the same length.

Given two strings s and t, the *threshold* q-gram distance between s and t is less or equal to the q-gram distance calculated on the same two strings:

$$TQD_q(s, t) \leq d_q(s, t)$$

Thus we have that the q-gram lemma is also applicable to the *threshold* q-gram distance. Let s and t be strings, let d_E denote the edit distance, then we have that

$$TQD_q(s, t)/(2q) \leq d_E(s, t)$$

The *threshold* q-gram distance is a pseudometric: for all $s, t, z \in \Sigma^*$:

1. $TQD_q(s, t) \geq 0$ (non-negativity),
2. $TQD_q(s, t) = TQD_q(t, s)$ (symmetry),

3. $TQD_q(s,t) \leq TQD_q(s,z) + TQD_q(z,t)$ (triangle inequality).

This can be seen as follows: the *threshold* q-gram distance is calculated as a sum of absolute values, and this explains the non-negativity. The symmetry is due to the fact the we count the number of q-grams that have a different status in the two strings: there is no order between the profiles. The triangle inequality can be rewritten using the definition of *threshold* q-gram distance:

$$\begin{aligned} |\{x \in \Sigma^q \mid S(s,x) \neq S(t,x)\}| \quad \leq \quad & |\{x \in \Sigma^q \mid S(s,x) \neq S(z,x)\}| \; + \\ & |\{x \in \Sigma^q \mid S(z,x) \neq S(t,x)\}| \end{aligned}$$

Let y be a q-gram, let w, u be two strings, we define:

$$E(s,w,u) = \begin{cases} 1, & \text{if } S(w,y) = S(u,y) \\ 0, & \text{otherwise} \end{cases}$$

Let $A(y) = E(s,t,y)$, $B(y) = E(s,z,y)$, and $C(y) = E(z,t,y)$. We prove that $A(x) \leq B(x) + C(x)$ for all $x \in \Sigma^q$ and hence the triangle inequality. We have four combination of $B(x)$ and $C(x)$. If $B(x) = 0$ (i.e. $S(s,x) = S(z,x)$) and $C(x) = 0$ (i.e $S(z,x) = S(t,x)$) then $S(s,x) = S(t,x)$ and $C(x) = 0$. If $B(x) = 1$ and/or $C(x) = 1$, then $A(x) \leq B(x) + C(x)$ is always true: on the right of the inequality we have at least one and $A(x)$ is at most one.

3.2 Protograph definitions

We saw that a string divided into q-grams can be represented by a de Bruijn graph. We would like to find a similar representation which can model the threshold q-gram distance. Thus we have to indicate in a different way, in the graph, the q-grams that are repeated: we represent the q-grams that are repeated with one type of arc (fat edge), and those not repeated with another type of arc (thin edge). We call such a graph a protograph. If we substitute a fat edge with two or more arcs we obtain a multigraph G, which we call a realization of the protograph H. In Figure 3.1 we can see an example of these graphs.

We can see a protograph as a multigraph where we zoom-out: from the distance you can see the edges that are used only once, but you cannot recognise how many edges there are if the multiplicity is greater than 1. And when you zoom-in there are many possibilities for the nuanced edges (the fat edges).

We will now make the ideas of protographs and realizations more precise.

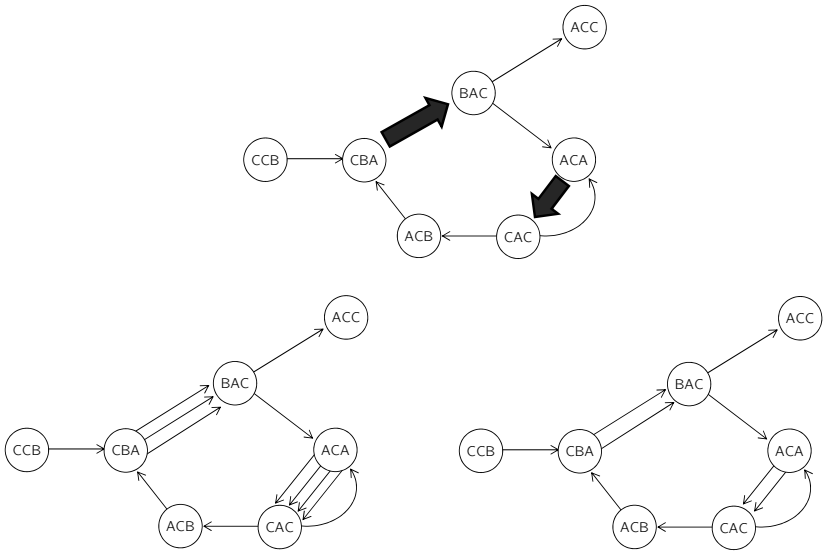


Figure 3.1: At the top of the figure we see a protograph, and below we see two different realizations.

Definition 6. A *protograph* H is a simple directed graph with loops, with two types of edges: thin edges and fat edges. It is defined by the triple (V, E, Γ) consisting of a vertex set $V = V(H)$, an edge set $E = E(H)$, and a function $\Gamma = \Gamma(H)$. The function is defined as $\Gamma : E \to \{0, 1\}$. We refer to an edge as *fat edge* if $\Gamma(e) = 1$, and as a *thin edge* if $\Gamma(e) = 0$.

Given a protograph H, we define two sets $E_F(H)$ and $E_T(H)$ as follows:

- $E_F(H) = \{e \in E(H) \mid \Gamma(e) = 1\}$
- $E_T(H) = \{e \in E(H) \mid \Gamma(e) = 0\}$

Let G be a multigraph, we call $H(G)$ the *protograph of* G, a protograph defined as follows:

- $V(H(G)) = V(G)$;
- $E(H(G)) = E(G)$;
- for all $e \in E(H(G))$: $\Gamma(e) = 1$ if $\Psi(e) \geq 2$, else $\Gamma(e) = 0$ (i.e. $\Gamma(e) = 0$ if and only if $\Psi(e) = 1$).

Let G be a directed multigraph defined by the triple (V, E, Ψ). We call G a *realization* of H if:

- $V(G) = V(H)$
- $E(G) = E(H)$.

- $\forall e \in E(G) : \Psi(e) \geq 2 \Leftrightarrow \Gamma(e) = 1$

Note that every multigraph G is a realization of $H(G)$; but there may exists other realizations $G' \neq G$ of $H(G)$.

A graph G is an *Eulerian realization* of protograph H if

1. G is a realization of protograph H, and
2. G contains an Eulerian trail or an Eulerian circuit.

We can see in Figure 3.1 a protograph and two realizations of it: only the realization on the right is an Eulerian realization.

We call a protograph H *Eulerian* if it has at least one Eulerian realization.

Recall that we denote by $DBG(s, q)$ the balanced de Bruijn subgraph of the string s using q for the value of the q-gram. Let s be a string and $q \in \mathbb{N}^*$. We define $H(s, q)$ the *protograph constructed on the string* s as $H(s, q) = H(DBG(s, q))$. When q is clear from the context, we just write $H(s)$. $H(s, q)$ is Eulerian because by construction $DBG(s, q)$ is Eulerian or contains an Euler trail.

Let s be a string and choose q, every Eulerian realization of $H(s, q)$ is to a balanced de Bruijn subgraph. Note that the number of Eulerian realizations of $H(s, q)$, hence the number of balanced de Bruijn subgraphs, could be greater than one. In other words, given a multigraph G, there is only one $H(G)$, but given a protograph H we may be able to find many realizations.

Let H be a protograph. We denote by $G^* = G^*(H)$ the set of realizations of H: $G^* = \{G \mid G \text{ is a realization of } H\}$. We denote by $G^E = G^E(H)$ the set of Eulerian realizations of H: $G^E = \{G \mid G \text{ is an Eulerian realization of } H\}$. We have that $G^E \subset G^*$.

Let H be a protograph. The *fat-graph* F of H, denoted $F(H)$, is a simple directed graph, a subgraph of H such that:

1. $E(F) = E_F(H)$
2. $V(F) = \{v \in V(H) \mid v \text{ is an endpoint of a fat edge in } H\}$

When we gave the definitions on graphs, we defined the indegree and the outdegree. These definitions are no longer valid for a protograph because it has two types of edges, so we define:

- $d_F^-(v) =$ the number of incoming fat edges of v;
- $d_F^+(v) =$ the number of outgoing fat edges of v;
- $d_T^-(v) =$ the number of incoming thin edges of v;
- $d_T^+(v) =$ the number of outgoing thin edges of v;

Let H be a protograph. We would like to define a number that represents the maximum number of thin edges into which a fat edge can be transformed in any Eulerian realization of H.

Definition 7. Let H be a protograph that does not contain directed fat cycles. For every fat edge $e = (u, v)$ we define

$$M(e) = |\{e_t \mid e_t \text{ is an incoming thin edge of } u\}| + 1 + \sum_{\substack{f \text{ is an incoming} \\ \text{fat edge of } u}} M(f).$$

Here is an example of how to calculate M:

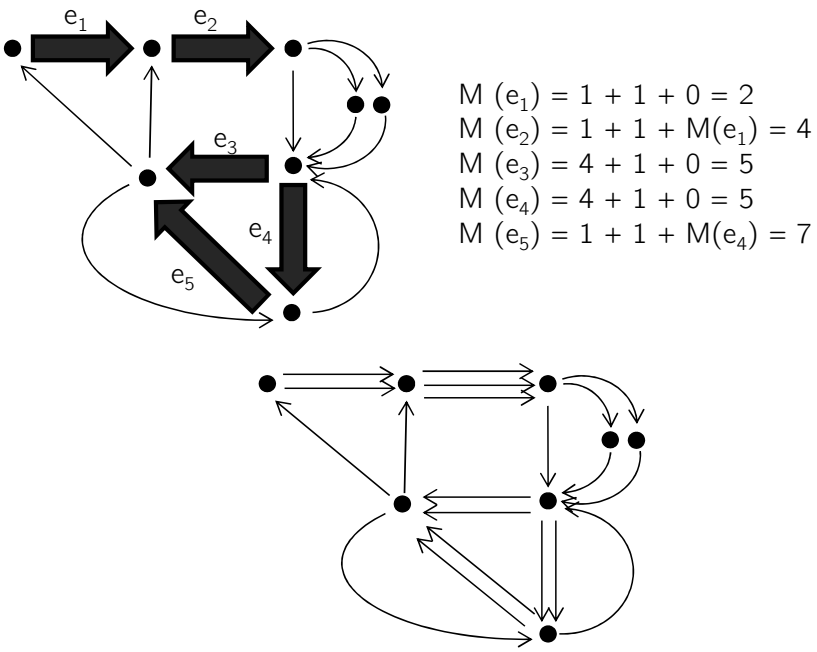


Figure 3.2: Example of a protograph (upper left) and its unique realization (below). We calculated the M of each fat edge (upper right).

The protographs that contain a directed fat cycle have an infinite number of realizations, see Theorem 8. Let H be a protograph constructed over a string, let C be a directed fat cycle in H, let p be an Eulerian trail in H (p exists because H is constructed over a string) and let u be a node that belongs to C. When we encounter u in the trail p, we can go through C many times and then complete the Euler trail.

The definition of M is recursive, i.e. implies the knowledge of M of the incoming fat edges. We now show that M is well defined and hence that it is always possible to calculate it.

Let H be an Eulerian protograph that does not have directed fat cycles. The fat-graph F of H is a directed acyclic graph.

A *directed acyclic graph* (DAG) is a directed graph with no directed cycles. There are no trails that start in v and come back to v. Thus every trail in a DAG is a path. In Figure 3.3 there is an example of a DAG. Note that this is the fat-graph of the protograph in Figure 3.2.

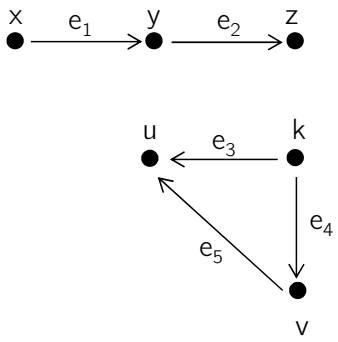


Figure 3.3: Example of a directed acyclic graph.

DAGs are widely used to represents problems of ordering a collection of objects, subject to constraints that certain object must perform an action earlier than others.

Every DAG has a *topological ordering* of the vertices, i.e. an ordering such that, for every edge e, the node $tail(e)$ occurs earlier in the ordering than the node $head(e)$. In general, this ordering is not unique. For example, two possible topological ordering of the DAG in Figure 3.3 are: (1) k, v, u, x, y, z and (2) x, y, z, k, u, v.

As we can see in Figure 3.4 if we write the nodes in topological order, then each edge is pointing to the right.

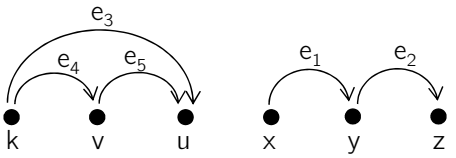


Figure 3.4: Example of a directed acyclic graph.

It is possible to compute a topological ordering on a DAG in time $O(|V|+|E|)$ [CLRS09].

For a DAG, we will call a *topological ordering of the edges* an ordering such that, for every pair of edges e_1 and e_2, if $head(e_1) = tail(e_2)$ then e_1 occurs earlier in the ordering than e_2. In general, this ordering is not unique.

For example, two possible topological orderings of the edges of the DAG in Figure 3.3 are:
1) e_1, e_2, e_3, e_4, e_5 and 2) e_4, e_1, e_3, e_5, e_2.

We can compute a topological ordering for the edges in the following way: first we create a topological ordering for the nodes, then we start from the first node to the last one and for every one we add all the outgoing edges, in a random order, to the ordering of the edges such that they came after all previously inserted edges.

Let H be a protograph that does not contains a directed fat cycle, and let e be a fat edge. To calculate $M(e)$ we proceed as follows:

1. Compute topological ordering of nodes of $F(H)$;
2. Compute topological ordering of edges of $F(H)$, say $e_1, e_2, ..., e_m$;
3. Now compute $M(e_i)$, for $i = 1, ..., m$ in this order.

3.3 Transformations

We saw that the *threshold* q-gram distance between two strings s and t can be 0 even if $s \neq t$. Given a string s we might ask how to find all strings $t \neq s$ s.t. $TQD(s, t) = 0$, in other words all strings t with the same *threshold* q-gram profile of s, i.e. $\{t \mid P_q(t) = P_q(s)\}$.

Ukkonen found two transformations on strings that do not change the q-gram profile of a string, which are transposition and rotation (see Section 2.1.3). Since the *threshold* q-gram distance can be 0 even if the q-gram distance is greater then 0, we expect that in the *threshold* version we have additional transformations.

We divide the problem of finding all strings with the same *threshold* q-gram profile in two subproblems:

1. find how we can obtain all different Eulerian realizations from a protograph, and
2. use on these realizations the previous transformations, which we called Ukkonen transformations.

You can see in Figure 3.5 a representation of this concept.

Let $s \in \Sigma^n$ be a string and let q be an integer s.t. $1 \leq q \leq n$. To find all strings with the same *threshold* q-gram profile of s:

1. we create the balanced de Bruijn subgraph $G = DBG(s, q)$;
2. we create the protograph $H(G)$;
3. we create the set $G^E(H)$ of all the Eulerian realization of H using the realization transformations;

4. on every balanced de Bruijn subgraph $G \in G^E(H)$ we find a string s which correspond to an Euler trail in G, and we apply the Ukkonen transformations to find all strings in G.

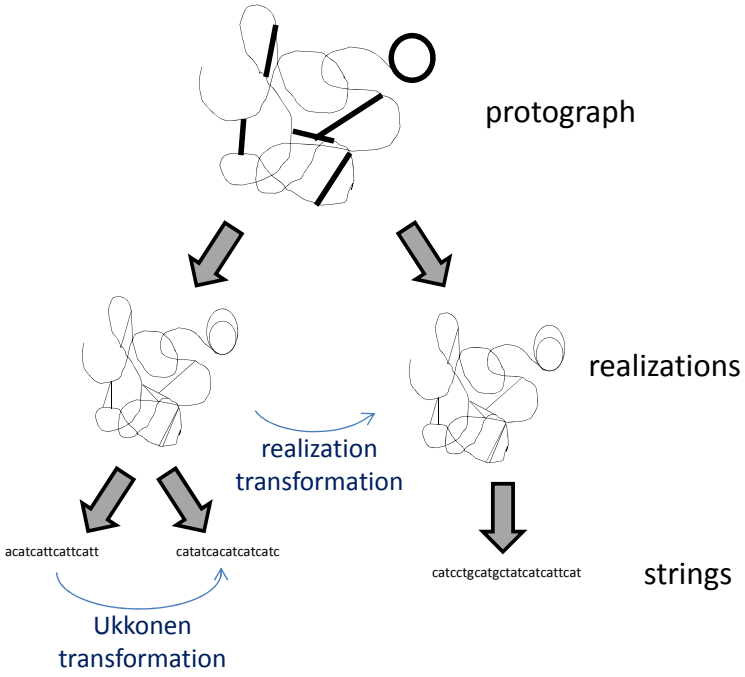


Figure 3.5: Representation of the realization transformations and the Ukkonen transformations.

The protograph defined in the previous paragraph is useful to study the realization transformations. We can study the *properties* on a protograph that, if present, result in at least two realizations. Some of the properties are better defined on the realizations, rather than on the protograph. In this case we will show that if a balanced de Bruijn subgraph G has a property π, then we can find another realization $G' \neq G$ s.t. $H(G) = H(G')$. In this section we study these properties and then in Section 3.4 we find the corresponding *operations* on strings.

We have a first theorem that shows a relation between the Ukkonen operations and the number of realizations.

Theorem 4. *Let s be a string, G the balanced de Brujin subgraph constructed from s and H the protograph of G s.t.*

1. *G is Eulerian, and*

2. *$E_F(H) \geq 1$,*

then H has at least two Eulerian realizations (we recall that a graph G is an Eulerian realization of H if G contains an Euler circuit or an Euler trail).

Proof. We want to show that starting from G, with the two properties below, we can construct a balanced de Brujin subgraph $G' \neq G$ s.t $H(G) = H(G')$.

Since G is Eulerian, it follows that for all v in V : $bal(v) = 0$.

Let (uv) be a fat edge in H.

G is a connected graph because it is constructed on a string, and every node has balance 0. Hence there exists an Euler circuit that starts in u and ends in u, which we call φ.

We define G' as follows:

- $V(G') = V(G)$
- $E(G') = E(G)$
- $\forall e \in E(G) \setminus \{(u, v)\} : \Psi_{G'}(e) = \Psi_G(e)$,
  $\Psi_{G'}(u, v) = \Psi_G(u, v) + 1$.

G' is a realization of H, because:

- $V(G') = V(G) = V(H)$
- $E(G') = E(G) = E(H)$
- the thin edges of H, are realized in the same way in G and G'; we only add an edge in G' between u and v; but in H this is a fat edge.

In particular, G' is an Eulerian realization of H, because:

- G' is a realization of H
- G' contains an Euler trail, which is φ, v. In fact the balance of each node in G is 0, in G' we have added an edge between u and v. Hence in G' we have $bal(u) = 1$ and $bal(v) = -1$ and all the other nodes have balance zero.

Thus, G and G' are two distinct Eulerian realizations of H. □

Corollary 1. *Let s be a string. If $H(s)$ contains at least one fat edge and if we can apply a rotation transformation to s, then $H(s)$ has at least two Eulerian realizations.*

Proof. Follows from Theorem 4 and the fact that we can apply a rotation transformation to s if and only if $DBG(s, q)$ is Eulerian. □

In the following, we will show that there are three properties that create more than one realization:

1. change the beginning or the end of an Euler trail;
2. alternating fat cycle;
3. fat cycle.

We will first introduce each of these properties with examples and then we will formally define them on the protograph or on a realization. After that we prove that every time we have this property, then we have more than one realization.

Finally, we will conjecture that avoiding these three properties is not only a necessary but also a sufficient condition for uniqueness of the Eulerian realization.

3.3.1 Change the beginning or the end of an Euler trail

As we saw from the Theorem 4, in order to have only one realization we have to fix the beginning and the end of the Euler trail in the protograph. In Figure 3.6 and 3.7 we can see two examples.

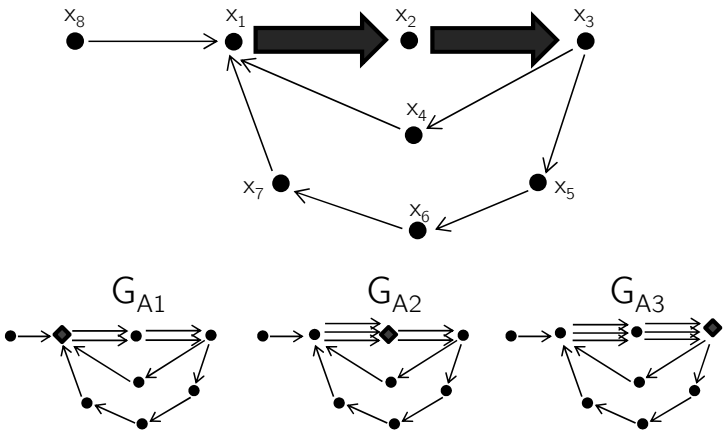


Figure 3.6: An example of a protograph where we can change the end of the Euler trail. In the three Eulerian realizations below we indicate with a rhombus the end node.

In Figure 3.6 we could change the end of the Eulerian trail: we see that the Eulerian trail must start from x_8, go to x_3, then back to x_1, again to x_3 and back to x_1. Once we are here, we have used each thin edge exactly once and each fat edge twice, so this is a realization (see G_{A1}). But from x_1 we can go on to x_2, and this is still a realization of the initial protograph (see G_{A2}), and the same for G_{A3}.

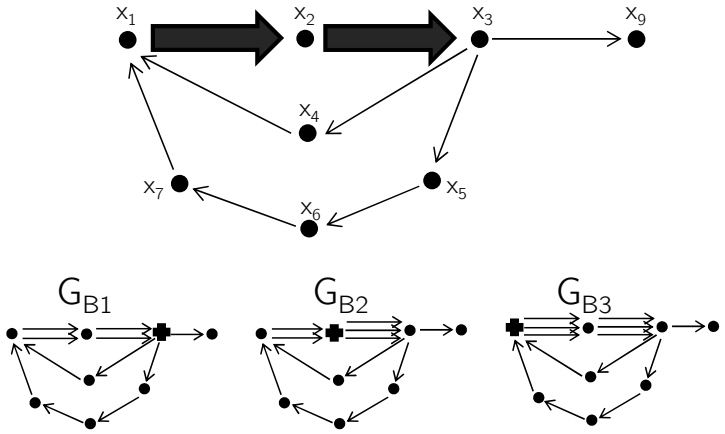
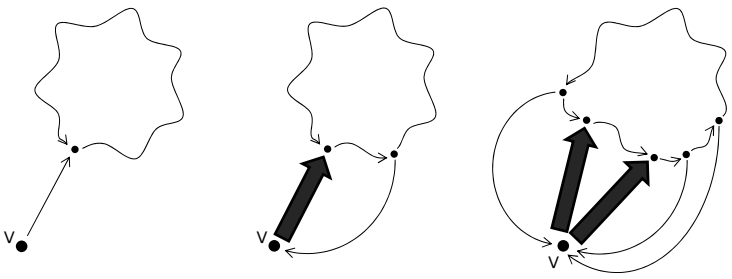


Figure 3.7: An example of a protograph where we can change the beginning of the Euler trail. In the three Eulerian realizations below we indicate with a cross the first node.

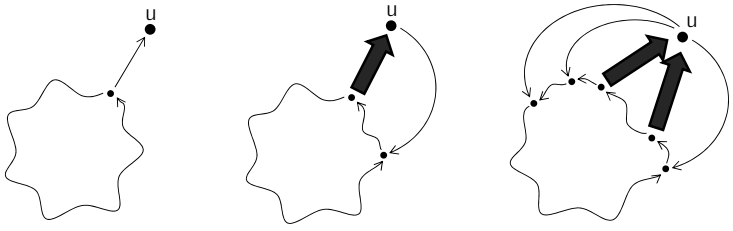
In Figure 3.7 we change the initial node of the Eulerian trail. In G_{B1} we start from x_3 and end in the same node and finally we add x_9, in G_{B2} we start from x_2 and end in x_9 and in G_{B3} we go from x_1 to x_9.

As we said, we want that the beginning (resp. end) of the Euler trail is fixed. In other words, every Euler trail must begin (resp. end) in the same node.

Let H be a protograph. The beginning of the Euler trail is fixed if H contains a node v as follows (formal definition to follow):



The end of the Euler trail is fixed if the protograph H contains a node u like this (formal definition to follow):



Definition 8. A protograph H *has the beginning fixed* if there exists a node $v \in H$ s.t.

$$d_F^-(v) = 0 \text{ and } 2 \cdot d_F^+(v) - 1 = d_T^-(v) - d_T^+(v).$$

H *has the end fixed* if there exists a node $u \in H$ s.t.

$$d_F^+(u) = 0 \text{ and } 2 \cdot d_F^-(u) - 1 = d_T^+(u) - d_T^-(u).$$

On the other hand, if either the beginning or the end is not fixed, i.e. if one of the two equations does not hold, then we have more than one Eulerian realization. We will prove this statement in Theorem 5, but first we introduce in Lemma 2 and Lemma 3 two concepts that are useful for proving the theorem.

In Lemma 2 we state the if the beginning is fixed in the node v, then every outgoing fat edge of v is used exactly 2 times in any Eulerian realization of H. An analogous statement holds if the end is fixed.

Lemma 3 states that if the beginning is not fixed in v, but v does not have an incoming fat edge, then there is at least one outgoing fat edge that could be used three times in an Eulerian realization of H. An analogous statement holds if the end is fixed.

Lemma 2. *Let H be a protograph that*
a) has the beginning fixed in $v \in V(H)$. Then every fat edge e that is outgoing from v is used exactly twice in any realization of H.
b) has the end fixed in $u \in V(H)$. Then every fat edge e that is incoming in v is used exactly twice in any realization of H.

Proof. We prove only a), part b) is proved analogously. The node v has no incoming fat edges and the following equation holds: $2 \cdot d_F^+(v) - 1 = d_T^-(v) - d_T^+(v)$. In the equation, the part on the right of the equal represents the total number of incoming thin edges that can be use to cross the outgoing fat edges of v. For example, if a node k has 5 incoming thin edges, 1 outgoing thin edge and one outgoing fat edge; the 4 out of 5 incoming thin edges are available to use the outgoing fat edge. Let n be the number of outgoing fat edges of v, then the equation says that we have $(n \cdot 2) - 1$ thin edges that can use the fat edges; but every fat edge must be use at least two times: $(n \cdot 2)$. Then we must start from v and use the other $(n \cdot 2) - 1$ thin edges in the outgoing fat edges that are used exactly two times. □

The next lemma follows easily from the previous discussion.

Lemma 3. *Let H be a protograph that*
a) has a node v with the following two properties: (1) $d_F^-(v) = 0$ and (2) $2 \cdot d_F^+(v) - 1 \neq d_T^-(v) - d_T^+(v)$. Then one of the outgoing fat edges of v must be used at least three times.
b) has a node u with the following two properties: (1) $d_F^+(u) = 0$ and (2) $2 \cdot d_F^-(u) - 1 \neq d_T^+(u) - d_T^-(u)$. Then one of the incoming fat edges of u must be used at least three times.

In the following theorem we prove that if the the beginning or the end is not fixed, then we have more then one Eulerian realization.

Theorem 5. *Let H be a protograph constructed over a string s such that $E_F(H) \neq 0$. Then if at least one of the following holds:*

(a) H does not have the beginning fixed, i.e. for every $v \in V(H)$: $d_F^-(v) \neq 0$ or $2 \cdot d_F^+(v) - 1 \neq d_T^-(v) - d_T^+(v)$

(b) H does not have the end fixed, i.e. for every $v \in V(H)$: $d_F^+(v) \neq 0$ or $2 \cdot d_F^-(v) - 1 \neq d_T^+(v) - d_T^-(v)$

Then, H has at least two distinct Eulerian realizations.

Proof. We prove only *(a)*, part *(b)* is proved analogously.

Let G be an Eulerian realization of $H(s)$ (such a realization exists because $H(s)$ is Eulerian). We want to show that starting from G we can construct a balanced de Bruijn subgraph $G' \neq G$ s.t $H(G) = H(G')$.

Let p be an Euler trail in G. We call the first node of p v, and its last node u.

If $v = u$, then all nodes have balance zero and, by Theorem 4, we have more than one Eulerian realization. Hence we assume $v \neq u$, which implies $bal_G(v) = 1$, $bal_G(u) = -1$ and the balance of every other node is zero.

Now we discuss two cases:

Case 1: There is at least one incoming fat edge (k, v).

Let G' be like G, except that we change $\Psi_{G'}(k, v) = \Psi_G(k, v) + 1$. G' is an Eulerian realization of H, in fact:

(i) G' is a realization of H because we increase the multiplicity $\Psi_{G'}(k, v)$, but (k, v) was a fat edge, hence $(k, v) \in E(H(G'))$ remains a fat edge;

(ii) G' contains an Euler trail that starts in k and ends in u. In fact, for the edge that we add between k and v in G', we have that $bal_{G'}(v) = 0$ and $bal_{G'}(k) = 1$, and the balance of u does not change, nor does the balance of the other nodes. If k was u then we have that $bal_{G'}(v) = bal_{G'}(k) = bal_{G'}(u) = 0$.

Hence we have found $G' \neq G$, but $H(G) = H(G') = H$.

Case 2: v does not have incoming fat edges.

Then $d_F^-(v) = 0$ and hence we must have $2 \cdot d_F^+(v) - 1 \neq d_T^-(v) - d_T^+(v)$. We can have two subcases:

1. $d_F^+(v) = 0$, i.e. no outgoing fat edges, then:

$$-1 \neq d_T^-(v) - d_T^+(v)$$

$$1 \neq d_T^+(v) - d_T^-(v)$$

In G this means $bal_G(v) \neq 1$. If $bal_G(v) \neq 1$, then it must be 0, but we saw at the beginning of this proof that if $bal_G(v) = 0$ then we have more than one realization.

2. $d_F^+(v) > 0$, i.e. there is at least one outgoing fat edges from v.

   As we saw in Lemma 3, one of the outgoing fat edges is used at least three times. Let edge $e = (v, w)$ be such an edge, i.e. $\Psi_G(e) \geq 3$.

   Let G' be like G, except for $\Psi_{G'}(e) = \Psi_G(e) - 1$. G' is an Eulerian realization of H, in fact:

   (i) G' is a realization of H because we decrease the multiplicity of $e \in E(G)$, but e was a fat edge and in $G : \Psi_G(e) > 3$, hence $e \in E(H(G'))$ remains a fat edge;

   (ii) G' contains an Euler trail that starts in w and ends in u. In fact the edge that we delete between v and w makes $bal_{G'}(v) = 0$ and $bal_{G'}(w) = 1$, and the balance of u is not changed, nor is the balance of the other nodes. If w was u then we have that $bal_{G'}(v) = bal_{G'}(w) = bal_{G'}(u) = 0$.

Hence we have found $G' \neq G$ s.t. $H(G) = H(G') = H$ in all cases.

□

3.3.2 Alternating fat cycle

Let n be an even number. We define an alternating cycle as a sequence of pairwise edge-disjoint directed paths $\alpha_1, \alpha_2, ...\alpha_n$, with $n \geq 2$, s.t.:

- the direction of the paths alternates, i.e. $end(\alpha_1) = end(\alpha_2)$, $beginning(\alpha_2) = beginning(\alpha_3)$, generalizing:

$$\begin{cases} end(\alpha_{i-1}) = end(\alpha_i) & \text{if } i \in \{2, 4, 6, 8, ..., n\} \\ beginning(\alpha_{j-1}) = beginning(\alpha_j) & \text{if } j \in \{3, 5, 7, 9, ..., n-1\} \end{cases}$$

- the first and the last path are connected, $beginning(\alpha_1) = beginning(\alpha_n)$.

In Figure 3.8 we can see some examples.

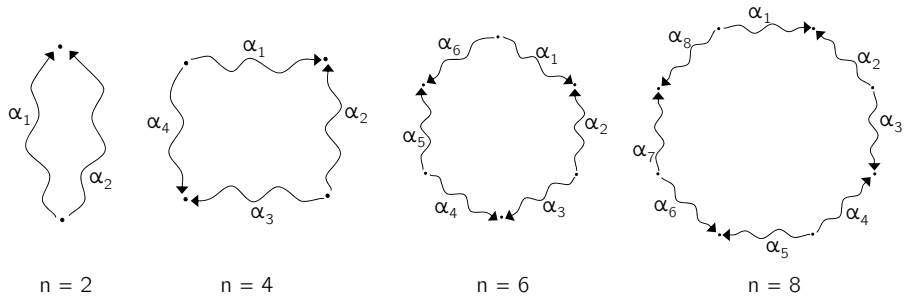


Figure 3.8: Examples of alternating cycles.

Let H be a protograph. We can show that an alternating fat cycle in H with certain properties creates more than one realization of H. An alternating fat cycle is a necessary condition to create more than one realization, but it is not sufficient. In fact you can find a protograph with an alternating fat cycle but only one realization. We call *overexpressed* an alternating fat cycle that has more than one realization.

In the next subsection we analyse the alternating fat cycle composed of two paths, because it is the simplest case and has particular properties. We refer to the alternating fat cycle composed of two paths as *multiple parallel fat paths*. After the parallel fat paths, we will analyse the general form with $n > 2$.

Parallel fat paths

Figure 3.9 shows that there are five thin edges incoming in the node x_2, and only two outgoing fat edges. These two fat edges belong to two fat paths that end in x_3. Hence, in the graph there are two fat paths that must be use five times, thus we have two realizations: one where α is used twice and β three times, or vice versa.

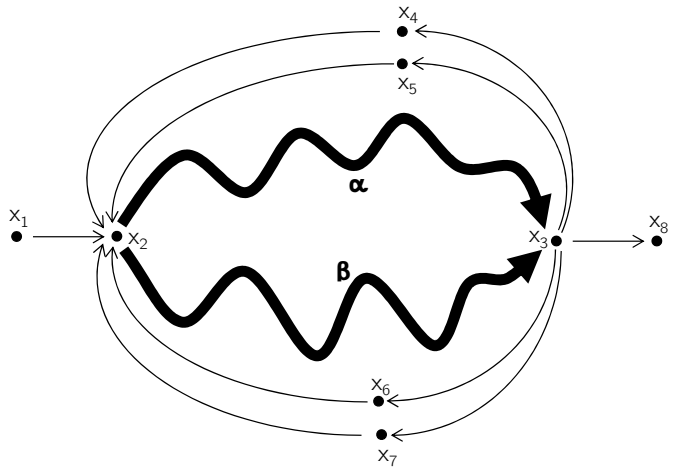


Figure 3.9: Graph with two fat paths: α and β from x_2 to x_3. We could use one path three times and the other twice, or vice versa.

Having two parallel fat paths is not a sufficient condition to have more than one realization, we can see in Figure 3.10 three examples of protographs that contain two parallel fat paths but have only one realization. If we want to define this property formally in a protograph, we have a problem. At first glance the property seems to relate to the number of thin paths that go back to x_2, coming from x_3. Hence, if we have two parallel fat paths, we should have at least five thin paths incoming in x_2. But, as we can see in

Figure 3.10 *a)* and 3.10*b)*, this is not true. Another complication is related to the choice of the first node of the two fat paths, for example we do not know which node is the first in Figure 3.10 *c)*.

It is easier to define a property on a de Bruijn graph, as shown in Figure 3.11.

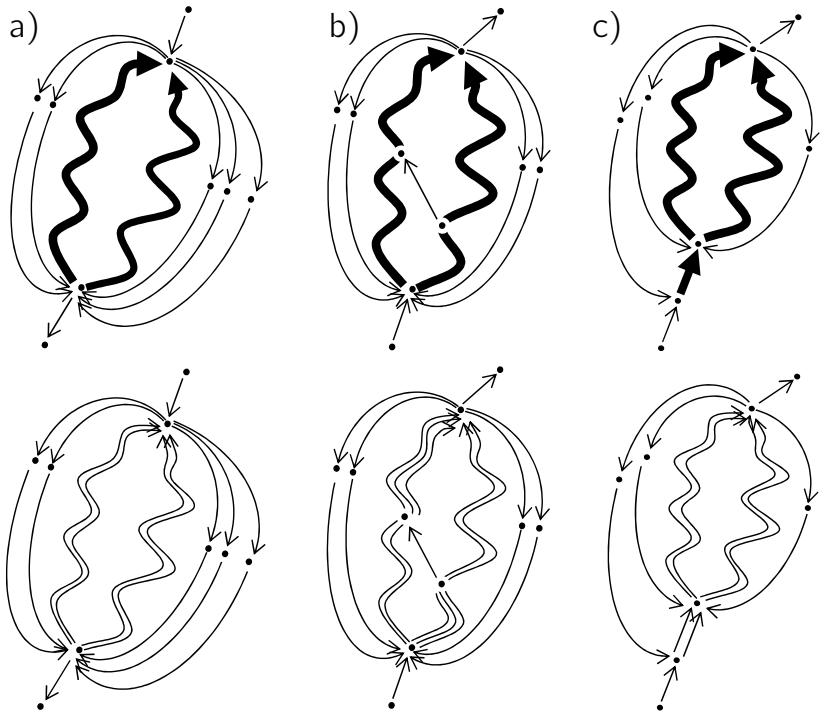


Figure 3.10: Three examples of protographs with multiple parallel fat paths. They all have only one realization, which is shown under each protograph.

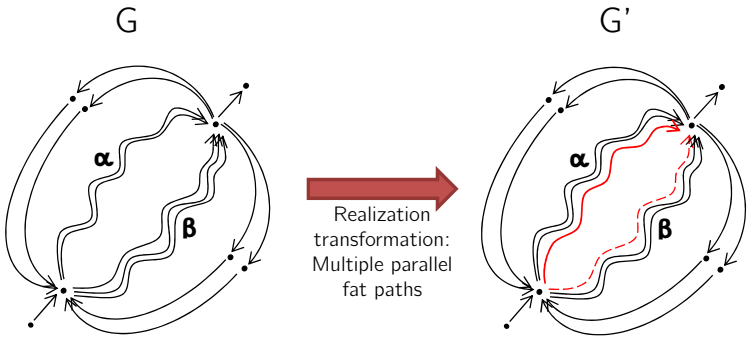


Figure 3.11: The realization transformation related to the multiple parallel fat paths. We can move one copy of path β to α.

As we can see in Figure 3.11, G must contain two parallel paths: α, where the multiplicity of the edges in the path is at least 2, and β where the multiplicity is at least 3. Now, if we take G' which increases the multiplicity of α by one and decreases the multiplicity of β by one, then also G' is an Eulerian realization of $H(G)$. In contrast, all the examples in Figure 3.10 do not have a fat path that is used at least twice and one fat path used at least three times.

Theorem 6. *Let $G = (V, E, \Psi)$ be an Eulerian realization of a protograph H with the following properties:*

1. *we have two paths in G: $\alpha = (k_1, k_2, ...k_n)$ and $\beta = (l_1, l_2, ...l_m)$*
2. *$k_1 = l_1$ and $k_n = l_m$*
3. *$\forall i \in \{1, 2, 3, ...n-1\}, \Psi((k_i, k_{i+1})) \geq 2$*
4. *$\forall j \in \{1, 2, 3, ...m-1\}, \Psi((l_j, l_{j+1})) \geq 3$*
5. *α and β are edge disjoint.*

Then we can find a balanced de Bruijn subgraph $G' \neq G$ which is an Eulerian realization of $H(G)$.

Proof. We take G' defined as:

- $V(G') = V(G)$
- $E(G') = E(G)$
- we assign the multiplicity of every edge $e \in E(G')$ as

$$\Psi_{G'}(e) = \begin{cases} \Psi_G(e) & \text{if } e \text{ does not belong to } \alpha \text{ nor } \beta \\ \Psi_G(e) + 1 & \text{if } e \text{ belongs to } \alpha \\ \Psi_G(e) - 1 & \text{if } e \text{ belongs to } \beta \end{cases}$$

Now we have to prove that : (1) $H(G) = H(G')$ and (2) that G' contains an Euler trail.

(1) The vertex set and the edge set of G and G' are the same, the only thing that can change is a fat edge that becomes a thin edge or vice versa. We change only the multiplicity of the edges in α and β which, in $H(G)$, are composed of fat edges. But in $H(G')$, too, they are fat edges, because we add 1 in the multiplicity of the edges in α; and for the edges in β we decremented by one, but their multiplicity was at least 3, so they remain fat edges.

(2) G' contains an Euler trail, since the balance of each node $v' \in V(G')$ is equal to the balance of the corresponding node $v \in V(G)$. To see this,

note that for the node $k_1 = l_1$ we increase the balance by one because it belongs to α and we decrease by one because it belongs to β, so the balance does not change. The same happened for $k_n = l_m$. For every node in the middle of α we add an incoming edge and we add an outgoing edge, so the balance does not change. For every node in the middle of β we remove an incoming edge and an outgoing edge, so the balance does not change.

□

The general case of alternating fat cycle

We will now discuss the general case of alternating fat cycle, i.e. $n \geq 4$, and when these are overexpressed. In Figure 3.12, there is an example of an alternating fat cycle composed of four paths. The protograph has two realizations, one where α_1 and α_2 are used three times and β_1 and β_2 twice, and vice versa. Note that we have changed the numbering of the paths: earlier we used only α, now we use α and β, see Figure 3.13 for examples. In this way it will be easier to define when an alternating fat cycle is overexpressed.

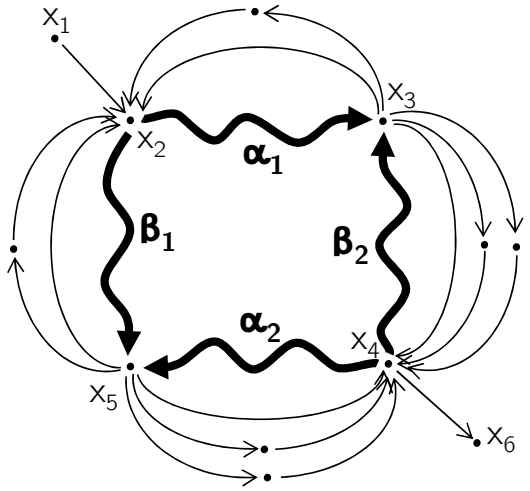


Figure 3.12: An example of a protograph containing an overexpressed alternating fat cycle.

As we did for the parallel fat paths, we define the property on a balanced de Bruijn subgraph. In the parallel fat paths case, there was only one path α and one path β. Now we have many fat paths, which we divide into two sets $A = \{\alpha_1, \alpha_2, ..., \alpha_m\}$ and $B = \{\beta_1, \beta_2, ..., \beta_m\}$. Remember that the number of paths is even, so we can divide it into two sets of the same size m.

Now, analogously to the parallel fat path case, every edge in a path in A must be used at least twice and every edge in a path in B must be used at least three times.

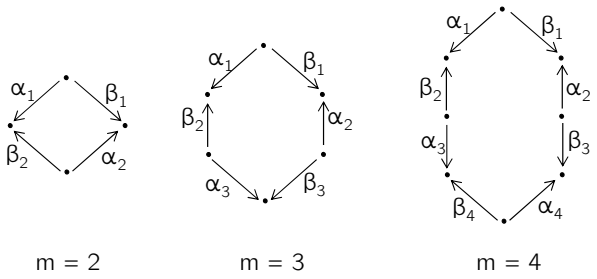


Figure 3.13: An example of how we number the fat paths in an alternating fat cycle.

Definition 9. Let $G = (V, E, \Psi)$ be a balanced de Bruijn subgraph, let m be an integer. An *alternating fat cycle* is a set $C = A \cup B$ of pairwise edge-disjoint directed paths, where A and B are two sets of path in G: $A = \{\alpha_1, \alpha_2, ..., \alpha_m\}$ and $B = \{\beta_1, \beta_2, ..., \beta_m\}$, such that:

1. the nodes that connect two paths, we denote them as *switching nodes*, have the following properties:
   (a) $end(\alpha_i) = end(\beta_{i+1})$, $1 < i < m$, i odd;
   (b) $beginning(\beta_i) = beginning(\alpha_{i+1})$, $1 < i < m$, i even;
   (c) $end(\beta_i) = end(\alpha_{i+1})$, $1 < i < m$, i odd;
   (d) $beginning(\alpha_i) = beginning(\beta_{i+1})$, $1 < i < m$, i even;
2. the first and the last switching nodes have:
   (a) $beginning(\alpha_1) = beginning(\beta_1)$;
   (b) $end(\alpha_m) = end(\beta_m)$ if m is odd,
   $beginning(\alpha_m) = beginning(\beta_m)$ if m is even;
3. the multiplicities of the edges in A and B are:
   (a) $\forall$ *edge* $e \in \alpha_i, \forall i \in \{1, 2, ...m\}, \Psi(e) \geq 2$
   (b) $\forall$ *edge* $e \in \beta_i, \forall i \in \{1, 2, ...m\}, \Psi(e) \geq 2$

Definition 10. Let $G = (V, E, \Psi)$ be a balanced de Bruijn subgraph, let m be an integer and let $C = A \cup B$ be an alternating fat cycle in G. C is *overexpressed* if

$$\forall \text{ edge } e \in \beta_i, \forall i \in \{1, 2, ...m\}, \Psi(e) \geq 3$$

Theorem 7. *Let $G = (V, E, \Psi)$ be an Eulerian realization of a protograph H. If G contains an alternating fat cycle $C = A \cup B$ which is overexpressed, then we can find a balanced de Bruijn subgraph $G' \neq G$ which is an Eulerian realization of H.*

Proof. Let m be the size of the alternating fat cycle C, i.e. $A = \{\alpha_1, \alpha_2, ..., \alpha_m\}$ and $B = \{\beta_1, \beta_2, ..., \beta_m\}$. Compare Figure 3.13 for a visualization of an alternating fat cycle. We take G' defined as:

- $V(G') = V(G)$
- $E(G') = E(G)$
- we assign the multiplicity of every edge $e \in E(G')$ as:

$$\Psi_{G'}(e) = \begin{cases} \Psi_G(e) & \text{if } e \text{ does not belong to a path in } A \text{ nor } B \\ \Psi_G(e) + 1 & \text{if } e \text{ belongs to a path in } A \\ \Psi_G(e) - 1 & \text{if } e \text{ belongs to a path in } B \end{cases}$$

Now we have to prove that: (1) $H(G) = H(G')$ and (2) G' contains an Euler trail.

(1) The vertex set and the edge set in G and G' are the same, the only thing that can change is a fat edge that becomes a thin edge or vice versa. We change only the multiplicity of the edges in the paths in A and in B, which in $H(G)$ are composed of fat edges. But also in $H(G')$ they are fat edges, because we add 1 in the multiplicity of the edges in the paths in A; and for the edges in the paths in B we decrement by one, but their multiplicity was at least 3, so they remain fat edges.

(2) G' contains an Euler trail, in fact the balance of each node $v' \in V(G')$ is equal to the balance of the corresponding node $v \in V(G)$. The switching nodes do not change their balance. In fact they have the beginning of an α and the beginning of a β as outgoing edges, or the end of a α and of a β. In both cases in the balance we add 1 and decrease 1. For the node in the middle of the paths in A we add an incoming and an outgoing edge, hence the balance does not change. The same happens for the nodes in the middle of the paths in B where we remove an incoming and an outgoing edge.

□

3.3.3 Fat cycle

In Figure 3.14 there is an example of a protograph containing a fat cycle. Recall that when we use the word cycle we mean directed cycle. We will show in this subsection that when a protograph H has a fat cycle, then H has an infinite number of Eulerian realizations. Moreover, a fat cycle is a necessary and a sufficient condition to have an infinite number of Eulerian realizations.

For the proof of the next theorem we will use the function M defined in Definition 7, Section 3.2. Let H be an Eulerian protograph, let $e \in E(H)$, recall that $M(e)$ represents an upper bound on the number of times that you can use the fat edge e in any Eulerian realization of H. Furthermore,

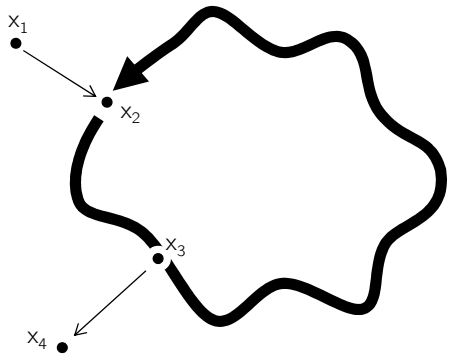


Figure 3.14: An example of graph containing a fat cycle.

it is defined only for protographs that do not contain a fat cycle, i.e. where $F(H)$ is a DAG.

Theorem 8. *Let H be an Eulerian protograph. There is an infinite number of Eulerian realizations of H if and only if H has at least one fat cycle.*

Proof. " $\Leftarrow$":
We have to show that if H contains a fat cycle then there is an infinite number of Eulerian realizations of H.

Let $G(V, E, \Psi)$ be an Eulerian realization of H (such a G exsists because H is Eulerian). Let C be a fat cycle in H. Construct a multigraph G' as follows

1. $V(G') = V(G)$,
2. $E(G') = E(G)$,
3. $\Psi_{G'}(e) = \begin{cases} \Psi_G(e) + 1, & \text{if } e \text{ lies on } C \\ \Psi_G(e), & \text{otherwise} \end{cases}$

G' is a realization of H:

1. $V(G') = V(G) = V(H)$,
2. $E(G') = E(G) = E(H)$,
3. we change only the multiplicities of the edges that lie on C. These edges are fat edges in H, and in G' we have increased the multiplicity by 1. Hence the edges that lie on C in G' are fat edges in $H(G')$.

G' is an Eulerian realization of H:

1. G' is a realization of H;

2. G' contains an Eulerian trail, since we add at each node of the cycle C an incoming edge and an outgoing edge, thus the balance of each node does not change. Since G is Eulerian, also G' is Eulerian.

We can find another graph $G'' \neq G$, G' which is an Eulerian realization of H, applying the transformation that we made to G to obtain G', to G'. In this way, we can find an infinite number of Eulerian realizations, in the presence of a fat cycle.

" $\Rightarrow$": We will show the contraposition of the claim. Thus assume that H has no fat cycle. We have to show that H has a finite number of Eulerian realizations.

In the trivial case, we have no fat edge. In this case there is only one realization.

Since now we are in the condition that $F(H)$ is a DAG, we can calculate M of each fat edge. Hence each fat edge has a maximum number of thin edges in which it can transform. In the worst case the number of Eulerian realizations could be equal to all the combinations of numbers of thin edges that we can assign to the fat edges.

$$\text{number of Eulerian realization of } H \leq \prod_{h\, fat\, edge\, in H} M(h),$$

which is a finite number.

□

3.3.4 A unique realization

We have presented and defined three properties on protographs. We have proved that when one of these property is present in a protograph H, then H has at least two Eulerian realizations.

For the following discussion we need the definitions of trees and forests:

A *tree* is an undirected graph that is connected and cycle-free. Recall that in a tree any two nodes are connected by a unique path. A directed tree is a directed graph which would be a tree if the directions on the edges were ignored. A *forest* is a disjoint union of directed trees.

A *rooted tree* is a directed tree with a special node (the *root*), where all edges are directed away from the root. In a rooted tree, an *ancestor* of a node x is any node y that belongs to the unique path between the root and x.

We conjecture the following two things (refer to Definition 8 for fixed beginning and end of a protograph):

Conjecture 1. *Let H be an Eulerian protograph. If $F(H)$ is a forest and H has the beginning and the end fixed, then we have only one Eulerian realization.*

Conjecture 2. *Let H be an Eulerian protograph. H has only one Eulerian realization if and only if the following three properties hold:*

1. *H has the beginning and the end fixed,*
2. *H does not contain fat cycles,*
3. *H does not contain overexpressed alternating fat cycles.*

The first conjecture is implied by the second one. In fact, if $F(H)$ is a forest, then it avoids both alternating cycles and cycles. This in H means that we avoid alternating fat cycles and fat cycles.

Note that in Conjecture 1 we avoid any alternating fat cycle, while in Conjecture 2 we can have alternating fat cycles that are not overexpressed.

In Figure 3.15 there is an example of $F(H)$ which is a rooted tree with the edges that are directed from the root towards the leaves. In the same figure we can see that if we add one edge between a node x in the tree and one of its ancestors then we obtain a cycle. If we add an edge between a node x and any other node except for one of its ancestors we obtain two parallel paths. If we add two edges we can create an alternating fat cycle composed of more than two paths.

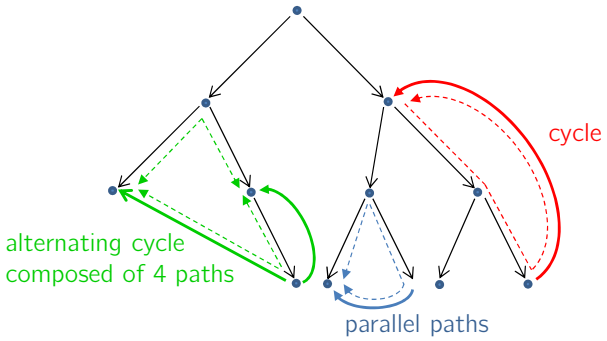


Figure 3.15: An example of a fat part of a protograph H, $F(H)$ where $F(H)$ is a tree in black. We have drawn how you can obtain a cycle (red), a parallel path (blue) and a general alternating cycles (green) by adding one or more edges.

3.4 Transformations on strings

We studied three transformation properties that can transform one balanced de Bruijn subgraph G into another balanced de Bruijn subgraph G' without changing the corresponding protograph, i.e. $H(G) = H(G')$. For every property, we can find a transformation on strings, we will present these in the following three subsections.

3.4.1 Elongation

The elongation is the transformation on strings that corresponds to the change of the beginning or the end of the Euler trail in a protograph H.

Definition 11. If a string y can be written as $y = y_1zky_2zky_3z$ for some $(q-1)$-gram z, for some strings y_1, y_2, y_3 and for a string k with length greater than 0; then the string $y' = y_1zky_2zky_3zx$, where $x \neq \lambda$ is a prefix of k, is called a *final elongation* of y.

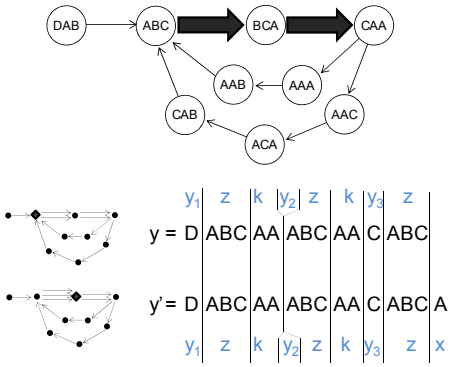


Figure 3.16: Example of a protograph where the the end of the Euler trail is not fixed. Below there are two possible realizations and the corresponding strings that we can obtain with a final elongation.

Definition 12. Let $y = zy_1kzy_2kzy_3$ be a string, or some $(q-1)$-gram z, for some strings y_1, y_2, y_3 and for a string k with length greater than 0. Then the string $y' = xzy_1kzy_2kzy_3$, where $x \neq \lambda$ is a suffix of k, is called an *initial elongation* of y.

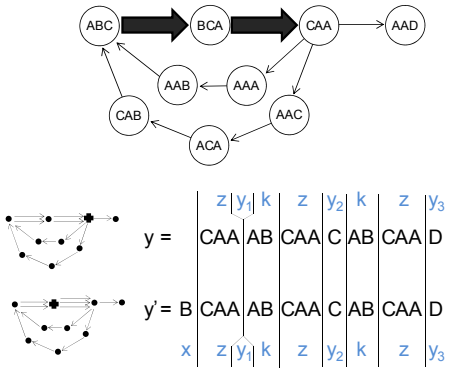


Figure 3.17: Example of a protograph where the the beginning of the Euler trail is not fixed. Below there are two possible realizations and the corresponding strings that we can obtain with an initial elongation.

We define now the inversion of an elongation: a cut.

Definition 13. If a word y can be written as $y = y_1zky_2zky_3zx$ for some $(q-1)$-gram z, for some strings y_1, y_2, y_3, for a string $k \neq \lambda$ and for a string $x \neq \lambda$ which is a prefix of k; then the word $y' = y_1zky_2zky_3zw$, where w is a prefix of x s.t. $w \neq x$, is called a *final cut* of y. The final cut is the inversion of the final elongation.

Following the protograph in Figure 3.16, a final cut could transform $y = DABCAAABCAACABCAA$ in $y' = DABCAAABCAACABCA$ where we have cut the last character, and if we apply the initial cut to y' we obtain $y'' = DABCAAABCAACABC$.

Definition 14. If a word y can be written as $y = xzy_1kzy_2kzy_3$ for some $(q-1)$-gram z, for some strings y_1, y_2, y_3, for a string $k \neq \lambda$ and for a string $x \neq \lambda$ which is a suffix of k; then the word $y' = wzy_1kzy_2kzy_3$, where w is a suffix of k s.t. $w \neq k$, is called an *initial cut* of y. The final cut is the inversion of the final elongation.

Following the protograph in Figure 3.17, an initial cut could transform $y = ABCAAABCAACABCAAD$ in $y' = BCAAABCAACABCAAD$ where we have cut the first character of y, and if we apply the final cut to y' we obtain $y'' = CAAABCAACABCAAD$.

3.4.2 Alternation

The alternation transformation on strings represents the alternating fat cycles in the protographs. We will first analyse an example with two directed parallel fat paths.

In Figure 3.18 we can see an example of a protograph containing two parallel fat paths.

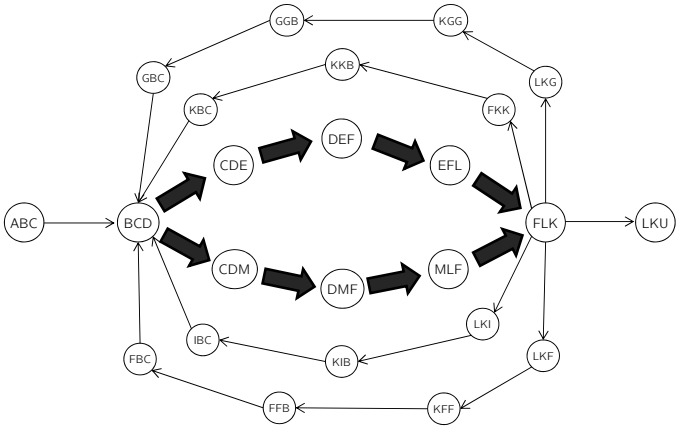


Figure 3.18: Graph with two directed fat paths (from *bcd* to *flk*) where we could use one path three times and the other twice, or vice versa.

If a word y can be written as

$$y = y_1z_1k_1z_2y_2z_1k_1z_2y_3z_1k_2z_2y_4z_1k_2z_2y_5z_1k_1z_2y_6$$

for some $(q-1)$-grams z_1 and z_2, for some strings $y_i, i \in \{1, ..., 6\}$ and for two strings $k_1 \neq \lambda$ and $k_2 \neq \lambda$; then the word

$$y' = y_1z_1k_1z_2y_2z_1k_1z_2y_3z_1k_2z_2y_4z_1k_2z_2y_5z_1k_2z_2y_6$$

is an alternation of y. Note that y is equal to y' except for the third last substring k_1 in y that is replaced by k_2 in y'.

Note that if we put $\alpha = z_1k_1z_2$ and $\beta = z_1k_2z_2$ then the alternation can be written as a transformation from

$$y = y_1\alpha y_2\alpha y_3\beta y_4\beta y_5\alpha y_6$$

to

$$y' = y_1\alpha y_2\alpha y_3\beta y_4\beta y_5\beta y_6$$

Here is an example of a string y and its alternation y', obtained from the protograph in Figure 3.18:

	y_1	z_1	k_1	z_2	y_2	z_1	k_1	z_2	y_3	z_1	k_2	z_2	y_4	z_1	k_2	z_2	y_5	z_1	k_1	z_2	y_6
y =	A	BCD	E	FLK	GG	BCD	E	FLK	K	BCD	M	FLK	I	BCD	M	FLK	FF	BCD	E	FLK	U
y' =	A	BCD	E	FLK	GG	BCD	E	FLK	K	BCD	M	FLK	I	BCD	M	FLK	FF	BCD	M	FLK	U
	y_1	z_1	k_1	z_2	y_2	z_1	k_1	z_2	y_3	z_1	k_2	z_2	y_4	z_1	k_2	z_2	y_5	z_1	k_2	z_2	y_6

Note that z_1 is the beginning of the two parallel fat paths and that z_2 corresponds to their end. Moreover, $\alpha = z_1k_1z_2$ represents one fat path and $\beta = z_1k_2z_2$ represents the other fat path.

We can analyse in the same way the alternating fat cycles composed of more than two fat paths. You can see an example in Figure 3.19.

We now define the alternation composed of four fat paths.

Let Σ be an alphabet. Let z_1, z_2, z_3, z_4 be strings in Σ^{q-1}, let k_1, k_2, k_3, k_4 be strings over Σ with a length greater than zero. Let $y_i \in \Sigma^*$, for $i \in \{1, ...11\}$. Let $\alpha_1 = z_1k_1z_2$, $\beta_1 = z_1k_2z_3$, $\beta_2 = z_4k_3z_2$ and $\alpha_2 = z_4k_4z_3$.

If a word y can be written as

$$y = y_1\beta_1y_2\beta_1y_3\alpha_1y_4\alpha_1y_5\alpha_1y_6\beta_2y_7\beta_2y_8\alpha_2y_9\alpha_2y_{10}\alpha_2y_{11},$$

then the word

$$y' = y_1\beta_1y_2\beta_1y_3\alpha_1y_4\alpha_1y_5\beta_1y_9\beta_2y_7\beta_2y_8\beta_2y_6\alpha_2y_{10}\alpha_2y_{11}$$

is an alternation of y.

We have highlighted in red the differences between the two strings. Note that in y we use the α's three times and the β's two times, in y' it is the vice-versa.

Note that we cannot simply exchange one α_1 for one β_1 and one α_2 for one β_2. We have to exchange $\alpha_1 y_6$ with $\beta_1 y_9$. If we look in the graph in Figure 3.19, the prefix of y before $\alpha_1 y_6$, which is the same of the part of y' before $\beta_1 y_9$, describes the first part of the Euler trail: $AA \to AB$, then we use two times β_1, then two times α_1 and we come back to AB. Now if we choose to use α_1 we create y, otherwise we use β_1 and we create y'. After that we go to AE, and if we are in AC (which means we used α_1), then we go through y_6, while if we are in AD (which means we used β_1), then we go through y_9. This explains why we have to exchange $\alpha_1 y_6$ and not only α_1. Following the Euler trail to the end you explain the same way the other exchange: $\alpha_2 y_9$ with $\beta_2 y_6$.

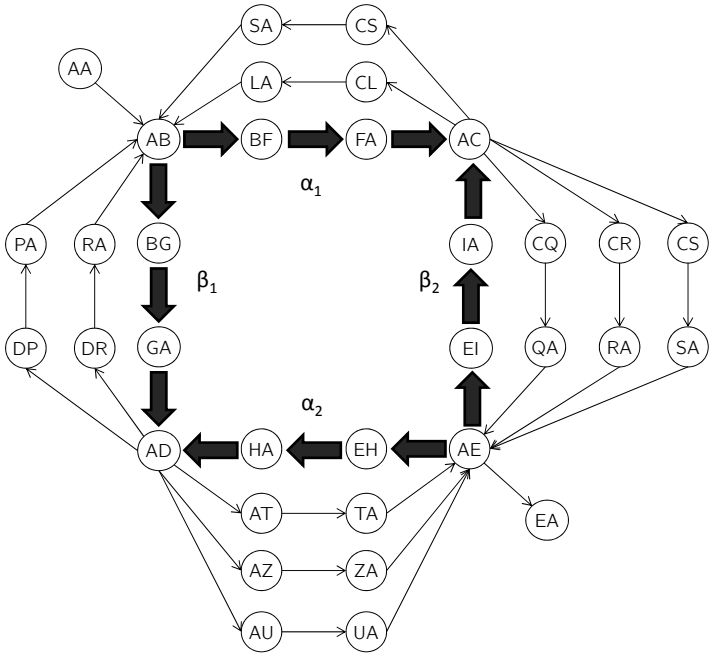


Figure 3.19: An example of a graph containing an alternating fat cycle.

Now this was a protograph with four alternating fat paths. For a general even number n, we have to define n $(q-1)$-grams for the z's and n strings for the k's. We unite it in α's and β's respecting the rules to create the alternating fat cycles that we saw in Theorem 6. We create the string y following the Euler trail and then we exchange one α_1 with one β_1, then one α_2 with one β_2, up to $\alpha_{n/2}$ and $\beta_{n/2}$. When we make these exchange we have to pay attention to exchanging not only one α for one β, but we have to exchange also some y's.

3.4.3 Expansion

This is the transformation on strings that corresponds to the fat cycles in the protographs.

Definition 15. If a string y can be written as $y = y_1zkzkzy_2$ for some $(q-1)$-gram z, for some strings y_1, y_2 and for a string $k \neq \lambda$; then the string $y' = y_1zkzkzkzy_2$ is called an *expansion* of y.

The inversion of expansion is a reduction:

Definition 16. If a string y can be written as $y = y_1zkzkzkzy_2$ for some $(q-1)$-gram z, for some strings y_1, y_2 and for a string $k \neq \lambda$; then the string $y' = y_1zkzkzy_2$ is called a *reduction* of y.

In Figure 3.20 we see an example of a protograph containing a fat cycle.

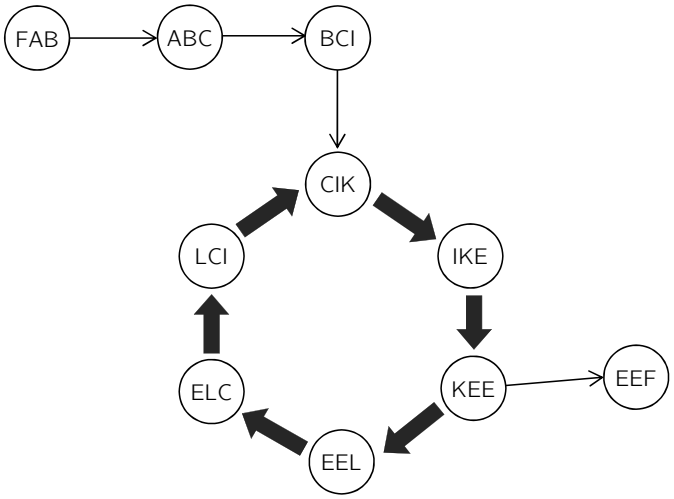


Figure 3.20: An example of protograph containing a fat cycle.

Based on Figure 3.20 we can do the following example of expansion:

	y_1	z	k	z	k	z	y_2		
y =	FAB	CIK	EEL	CIK	EEL	CIK	EEF		
y' =	FAB	CIK	EEL	CIK	EEL	CIK	EEL	CIK	EEF
	y_1	z	k	z	k	z	k	z	y_2

3.5 Protographs and linear algebra

In this section we describe an approach to the problem of finding all Eulerian realizations of a protograph, which uses some basic linear algebra. We did not follow though this approach, so we only present some initial steps.

We can see an Eulerian realization G of a protograph H as a function $f : E_F(H) \to \mathbb{N}^*$. This function assigns to every fat edge $e \in E(H)$ the multiplicity of $e \in E(G)$.

Our first attempt consist of using a system of linear equations where the number of unknowns is equal to the number of fat edges, and the number of equations is equal to the number of nodes which are endpoints of a fat edge.

Let H be a protograph where the beginning and the end are fixed. We define a system of linear equations $SE(H)$ composed of an equation for every node that is endpoint of a fat edge. The equation of a node v is given by:

$$d_T^-(v) + \sum_{\substack{e_i \text{ is an incoming} \\ \text{fat edge of } v}} e_i = d_T^+(v) + \sum_{\substack{e_i \text{ is an outgoing} \\ \text{fat edge of } v}} e_i$$

and the set of all the e_i are the unknowns.

In Figure 3.21 there is an example of a protograph and its system of linear equations.

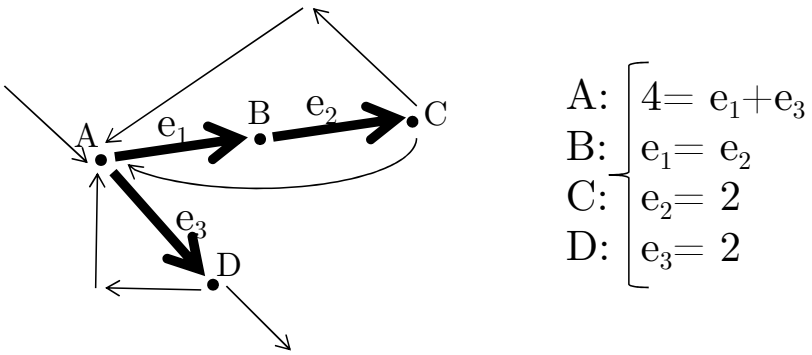


Figure 3.21: On the left an example of protograph where the beginning and end are fixed. On the right there is its system of linear equations. The solutions of the system is unique: $e_1 = e_2 = e_3 = 2$.

We originally expected that every solution to the system represents an Eulerian realization. This is not true, however, because we must insert the information that every fat edge must be used at least 2 times. Moreover it works only on protographs whose beginning and end are fixed.

In our second attempt, we defined a system of linear inequalities, based on the system of linear equations, but which solves these problems. Let H be a protograph, we want to model the maximum and the minimum number of times that we can cross through a node in any Eulerian realization of H. These two numbers are possibly distinct only for the nodes that are endpoints of a fat edge. Hence, for every node $x \in F(H)$ we have two objects:

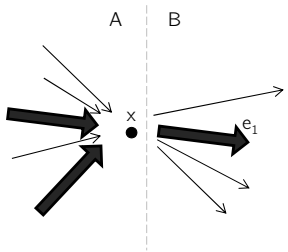
1. $Max(x)$, the maximum number of times that you can cross and exit from x in any Eulerian realization of H:

$$Max(x) = max\{d^+(x) \in V(G) \mid x \in V(F(H)), G \in G^E(H)\}$$

2. $Min(x)$, the minimum number of times that you can cross and exit from x in any Eulerian realization of H:

$$Min(x) = min\{d^+(x) \in V(G) \mid x \in V(F(H)), G \in G^E(H)\}$$

We can explain the idea as follows. Let x be a node in a protograph:



On the left we have the incoming edges and on the right we have the outgoing edges of x. $Min(x)$ measures the minimum number of times that an Eulerian trail has to get in the part B, without stopping in x; and $Max(x)$ measure the maximum number of times that you can cross x and go to B. Afterwards we create two inequalities:

- $MAX(x)$, that says that what is in B must be less than $Max(x)$, in this case would be $MAX(x) : 3 + e_1 \leq Max(x)$.
- $MIN(x)$, that tells you the minimum number of times that you have to use the edges in B, in this case $MIN(x) : 3 + e_1 \geq Min(x)$.

In Figure 3.22 there is an example of a protograph, with its system of linear equations and its system of linear inequalities. We will make clear afterwords how to construct the system of linear inequalities. Note that the protograph in (a) has the beginning and the end fixed, which correspond to the nodes a and e respectively. If they were not fixed, then we could not write the system of linear equations.

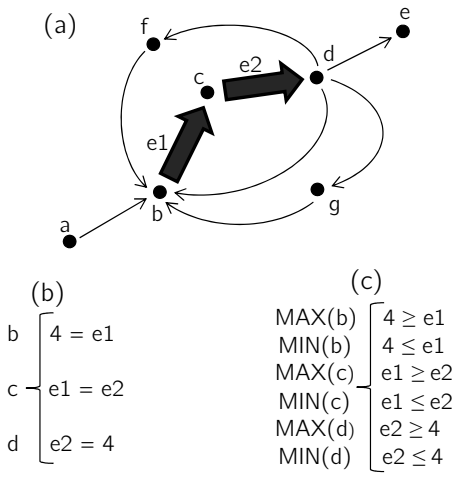


Figure 3.22: A protograph (a), its system of linear equations (b) and its system of linear inequalities (c).

We need to insert in the system an information about the *threshold* we are using. Hence we add an inequality for every fat edge. We state that every fat edge must be used at least $threshold + 1$ times. In Figure 3.23 there is an example of a protograph and the two systems of inequalities for $threshold = 1$ and $threshold = 2$. Note that the only thing that changes between 3.23(b) and 3.23(c) is the inequality corresponding to existence of $e1$.

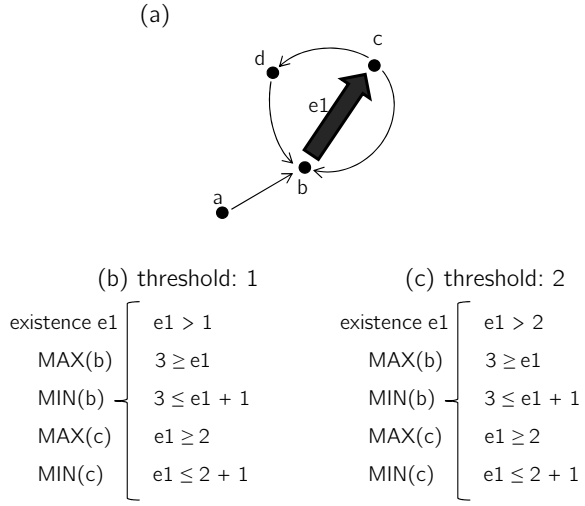


Figure 3.23: A protograph (a), and its system of linear inequalities with threshold = 1 (b) and threshold = 2 (c). The solutions of the two systems are $\{2, 3\}$ for (b) and $\{3\}$ for (c).

Now we will describe how to create a system of linear inequalities. First of all we define two expressions that will make the following definitions easier. Let H be a protograph, let $x \in V(H)$, we define the expression for the incoming edges in x as follows:

$$E_{in}(x) = d_T^-(x) + \sum_{\substack{e_i \text{ is an incoming} \\ \text{fat edge of } x}} e_i$$

Let H be a protograph, let $x \in V(H)$, we define the expression for the outgoing edges in x as follows:

$$E_{out}(x) = d_T^+(x) + \sum_{\substack{e_i \text{ is an outgoing} \\ \text{fat edge of } x}} e_i$$

Note that the e_i will be the unknowns of our inequalities.

Let H be a protograph, let F be the fat part of H, $F = F(H)$. We define the system of linear inequalities of H, denoted $S(H)$, as follows:

1. *existence inequalities*, we have $|E(F)|$ linear inequalities, one for every fat edge of H:
$$\forall\ e_i \in E_F(H) :\ e_i > threshold$$

2. *maximum inequalities*, $|V(F)|$ inequalities. For every node $x \in V(F)$ we have the inequality $MAX(x)$. We must distinguish three cases that may occur in a protograph (refer to Definition 8 for fixed beginning and end of a protograph):

   (a) if the beginning is not fixed:
   $$\forall x \in V(F) : 1 + E_{in}(x) \geq E_{out}(x)$$

   (b) if the beginning is fixed in the node ini and $ini \notin V(F)$:
   $$\forall x \in V(F) : E_{in}(x) \geq E_{out}(x)$$

   (c) if the beginning is fixed in the node ini and $ini \in V(F)$:
   $$\begin{cases} E_{in}(x) \geq E_{out}(x) & \forall x \in V(F) \setminus \{ini\} \\ 1 + E_{in}(ini) \geq E_{out}(ini) \end{cases}$$

3. *minimum inequalities*, $|V(F)|$ inequalities. For every node $x \in V(F)$ we have the inequality $MIN(x)$. We must distinguish three cases that may occur in a protograph (refer to Definition 8 for fixed beginning and end of a protograph):

(a) if the end is not fixed:

$$\forall x \in V(F) : -1 + E_{in}(x) \leq E_{out}(x)$$

(b) if the end is fixed in the node *end* and $end \notin V(F)$:

$$\forall x \in V(F) : E_{in}(x) \leq E_{out}(x)$$

(c) if the end is fixed in the node *end* and $end \in V(F)$:

$$\begin{cases} E_{in}(x) \leq E_{out}(x) & \forall x \in V(F) \setminus \{end\} \\ -1 + E_{in}(end) \leq E_{out}(end) \end{cases}$$

We now give three examples of systems of linear inequalities in Figures 3.24, 3.25 and 3.26.

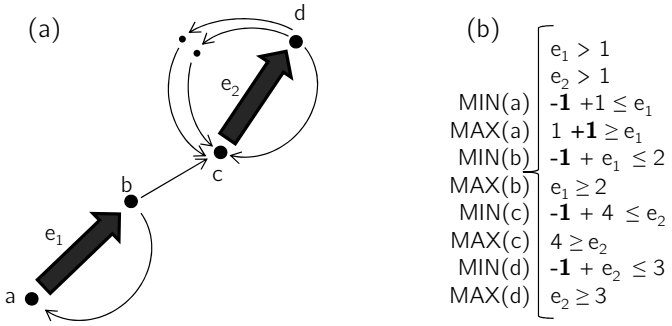


Figure 3.24: Example of a protograph (a) and its system of linear inequalities (b). The protograph has the beginning fixed in a, which is an endpoint of a fat edge. Hence in $MAX(a)$ we have a $+1$. The end is not fixed, hence in every MIN we have a -1. The solution of the system are $\{(e_1 = 2, e_1 = 3), (e_1 = 2, e_1 = 4)\}$.

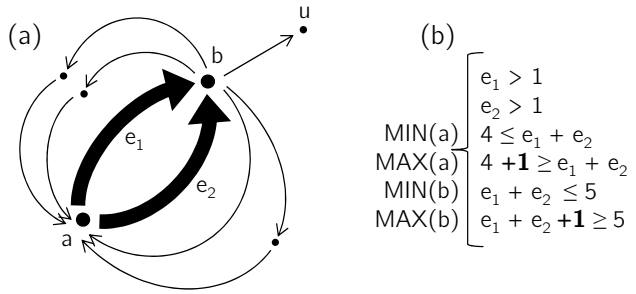


Figure 3.25: Example of a protograph (a) and its system of linear inequalities (b). In the protograph the beginning is not fixed, hence we add a $+1$ in every inequality of MAX. The end is fixed in u, which is not an endpoint of a fat edge. Hence in every MIN we do not add 1 or decrease 1.The solution of the system are $\{(e_1 = 2, e_1 = 3), (e_1 = 3, e_1 = 2)\}$.

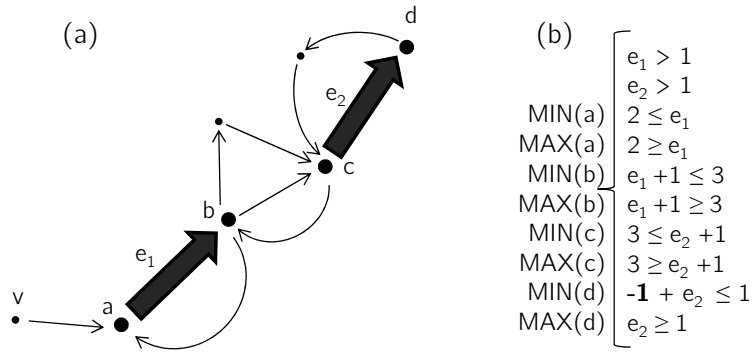


Figure 3.26: Example of a protograph (a) and its system of linear inequalities (b). The protograph has the beginning fixed in v, which is not an endpoint of a fat edge. Hence in every MAX we do not add 1 or decrease 1. The end is fixed in d, which is an endpoint of a fat edge. Therefore we put -1 in $MIN(d)$. The solution of the system in (b) is $\{(e_1 = 2, e_1 = 2)\}$.

Let H be a protograph, we conjecture that there is a one-to-one correspondence between an Eulerian realization of H and a solution of the system of linear inequalities $S(H)$.

Let A be a *solution*, defined as $A \in E_F(H) \times \mathbb{N}^*$. Let $e \in E_F(H)$ be a fat edge, we define the value that the solution A gives to e as $A(e)$, i.e. $A(e) \in \mathbb{N}^*$.

We define a *realization of H using A*, a multigraph G s.t.

1. $V(G) = V(H)$
2. $E(G) = E(H)$
3. For each edge $e \in E(G)$:

$$\Psi_G(e) = \begin{cases} 1 & \text{if } e \text{ is a thin edge} \\ A(e) & \text{if } e \text{ is a fat edge} \end{cases}$$

Let H be a protograph and let $R(H)$ be the set of solutions that we obtain from the system of linear inequalities $S(H)$. Let r be the number of solutions of $R(H)$, i.e. $r = |R(H)|$. Hence we can write $R(H) = \{R_1, ..., R_r\}$.

We conjecture that the set of all Eulerian realizations of H, $G^E(H)$, has size r, i.e. $|G^E(H)| = |R(H)|$. Moreover we conjecture that

$$G^E(H) = \{G \mid \forall R_i \in R(H), G \text{ is a realization of } H \text{ using } R_i\}$$

Computing all integer solutions of a system of linear inequalities is a NP-hard problem, but we have not investigated if our systems have some properties that may make them easier to solve.

Chapter 4

Algorithms and data structures used

Contents

In this chapter we discuss how to implement the *threshold* q-gram distance. We discuss how to create an efficient implementation and we describe the algorithms that we used to compute it. We analyse the time and space complexity, which data structures we could use and the advantages and drawbacks of each solution.

We have implemented our algorithms in C++. In this thesis we will not give any C++ code, we will write some pieces of pseudocode to clarify the implementation and the analysis. We will give some details of implementations in C++ only to discuss the limits of the implementations.

To calculate the *threshold* q-gram distance we have to evaluate the distance between the *threshold* q-gram profiles of two strings. First of all we need to store the strings and the alphabet. We discuss this in Section 4.1, together with the representation of a q-gram.

In Section 4.2 we will see the implementation of the *threshold* q-gram distance. We will analyse the space and time complexity that we obtain with the different data structures that we can use to represent the profiles.

For this chapter we will mainly follow the book *Introduction to Algorithms* [CLRS09].

We use lower case letters for integer and character variables. The first letter of the name of an array is a capital letter. The name of the functions is written in SMALL CAPITALS.

4.1 String and q-gram conversion

We represent the strings as arrays of characters. In Chapter 2, we considered the first element of a string as in position 1. For example let s be a string, the characters are numbered from 1 to $|s|$. However, the arrays in C++ are numbered from 0. Hence, let A be an array, its elements are numbered from 0 to $|A|-1$. Since we represents strings with arrays, in the pseudo code also the strings are numbered from 0.

We will see in the next subsection how to represent an alphabet and how to convert the strings from alphanumeric to numeric and vice versa.

After that we will see how we can represent a q-gram using an integer value and we will discuss how we can do this efficiently.

4.1.1 Alphabets and strings

Let Σ be the alphabet and σ its size. We want to map every character in Σ to a number using a bijective function $f : \Sigma \to \{0, 1, ..., \sigma - 1\}$. Let S be an array of characters, we convert S in an array of integers $NumS$ s.t. $NumS[i] = f(S[i])$, for $i = 0, ..., |S| - 1$.

Example 10. Let $\Sigma_{DNA} = \{A, C, T, G\}$, let $S = ATTAG$, then a natural conversion is $NumericS = 02203$, where the function f assigns to a character in S the value of its position in Σ_{DNA}.

The *threshold* q-gram distance is defined for two strings. Hence we need two arrays of characters: S_1 and S_2. To standardize the computation we convert the alphanumeric strings S_1 and S_2 in two numeric strings $NumS_1$ and $NumS_2$.

The first step is to define an order on Σ, so that we use an *ordered* alphabet Σ.

If we convert both strings in integer arrays, then the new alphabet is defined only by its length σ. In this way all calculations are independent of the alphabet and only depend on its size. Moreover, we will see that this will simplify the conversion of a q-gram to an integer.

In our implementation, we can choose from three different alphabets: *1)* the DNA alphabet $\Sigma_{DNA} = \{A, C, T, G\}$, *2)* the amino acid alphabet corresponding to the one-letter codes for the twenty amino acids: $\Sigma_{AA} = \{A, R, N, D, C, Q, E, G, H, I, L, K, M, F, P, S, T, W, Y, V\}$ and *3)* the standard alphabet of a given length n using the lexicographic order. For example, the alphabets of length three and six are $\Sigma_3 = \{A, B, C\}$ and $\Sigma_6 = \{A, B, C, D, E, F\}$, respectively.

To represent the DNA alphabet and the amino acid alphabet, we need a data structure to store them, for the alphabet in the lexicographic order we only need an integer that stores the size of the alphabet. We first discuss how to implement the algorithm for the two alphabets Σ_{DNA} and Σ_{AA} and then for the alphabet in lexicographic order. Both implementations take linear time.

We write a function CONVERT(S, Σ) which returns an integer array of the same size of S:

```
CONVERT (S, Σ)
1  create an array of integers V, whose size is |S|
2  if Σ = Σ_DNA then
3  |  V = CONVERTDNA (S)
4  |  return V
5  end
6  if Σ = Σ_AA then
7  |  V = CONVERTAA (S)
8  |  return V
9  end
10 σ = |Σ|
11 V = CONVERTLEXORD (S)
12 return V
```

Algorithm 1: Algorithm that convert an array of characters S in an array of integers V. Σ is the alphabet of S.

We will now show how to implement the three subroutines in linear time and space.

The biological alphabets Σ_{DNA} and Σ_{AA}

We will discuss only the conversion of Σ_{DNA}, the conversion for Σ_{AA} is analogous. One solution is to have an array to represent the alphabet. Let R be an array of characters that contains the letters of the alphabet and let S be a string. To convert S in $NumS$ we scan every character in S and we search it in R: for every $i \in \{1, ..., |s|\}$ we search $S[i]$ in R by scanning from left to right. Let p be a number such that $R[p] = S[i]$, then we put $NumS[i] = p$.

In Figure 4.1 there is an example for the nucleotide alphabet.

Let s be a string over Σ and let $\Sigma = \sigma$. The time complexity is $O(\sigma \cdot |s|)$ since for every position in the string we have at most σ comparison in R. The space complexity is $O(\sigma + |s|)$ and since σ is much smaller that $|s|$ we can write $O(|s|)$.

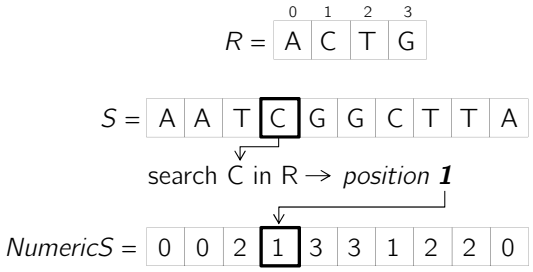


Figure 4.1: Conversion of an alphanumeric string in a numeric string using an array that contains the characters of the alphabet. For each character in $s[i] \in s$ we have to scan R to find $s[i]$.

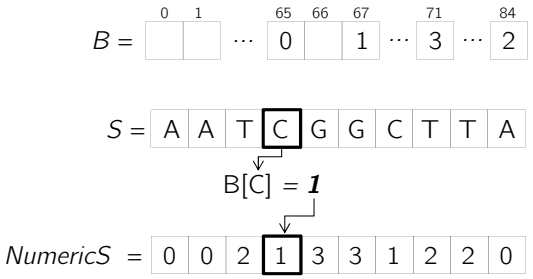


Figure 4.2: Conversion of an alphanumeric string in a numeric string using an array to represent the number that corresponds to a character in s. For every $s[i] \in s$ we use as index for B the value of the character $s[i]$ in the ASCII code.

Another solution is to exploit the ASCII code. The ASCII code encodes 128 specified characters into 7-bit binary integers. It contains numbers, upper case letters, lower case letters and control symbols. For example, the character A is encoded as binary 1000001 and decimal 65. In C++ we can use a character as a number following the ASCII code. For example, let T be an array and $c =$ 'B' be a character. When we write $T[c]$, this is the same as write $T[66]$, since the ASCII code for 'B' is 66.

Hence, for our translation from alphanumeric to numeric we can create an array B whose size is 128. To access B we will use the characters in the string that we have to convert. Therefore, if we have to convert the character 'C' in 1, then we put $B[C] = B[67] = 1$. In Figure 4.2 there is an example for the nucleotide alphabet.

In this way we have direct access to the alphabet. Instead in the first solution we had to scan the array that represent the alphabet to find the right position. In Algorithm 2 there is the pseudocode to convert a string S (represents as an array of characters) over Σ_{DNA} to an array of integer. The pseudocode for ConvertAA (S) can be obtained analogously.

```
ConvertDNA (S)
1  create an array of integers V, whose size is |S|
2  create an array of integers B, whose size is 128
3  B[65] = 0
4  B[67] = 1
5  B[71] = 3
6  B[84] = 2
7  for i = 0 to |S| do
8  |   V[i] = B[S[i]]
9  end
10 return V
```

Algorithm 2: Algorithm that convert an array of characters S in an array of integers V. S contains only characters of Σ_{DNA}.

The time complexity is $O(|s|)$ as opposed to $O(\sigma \cdot |s|)$ in the previous solution. We pay for this better performance in time with a higher space requirement. The space complexity is $O(|s|+\sigma)$, but in the previous solution $\sigma = 4$, and now $\sigma = 128$. But it is still true that $|B|$ is much smaller than $|s|$. For example the length of the shortest bacterial genome is $490,885$ nucleotides of *Nanoarchaeum equitans* and it is not comparable with 4 or 128 of the alphabet. Therefore we can safely approximate the space complexity with $O(|s|)$.

To convert a numeric string in alphanumeric, we use the array R of the first solution. Let $NumS$ be an array of integers obtained from a conversion of a string s over an alphabet R. To convert $NumS$ in an array of characters S we just have to put $S[i] = R[NumS[i]]$, for $i = 0, ..., |NumS| - 1$.

We have direct access to R. The time complexity is $O(|s|)$ because we have to scan the entire string. The space complexity is $O(|s|)$.

The alphabet in lexicographic order

For this alphabet we need only the information about the length of the alphabet. Even for this application we can use the ASCII code. But now we do not need any array. We just need to subtract the ASCII code of the first character in the alphabet.

For example, let $\Sigma_5 = \{A, B, C, D, E\}$, the integer value of C is 2, since C is in position 2 in Σ_5. When we see a C, then we have to subtract the ASCII code of A (65) from the ASCII code of C (67), which is exactly 2.

In Algorithm 3 there is the pseudocode to convert a string S (represents as an array of characters) over an alphabet in lexicographic order to an array of integer.

```
ConvertLexOrd (S)
1 create an array of integers V, whose size is |S|
2 for i = 0 to |S| do
3 |   V[i] = S[i] − 65
4 end
5 return V
```

Algorithm 3: Algorithm that convert an array of characters S in an array of integers V. S represents the string, whose alphabet is in lexicographic order.

The time complexity is $O(|s|)$ because we have one subtraction for every position in the string. And the space complexity is $O(|s|)$ to store the string, and we do not store the alphabet.

To convert an array of integer in an array of character we simply add 65 to every number and we convert it in a character using the ASCII code. The time and space complexity is $O(|s|)$.

4.1.2 q-grams

To represent a q-gram we can use an array of size q or we can use an integer to represent it. We already explained how to generate the integer code of a q-gram in Section 2.1.3, where we described the q-gram distance. Let $v = b_1 b_2 ... b_q$ be a q-gram on alphabet Σ of length σ. We remember that now the alphabet is composed of numbers. Then the *integer code* of v is

$$\tilde{v} = b_1 \cdot \sigma^{q-1} + b_2 \cdot \sigma^{q-2} + ... + b_q \cdot \sigma^0.$$

Example 11. Let $\Sigma = \{0, 1, 2\}$ and let $v = 1120$ be a 4-gram. The integer code of v is $\tilde{v} = 1 \cdot 3^3 + 1 \cdot 3^2 + 2 \cdot 3^1 + 0 \cdot 3^0 = 42$.

This representation has some limitations. The number of q-grams of a given q and a given alphabet Σ of size σ is σ^q. The classical basic types in C are *int* and *long*, which are at least 16 bits and 32 bits in size, respectively. An *unsigned long* type can represent the range $[0, 2^{32}-1]$. Using the *unsigned long* and the nucleic alphabet composed of 4 characters, we can represents all q-grams with $q \leq 16$. In fact, for $q = 16$ we have $4^{16} = 2^{32}$ possible 16-grams. For our implementation on the nucleic alphabet, we use the basic type of C++ *unsigned long long int* which uses 64 bits and hence permits to use a q up to 32. In Chapter 5 all experiments use a value of q no greater than 32.

The procedure to convert a q-gram in its integer code is represented in Algorithm 4.

```
   IntegerCode (V, σ)
1  IntV = 0
2  q = |V|
3  for i = {0, ..., q − 1} do
4  |   temp = V[i] · σ^{q−(i+1)}
5  |   IntV = IntV + temp
6  end
7  return IntV
```

Algorithm 4: Algorithm that calculates the integer code of a given q-gram V. In the input we have the array of integers V that represents the q-gram and σ that is the size of the alphabet. In the output we have an integer that represents the integer code of V.

The function IntegerCode takes q steps to execute the for-loop in line 3, hence the time complexity is $O(2 + (q \cdot 2) + 1) = O(q)$. The space complexity is $O(1)$ because we need only three variables $(IntV, q, temp)$.

The number of q-grams in a string s is $|s| - q + 1$, and since in usual applications $|s|$ is much greater than q, we can consider to have $|s|$ q-grams. To convert every q-gram of s using the function IntegerCode the time complexity is $O(|s| \cdot q)$, because for every q-gram we have $O(q)$ time.

Actually, in the previous solution we recalculate every time the integer code of a q-gram. But the integer code of a q-gram v in s is related to the previous q-gram, in fact the $(q-1)$-prefix of v is the $(q-1)$-suffix of the previous q-gram. We will investigate in Subsection 4.2.1 how to improve it.

On the other hand, if we want to obtain the array that represents the q-gram, given its integer code, then we can invert the procedure that we have used below.

Let Σ be our alphabet, σ its size and choose q. Let $\tilde{v}$ be the integer code of the q-gram v that we want to find, i.e. we know $\tilde{v}$ and not v. The number in the first position in v is equal to the integer division of $\tilde{v}$ by σ^{q-1}. Now we put $r = \tilde{v}$ % σ^{q-1}, i.e r is equal to the remainder of the integer division of $\tilde{v}$ by σ^{q-1}. The number in position 2 is equal to the integer division of r by σ^{q-2}. Now we put $r = r$ % σ^{q-1}, and we can go on till we have calculate the last number in the v.

Example 12. Let $\Sigma = \{0, 1, 2, 3, 4\}$, $\sigma = 5$ and $q = 4$. Let v be the 4-gram 1402, whose integer code is $\tilde{v} = 227$. The q-gram v is obtained from the following integer divisions:

	remainder
$227 : 5^3 = \mathbf{1}$	102
$102 : 5^2 = \mathbf{4}$	2
$2 : 5^1 = \mathbf{0}$	2
$2 : 5^0 = \mathbf{2}$	0

In Algorithm 5 we can see the pseudocode to calculate the array V, which represents the q-gram, given its integer code $intV$.

```
intToArray (intV, σ, q)
1 create an array V, whose size is q
2 for i = 1 to q − 1 do
3 |  V[i − 1] = intV / σ^(q−i)      // integer division
4 |  intV = intV % σ^(q−i)         // remainder of integer
  |                                // division
5 end
6 V[q − 1] = intV
7 return V
```

Algorithm 5: Algorithm that creates the array of integers representing the q-gram, given its integer code.

The time complexity is $O(q)$ because there are only two steps in the for-loop, which is executed q times. The space complexity is $O(1)$.

There is a smarter procedure to calculate the q-gram v starting from its integer code $\tilde{v}$: the remainder r of the division of $\tilde{v}$ by σ is equal to the last number of v. Now we put $\tilde{v}_1 = \tilde{v} \ / \ \sigma$, and we repeat the procedure on $\tilde{v}_1$.

Example 13. Let $\Sigma = \{0, 1, 2, 3, 4\}$, $\sigma = 5$ and $q = 4$. Let v be the 4-gram 1402, whose integer code is $\tilde{v} = 227$. The q-gram v is obtained from the remainder of the following integer divisions:

	remainder
$227 : 5 = 45$	**2**
$45 : 5 = 9$	**0**
$9 : 5 = 1$	**4**
$1 : 5 = 0$	**1**

Note that we obtain the number in v in reverse order, i.e the first number we calculate is $v[q]$. We can understand how it works if we look at the example on the other way round: the integer code 227 is equal to $45 \cdot 5 + 2$, and 45 is equal to $9 \cdot 5 + 0$. Hence we can write:
$227 = 45 \cdot 5 + 2$
$227 = (9 \cdot 5 + 0) \cdot 5 + 2$
$227 = ((1 \cdot 5 + 4) \cdot 5 + 0) \cdot 5 + 2$
if we carry out the calculations:
$227 = 1 \cdot 3^3 + 4 \cdot 3^2 + 0 \cdot 3^1 + 2$

If we substitute the last 2 with $2 \cdot 3^0$, which in actual fact is 2, we obtain the formula to calculate the integer code starting from the q-gram.

In Algorithm 6 we can see the pseudocode to calculate the array V given its integer code $intV$, using this technique.

```
intTOarrayFast (intV, σ, q)
1 create an array V, whose size is q
2 for i = q − 1 to 1 do
3 |   V[i] = intV % σ     // remainder of integer division
4 |   intV = intV/σ     // integer division
5 end
6 V[0] = intV
7 return V
```

Algorithm 6: Algorithm that creates the array of integers representing the q-gram, given its integer code.

The time complexity is $O(q)$ because there are only two steps in the for-loop, which is executed q times. The space complexity is $O(1)$. The time complexity of intTOarray is the same as intTOarrayFast, but in the first solution we have to deal with exponentials which requires more times to be computed.

4.2 *threshold* q-gram distance

If we want to calculate the *threshold* q-gram distance between two strings, then we need the following inputs:

- the alphabet Σ;
- $str1$ and $str2$ which are two strings over an alphabet Σ;
- the value of q;
- the value of the threshold. Even if we have discussed in the theoretical part only of *threshold* equal 1, it is easy to implement the *threshold* q-gram distance for any value of *threshold*.

We see now how to create a simple implementation of the *threshold* q-gram distance using an array to represent the *threshold* q-gram profile. We call it brute force and it is a simple implementation. We discuss in Subsection 4.2.2 the other data structure that we can use to represent a profile.

For the implementation of the brute force solution, we first define a function CreateProfile that returns an array of size σ^q which represents the *threshold* q-gram profile. In all implementations we represent the status

of a q-gram as a number in $\{0, 1, ..., threshold + 1\}$. For example if we use $threshold$ equal to 1, the possible values are 0, 1 and 2, where 2 represents ∞.

The function to create the profile takes as input the array that represents the string ($NumS$), as well as the values of q, $threshold$ and σ (the size of the alphabet). It returns an array A that contains the $threshold$ q-gram profile.

To implement it we calculate the integer code of every q-gram in $NumS$, using the function IntegerCode. Let v be an integer code of a q-gram. Now we can use v to directly access A. Since the maximum value in A is $threshold + 1$, then we update the $threshold$ q-gram profile if $A[v]$ has not exceeded the limit of $threshold + 1$. Algorithm 7 is the pseudocode of CreateProfile.

```
CreateProfile (NumS, q, threshold, σ)
1 create an array A of size σ^q and set all the elements to 0
2 for i = 0 to |NumS| − q + 1 do
3 |   Qgram = NumS[i..q + i − 1]
4 |   v = IntegerCode (Qgram,σ)
5 |   if A[v] < threshold + 1 then
6 |   |   A[v] = A[v] + 1
7 |   end
8 end
9 return A
```

Algorithm 7: Algorithm that creates the $threshold$ q-gram profile of a string $NumS$. We represent the profile with an array.

As we did before, we can consider $|NumS| - q + 1$ as $|NumS|$ when we are calculating the time complexity. Inside the for-loop we have: the lines 3 and 4 which requires $O(q)$ both, and lines $5-7$ requiring $O(1)$. The for-loop is executed $|NumS|$ times, hence the total time complexity is $O(|NumS| \cdot q)$. We do not have talk about line 1, this may take $O(1)$ if it would only allocate the memory, but we have to set all elements to 0. Hence the time complexity is $O(\sigma^q)$.

Let S be the initial string from which we obtain $NumS$. The function CreateProfile in total takes $O(|S| \cdot q + \sigma^q)$ times.

The space complexity is $O(|S| + \sigma^q)$ to store the string and the profile.

BruteForceTQD takes as input two alphanumeric strings, as two arrays S_1 and S_2. We also need information about which alphabet we are using, and the values of q and $threshold$.

In Algorithm 8 we can see the pseudocode of BruteForceTQD. First of all, we convert the two alphanumeric strings into two arrays of integers. Next, we calculate the two profiles A_1 and A_2, and finally we calculate the

distance: we scan the two profiles and if we find that two elements in the same position i have $A_1[i] \neq A_2[i]$ (i.e. the q-gram with integer code i has different status in the two strings), then we increment the distance by one.

```
   BruteForceTQD (S_1, S_2, Σ, q, threshold)
 1 σ = |Σ|
 2 NumS_1 = Convert(S_1, Σ)
 3 NumS_2 = Convert(S_2, Σ)
 4 A_1 = CreateProfile(NumS_1, q, threshold, σ)
 5 A_2 = CreateProfile(NumS_2, q, threshold, σ)
 6 distance = 0
 7 for i =0 to σ^q − 1 do
 8 |   if A_1[i] ≠ A_2[i] then
 9 |   |   distance = distance + 1
10 |   end
11 end
12 return distance
```

Algorithm 8: Algorithm that computes the *threshold* q-gram distance representing the *threshold* q-gram profiles with two arrays.

The space complexity has to consider the strings and the *threshold* q-gram profiles. The space complexity of BruteForceTQD is $O(|S_1|+|S_2|+\sigma^q)$. Let S be the longer one of the two strings S_1 and S_2, then we can write the space complexity as $O(|S| + \sigma^q)$.

To analyse the time complexity we can see that we have: $O(1)$ for line 1, $O(|S_1|)$ and $O(|S_2|)$ for lines 2and 3, $O(q \cdot |S_1|)$ and $O(q \cdot |S_2|)$ for the lines 4 and 5. To calculate the distance we need σ^q steps. In total the time complexity of BruteForceTQD is $O(q \cdot (|S_1| + |S_2|) + \sigma^q)$. If we take S as the longer string, then we have $O(q \cdot |S| + \sigma^q)$.

In the practical applications S_1, S_2 and σ are fixed, while we choose the value of q. Depending on the value of q the time complexity is more influenced by the first component or by the second one. As an example, let Σ_{DNA} be the alphabet, $\sigma = 4$, and the genome of *N. equitans* that is composed of about 500,000 nucleotides:

1. for $q = 2$ we have $|S| \cdot q$ = 1,000,000 and $\sigma^q = 16$;
2. or $q = 20$ we have $|S| \cdot q$ = 10,000,000 and $\sigma^q = 4^{20} \sim$ 1,000,000,000,000.

As we can see, σ^q grows exponentially and $|S| \cdot q$ linearly with q. Hence σ^q is predominant for high values of q, but it is not obvious that we need to use such a high value for q: it could even be that we have so many possible q-grams for $q = 20$ that it does not make sense to calculate the *threshold* q-gram distance. We will do an initial analysis about it in Chapter 5.

4.2.1 Improvements

In this chapter we analyse some possible data structures to represent the *threshold* q-gram profiles, so that we can choose the best one on the basis of the problem. Hence, the solution that uses the arrays as data structure could not be the worst solution. We analyse two improvements that we can make to the BruteForceTQD to refine the time complexity:

1. use the formula to calculate the next q-gram introduced in the Rabin-Karp algorithm [KR87] and described at page 15 where we studied the q-gram distance.

   Let $s = s_1...s_n$ be a string over an alphabet Σ of size σ. Let $v_i = s_i...s_{i-q+1}$, $1 \leq i \leq n-q+1$ be the i-th q-gram of s. Then the integer code of the next q-gram is

   $$\tilde{v}_{i+1} = (\tilde{v}_i - s_i \cdot \sigma^{q-1}) \cdot \sigma + s_{i+q}.$$

   Hence, if we know the integer code of the q-gram in position i, it takes only $O(1)$ to calculate the q-gram in position $i+1$.

   To do that we have to change CreateProfile so that: (a) we calculate the integer code of the first q-gram outside the for-loop using IntegerCode, and (b) we change lines 3 and 4 to $v = (v - NumS[i-1] \cdot \sigma^{q-1}) \cdot \sigma + numS[i+q-1]$.

   In this way the time complexity needed to calculate the profiles goes from $O(|S| \cdot q + \sigma^q)$ to $O(|S| + \sigma^q)$.

2. use an array L that represents the integer codes of q-grams that actually occur in S: L is an array of integer of size $|S| - q + 1$. In the code of CreateProfile we first calculate the integer code of every q-gram v_i, for all i in $\{0, ..., |S| - q\}$, and we insert it in L, i.e. $L[i] = v_i$. To calculate L we need $O(|S|)$.

   After that we create the array A of size σ^q and we set to 0 only the elements in L, i.e. $A[v_i] = 0$, for all v_i in L. Now this operation requires $O(|S|)$ instead of $O(\sigma^q)$. Now we can compute A in the same way we did before. With this improvement we reduce the time to create the profile from $O(|S| + \sigma^q)$ to $O(|S| + |S|) = O(|S|)$.

   Moreover we change even the time to calculate the *threshold* q-gram distance. Instead of checking if $A_1[i] \neq A_2[i]$ for each $i \in \{0, ..., \sigma^q - 1\}$, we have to do it only for $i \in L$ (line 7 of BruteForceTQD). Hence the time complexity to calculate the distance changes from $O(\sigma^q)$ to $O(|S|)$.

   Note that in L there can be more occurrences of the same q-gram. Hence, when we calculate the distance, we have to insert in the if-clause at line 8 also a command $A_1[i] = A_2[i] = 0$.

Algorithm 9 is the pseudocode that creates the *threshold* q-gram profile with the two improvements below. Note that in line 1 we create the array A as in CreateProfile, but now we do not set all the elements to 0. Algorithm 10 calculate the *threshold* q-gram distance efficiently.

In Table 4.1 we can see a summary of these techniques.

technique	time complexity		space complexity
	create profile	calculate distance	
BruteForceTQD	$\|S \cdot q\| + \sigma^q$	σ^q	$\|S\| + \sigma^q$
BruteForceTQD + efficient next q-gram(1)	$\|S\| + \sigma^q$	σ^q	$\|S\| + \sigma^q$
BruteForceTQD + array of occurrences L (2)	$\|S \cdot q\|$	$\|S\|$	$\|S\| + \sigma^q$
BruteForceTQD + (1) + (2)	$\|S\|$	$\|S\|$	$\|S\| + \sigma^q$

Table 4.1: Table that shows the time and space complexity of the computation of the *threshold* q-gram distance for various solutions that use an array to represent the *threshold* q-gram profile.

The time complexity of the last solution is the better that we can have. Because it is linear with the input. But, if we use an high value of q, then the space complexity σ^q is not practicable in real application. Moreover, the array that represents a string will be sparse: it contains many zeros and the information will be only in few elements of the array. Therefore we will describe in the next subsection some possible data structures that we can use to represent the *threshold* q-gram profile. In particular, we will decrease the space complexity trying not to increase the time complexity.

CreateProfileFast ($NumS$, q, $threshold$, σ)

create an array A of size σ^q
create an array L of size $NumS - q + 1$
$Qgram = \text{NumS}[i..q + i - 1]$
$v =$ IntegerCode ($Qgram$,σ)
$A[v] = 0$
$L[0] = v$
for $i = 1$ **to** $|NumS| - q + 1$ **do**
    $v = (v - NumS[i - 1] \cdot \sigma^{q-1}) \cdot \sigma + NumS[i + q - 1]$
    $A[v] = 0$
    $L[i] = v$
end
for $i = 0$ **to** $|NumS| - q + 1$ **do**
    $v = L[i]$
    if $A[v] < threshold + 1$ **then**
        $A[v] = A[v] + 1$
    end
end
return $[A, L]$

Algorithm 9: Algorithm that creates the *threshold* q-gram profile of a string $NumS$. We represent the profile with an array A. We return both the profile and the array with the q-grams that are actually used.

```
FastTQD (S_1, S_2, Σ, q, threshold)
σ = |Σ|
NumS_1 = Convert(S_1, Σ)
NumS_2 = Convert(S_2, Σ)
[A_1, L_1] = CreateProfileFast(NumS_1, q, threshold, σ)
[A_2, L_2] = CreateProfileFast(NumS_2, q, threshold, σ)
distance = 0
for i =0 to |L_1| do
    v = L_1[i]
    if A_1[v] ≠ A_2[v] then
        distance = distance + 1
        A_1[v] = A_2[v]
    end
end
for i =0 to |L_2| do
    v = L_2[i]
    if A_1[v] ≠ A_2[v] then
        distance = distance + 1
        A_1[v] = A_2[v]
    end
end
return distance
```

Algorithm 10: Algorithm that computes the *threshold* q-gram distance representing the *threshold* q-gram profiles with two arrays.

4.2.2 The profiles

In the array implementation we studied above the space complexity is $|S| + \sigma^q$, which is prohibitive for high values of q. In this subsection we improve the space complexity, trying not to increase the time complexity. In the previous section we created a profile for one string. Now we use data structures that represents both profiles at the same time.

We study four different data structures to represents the two profiles: *1.* list, *2.* hash table, *3.* binary search tree, and *4.* red-black tree. We implement these data structures using a node as basic structure. Every node accounts for a q-gram v, and has at least three informations: *1.* the integer code of v, *2.* the status of v in the first string, *3.* the status of v in the second string.

Using these nodes, we will store only the q-grams that really occurs in both strings. Given two strings s and t, the maximum number of distinct q-grams that appears in both strings is $(|s| - q + 1) + (|t| - q + 1)$. In the array implementation we fixed the size of the array to σ^q. Now, if we put $|S|$

the longer string, the space complexity is equal to $n\_qgrams = min(|S|, \sigma^q)$.

For the time complexity, every data structures has three times to be considered:

1. the time needed to initialize the data structure, defined as $tINI$,
2. the time needed to insert a q-gram v in the structure. Let $tINS$ the time needed to insert a q-gram. The time complexity to insert all the q-grams of s and t is $O(|S| \cdot tINS)$, if we consider S the longer between s and t.
3. the time to calculate the distance, $tDIS$.

For every of the three time we considered, we can use a function: *1.* INITIALIZE(), *2.* INSERT (v, n_string), that insert the q-gram v in the profile (n_string indicates which string we are analysing and could have value 1 or 2); and *3.* CALCULATEDISTANCE().

In Algorithm 11 there is the pseudocode of the *threshold* q-gram distance for a general data structure that represents the *threshold* q-gram profiles of the two strings.

We will now analyse the time complexity and space complexity leaving the time for the three methods unknown (we use the three variables $tINI, tINS, tDIS$ defined below). We will then analyse the space and time on some data structures, substituting the time and space of these data structures.

From lines 1 to 4 we initialize the variables and we convert the strings from alphanumeric to numeric. This group of operations requires $O(1 + |S_1| + |S_2| + tINI)$. Hence, if we use S as the longer string, we get a time $O(|S| + tINI)$.

From lines 5 to 11 we have the computation of the first profile. Which is $O(q + q + tINS) = O(q + tINS)$ for the lines $5 - 7$. And for the for-loop we have $O(|S_1| \cdot (1 + tINS))$. Hence the total time is the sum of these two: $O(q + tINS + (|S_1| \cdot tINS)) = O(|S_1| \cdot tINS)$.

For lines 12 to 18 we have analogously: $O(|S_2| \cdot tINS)$

Finally at line 19 we compute the distance, whose time is $tDIS$.

Let S be the longer string of S_1 and S_2. Then the total time to create the profiles (plus the initialization) is $O(|S| \cdot tINS + tINI)$. The total time to calculate the distance, after we have created the profiles, depends solely on CALCULATEDISTANCE. And the total time to calculate the *threshold* q-gram distance is equal to the sum of these two times: $O(tINI + |S| \cdot tINS + tDIS)$.

For the space complexity, we have to consider the space for the strings plus the space for the data structure.

```
ThresholdQgramDistance (S_1, S_2, Σ, q, threshold)
1  σ = |Σ|
2  NumS_1 = Convert(S_1, Σ)
3  NumS2 = Convert(S_2, Σ)
4  Initialize()
     // compute the first profile
5  Qgram = NumS_1[i..q + i − 1]
6  v = IntegerCode (Qgram,σ)
7  Insert(v, 1)
8  for i = 0 to |NumS_1| − q + 1 do
9  |    v = (v − NumS_1[i − 1] · σ^{q−1}) · σ + NumS_1[i + q − 1]
10 |    Insert(v, 1)
11 end
     // compute the second profile
12 Qgram = NumS_2[i..q + i − 1]
13 v = IntegerCode (Qgram,σ)
14 Insert(v, 2)
15 for i = 0 to |NumS_2| − q + 1 do
16 |    v = (v − NumS_2[i − 1] · σ^{q−1}) · σ + NumS_2[i + q − 1]
17 |    Insert(v, 2)
18 end
     // compute the distance
19 distance = CalculateDistance()
20 return distance
```

Algorithm 11: Algorithm that compute the *threshold* q-gram distance representing the *threshold* q-gram profiles as a data structure to whome we can apply three functions: Initialisation, Insert, Calculate-Distance.

We now consider four data structures and we analyse the time to create the profile and the time to calculate the distance, as well as the space complexity. Since the worst case is the same in all implementations, we will describe it now: the worst case is that all q-grams in S are different, and hence that we have to insert $|S| - q + 1$ q-grams. Moreover, when we have to evaluate the distance, we have to check $|S| - q + 1$ positions in the data structure. As before, we simplify $|S| - q + 1$ with $|S|$.

The data structures that we can use to represent the profiles are:

1. **List**. A node is composed of the integer code of the q-gram and the status of the q-gram in the two strings. Hence three integer plus a pointer to the next node: the space complexity is $O(|S|)$ because we have at most $|S|$ q-grams.

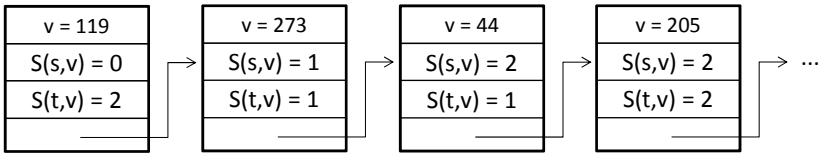


Figure 4.3: Example of list that represents the *threshold* q-gram profiles of two strings s and t.

When we use INSERT(v, n_string) we have to search in the list if v is present and then update the status, or, if it is not present, create a new node. Creating a new node requires $O(1)$. The worst case is that we scan the entire list and we do not find v and hence we create a new node. Let m be the length of the list, then INSERT takes $O(m)$. If we have to insert $|S|$ nodes then the time complexity is $O(\sum_{i=1}^{|S|} i) = O(|S|^2)$.

The time complexity to calculate the distance is $O(|S|)$. To calculate the distance we have to scan the entire list and verify if the two status are different. Since the maximum number of q-grams in S is $|S|$, then the time complexity to calculate the distance is $O(|S|)$.

2. **Hash Table**. We have an array of buckets HT, of which we can choose the size. Given a key (a q-gram), the Hash function computes an index that suggests where the entry can be found in HT. If two different keys end up in the same bucket, we have a collision. When we have a collision, we have to store all keys that arrive in the same index of HT. One possible solution is that HT is an array of lists (as defined in point 1).

   Using an Hash table we reduce the space complexity. We can choose a size for the Hash table equal to $|S|$ so that the expected number of

collisions is smaller than one. A good function could be the modulo function, with a prime number as modulus . In fact the integer code of the q-grams goes from 0 to $\sigma^q - 1$ and if we calculate the modulo a prime number we have a good division of the keys in different buckets.

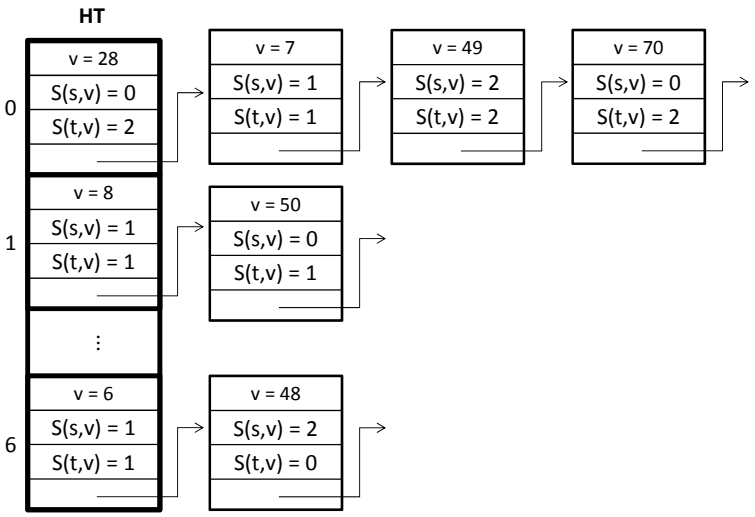


Figure 4.4: Example of hash table that represents the *threshold* q-gram profiles of two strings s and t. As a function for the hash table we use the modulo 7.

The average time complexity is $O(1)$ to insert, and hence is $O(|S|)$ to create the profile and $O(|S|)$ to calculate the distance. Hence it would have the same time complexity of the implementation with the arrays and the space complexity is $O(|S|)$.

Unfortunately in the worst case all keys go in the same position and the time complexity is the same as the List described above.

3. **Binary search tree**. We use a node as in the list, but now instead of a pointer to the next node, we have two pointers: one for the right child and one for the left child. Hence the space complexity remains $O(|S|)$ because we store only the q-grams that we see in S.

   For every node it holds that the child on the left has a value of the integer code that is lower than the node itself, and the child on the right have a value of the integer code that is higher than the node itself. Given n the number of nodes in the tree, a balanced tree has the property that the longest path from the root to a leaf has length $\log(n)$. In this case inserting a q-gram takes $O(\log(n))$. Hence the average time complexity to create the *threshold* q-gram profile is $O(|S| \cdot \log|S|)$.

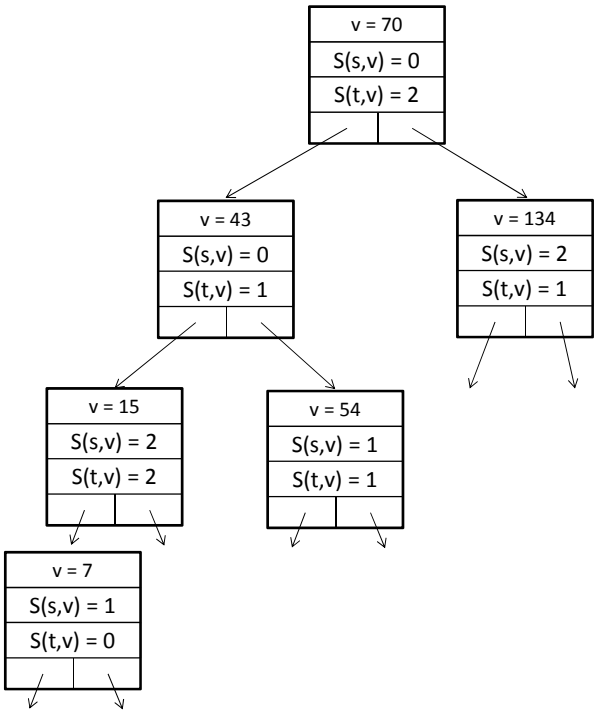


Figure 4.5: Example of binary search tree that represents the *threshold* q-gram profiles of two strings s and t.

Unfortunately, the worst case is not better than the solution with the list. In fact if every value of v that we insert is greater than the previous one, then the tree becomes a list.

The time complexity to calculate the distance is $O(|S|)$ because we have to scan every node in the tree.

4. **Red-black tree**. A red-black tree is a binary search tree that can self-balance itself. In this way it guarantees the operation of searching, and hence insertion, in $O(\log(n))$. Therefore the time complexity to create the *threshold* q-gram profile is $O(|S| \cdot \log|S|)$. To do that every node have a variable *colour*, that could be red or black, and there are some rules that prevent the tree from being unbalanced. Every time we insert a node we check if the rules are still true, if it is not the case (the tree is not balanced) we rearrange the tree so that the rules returns true. The operations to rearrange the tree takes $O(\log(n))$. For details on the implementation see *Introduction to Algorithms* [CLRS09].

   The space complexity remains $O(|S|)$ because we have only add one variable (the *colour*) to every node. The time complexity to evaluate the distance is the same as in the Binary search tree.

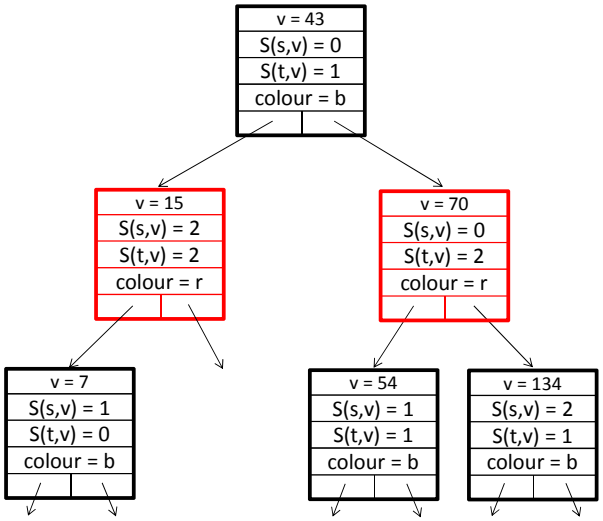


Figure 4.6: Example of red-black tree that represents the *threshold* q-gram profiles of two strings s and t. Note that the nodes are the same as in Figure 4.6, but here the tree is balanced.

In Table 4.2 we have a summary of the time and space complexity to calculate the *threshold* q-gram distance for the data structures we analysed.

data structure to represent the profile	time complexity		space complexity
	create profile	calculate distance	
Array	$\|S\|$	$\|S\|$	$\|S\| + \sigma^q$
List	$\|S\|^2$	$\|S\|$	$\|S\|$
Hash Table	$\|S\|^2$	$\|S\|$	$\|S\|$
Binary search tree	$\|S\|^2$	$\|S\|$	$\|S\|$
Red-Black Tree	$\|S\| \log(\|S\|)$	$\|S\|$	$\|S\|$

Table 4.2: Table that represents the time and space complexity of the calculation of the *threshold* q-gram distance for some data structures that we can use to represents *threshold* q-gram profile. The time complexity represents the worst case.

If $|S| > \sigma^q$ then the best solution is the one with arrays. If $\sigma^q > |S|$ then the best solutions are (1) the Hash table that has an average time complexity equal to the solution with arrays and space complexity that depends linearly on S, and (2) the Red-black tree that guarantees $O(|S| \cdot \log |S|)$ to compute the profile.

Chapter 5

Experiments

Contents

In this chapter we will apply the *threshold* q-gram distance to real genomes. We first describe how to deal with errors in the sequencing process of a genome that leads to an unknown character, which is usually represented with an N in the sequence. Afterwards we analyse three possible lines of research: *1)* the analysis of the repeats and hapax that two strings have in common or that are unique to only one of the two strings. We apply the *threshold* q-gram distance to two strains of *E. coli*. *2)* use random sequences to extract the relevant information from biological sequences. *3)* We evaluate how the *threshold* q-gram distance can be useful to understand the distance between genomes, some of its issues and how to choose q. We compute the distance between five bacterial genomes with different degrees of relatedness.

5.1 The N's in the genomes

When we work with real biological sequences, which are obtained from experiments, we have to deal with errors. Moreover the genomes that we use nowadays are obtained as a mean of the genomes of many individuals. In the Human Genome Project [Con01], the DNA from several anonymous donors

was used. Together, they represent around 65-fold coverage (redundant sampling) of the genome. In the same project, more than 1.4 million single nucleotide polymorphisms (SNPs) were identified. A Single Nucleotide Polymorphism is a DNA sequence variation occurring in more than 1% of a population in which a single nucleotide in the genome differs between members of the species.

When a SNP is present, in the genome sequence we see an N instead of the classical characters A, T, C or G. An N in position i in the genome could also mean an error in the sequencing of the i-th nucleotide. In the standard IUB/IUPAC nucleic acid codes N means any character $A/T/C/G$.

When we compute the *threshold* q-gram distance, the presence of the N's in the sequences is problematic. We can consider an N in one of three ways:

1. We can create a new alphabet $\Sigma = \{A, T, C, G, N\}$ and consider N as a different nucleotide. But this is not correct, since every occurrence of N is in actual fact one of the four nucleotides.
2. We can ignore the N's and create a new sequence without them. For example, let $s = ATCGANNNAC$, we can calculate the *threshold* q-gram profile using $s' = ATCGAAC$. But this solution is not correct. In fact in s' we create some q-grams that do not occur in s. In the example, in s' the last 3-gram is AAC which is not present in s.
3. We can consider a q-gram with an N as any of the q-grams that we can obtain replacing the N with any nucleotide character. For example, let $v = ATCCN$ be a 5-gram, when we have to update the *threshold* q-gram profile we do it as if we had seen four 5-grams: $ATCCA, ATCCT, ATCCC, ATCCG$. This solutions is not appropriate because we create more q-grams. Moreover, the number of q-grams that we have to add increases exponentially with the number of N's in the q-gram. If the q-gram is composed only of N's then we have to update the status of every possible q-gram.

To solve this problem, we first study how many N's are present and how they are distributed in the Human genome.

We have analysed the 24 chromosomes of the Human genome: 22 autosomes plus X and Y. We use the assembly *GRCh38* provided by the *Genome Reference Consortium* and released on 17 December 2013. In this assembly, the human chromosome sequences have the RefSeq accessions *NC_000001 - NC_000024*, followed by a point and a number that represent the version of each chromosome. We downloaded the sequences from NCBI between the 28 and 29 October 2014.

In total we find 150,630,620 N's, which constitute 4.9% of the entire genome. In Table 5.1 we list the number of N's for each chromosome and the percentage of N's with respect to the length of the chromosomes. In Figure 5.1 you can see two graphs that represent the values inside the table.

chromosome	accession number	length	number of N's	percentage of N's
1	NC_000001.11	248,956,422	18,475,408	7.42
2	NC_000002.12	242,193,529	1,645,292	0.68
3	NC_000003.12	198,295,559	195,417	0.10
4	NC_000004.12	190,214,555	461,888	0.24
5	NC_000005.10	181,538,259	272,881	0.15
6	NC_000006.12	170,805,979	727,456	0.43
7	NC_000007.14	159,345,973	375,838	0.24
8	NC_000008.11	145,138,636	370,500	0.26
9	NC_000009.12	138,394,717	16,604,164	12.00
10	NC_000010.11	133,797,422	534,424	0.40
11	NC_000011.10	135,086,622	552,880	0.41
12	NC_000012.12	133,275,309	137,490	0.10
13	NC_000013.11	114,364,328	16,381,200	14.32
14	NC_000014.9	107,043,718	16,475,569	15.39
15	NC_000015.10	101,991,189	17,349,864	17.01
16	NC_000016.10	90,338,345	8,532,401	9.44
17	NC_000017.11	83,257,441	337,225	0.41
18	NC_000018.10	80,373,285	283,680	0.35
19	NC_000019.10	58,617,616	176,858	0.30
20	NC_000020.11	64,444,167	499,910	0.78
21	NC_000021.9	46,709,983	6,621,361	14.18
22	NC_000022.11	50,818,468	11,658,686	22.94
X	NC_000023.11	156,040,895	1,147,861	0.74
Y	NC_000024.10	57,227,415	30,812,367	53.84

Table 5.1: Table that represents the frequency of the character 'N' in the 24 chromosomes of the Human genome.

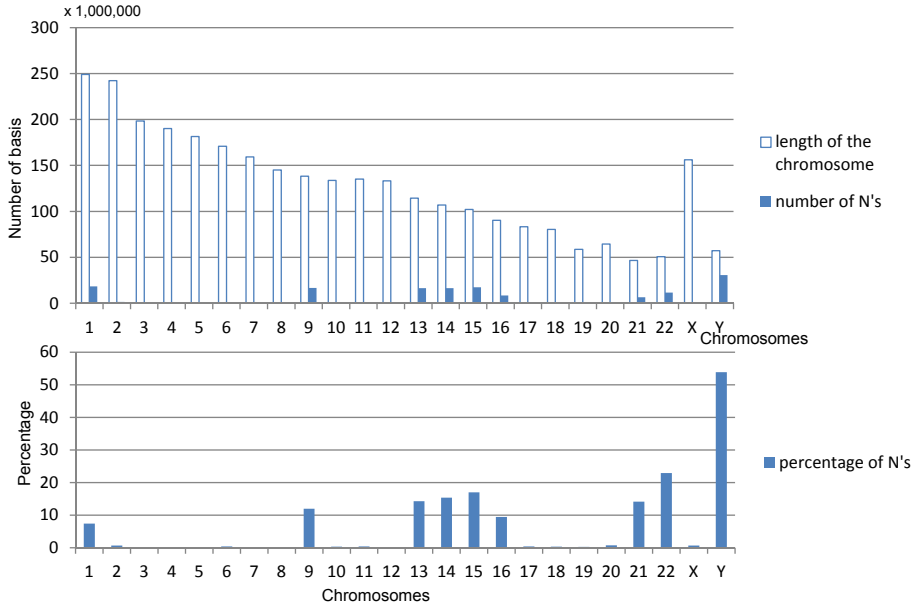


Figure 5.1: Representation of Table 5.1. In top histogram we show the length of the chromosomes and the number of N's. In bottom histogram we represent the percentage of N's with respect to the length of the chromosomes.

5.1.1 Distance between consecutive N's

If the 150 million N's were equally distributed over the 3 billion nucleotides of the Human genome, then we would have one N every 20 nucleotides. Since every N can change up to q q-grams (remember the q-gram lemma) then the application of the *threshold* q-gram distance would not be practicable.

We can calculate the distance between consecutive N's on the Human genome. Let $\Sigma = \{A, T, C, G, N\}$ be an alphabet, let $s \in \Sigma^m$ be a string, let $p(s) = \{p_i \mid \forall i \in \{1, ..., n\}, s[p_i] = N\}$ be the set of positions of the N's in s. Let $n = |p|$, then the *multiset of distances of consecutive N's* is defined as $D(s) = \{p_{i+1} - p_i \mid \forall i \in \{1, ..., n-1\}\}$.

Example 14. Let $s = aattcNNNNNccNcNct$: there are 7 N, i.e. $n = 7$. The set of the position of the N's in s is $p(s) = \{6, 7, 8, 9, 10, 13, 15\}$. The multiset of distances of consecutive N's of s is $D(s) = \{1, 1, 1, 1, 3, 2\}$.

If we calculate the multiset of distances of consecutive N's on the 24 chromosomes of the Human genome we obtain that 150,629,717 N's are at distance 1. This means that $99,9994\%$ of all the N's are linked together to form many N's islands. If we define an N *island* a substring composed only of N's whose length is at least 100, then we have 786 islands in the Human genome. At the beginning of this subsection we calculated that if the N's are equally distributed, then we would have an N every 20 nucleotides. Now we have 786 N islands: this means that, if the N islands are equally distributed, then we expect to find an N island every 3.8 Mb.

We can visualize the distribution of the N's on the chromosomes using the following strategy. Let s be a chromosome string over $\Sigma = \{A, T, C, G, N\}$. We divide each chromosome in 300 equal-length segments: let $l = |s|/300$ be the length of a segment. For each segment Seg_i we can calculate the percentage of N's as the number of N's inside Seg_i divided by l. Finally we plot the 300 values of the percentage of N's for each segment.

In Figure 5.2 we see few examples. The second plot, relative to chromosome 6, is the most representative. Chromosome 1 and 9 (data on Chromosome 9 not shown) have a large central segment that is full of N's and it is in correspondence of the centromeres of the two chromosomes. Chromosomes 13, 14, 15 and 21, 22 are all acrocentric, which means that the short arm of the chromosome is almost invisible in the karyotype and hence the centromere is near the beginning of the sequence. And in this position we have a high content of N. All other chromosomes, except for Y, are similar to chromosome 6. In other words, the N's are usually concentrated in the centromere, as we saw for chromosome 1, or in the telomeres as we can see at the beginning of chromosome Y or at end of chromosome 6. Centromeres and telomeres are known to be highly repetitive parts in the chromosomes, and they have a structural function in the maintenance of the chromosomes and in the replication process.

Chromosome Y is different from all other chromosomes. In fact, more than half of the characters in its sequence are N's. This is because the Y chromosome is one of the fastest evolving parts of the human genome [Con01] and the large segment full of N's is highly variable.

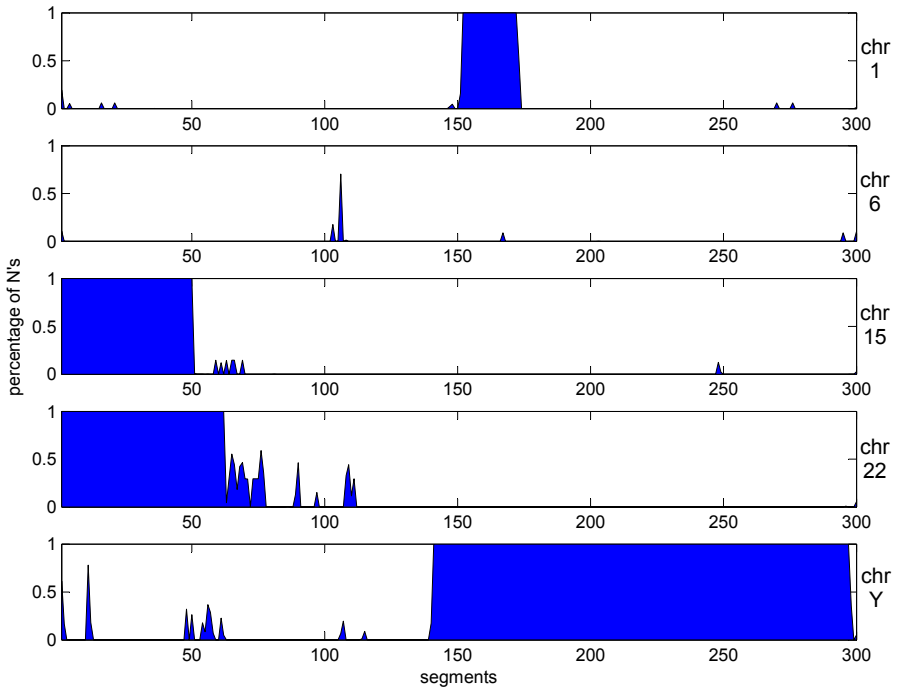


Figure 5.2: Examples of distributions of the N's in five representative chromosomes. On the right of every plot the chromosome that represents the plot is indicated.

5.1.2 Solution

Now that we have understood that the N's are grouped together in islands, we can decide how to deal with the N's. The best solution is to find the segments of the genomes that do not contain any N's and compute the *threshold* q-gram distance separately.

Let $\Sigma = \{A, T, C, G, N\}$, let $s \in \Sigma^n$ be a string. Choose a number $threshold_N \in \mathbb{N}^*$ s.t. $threshold_N \leq n$, that represent the minimum length of a segment without N's. Let $t = s[i, j]$, for $1 \leq i \leq j \leq n$, be a substring of s. We say that $t = s[i, j]$ is a *maximal N-free substring* of s if:

1. $|t| \geq threshold_N$;
2. if $i \geq 2$, then $S[i-1] = N$, and
   if $j \leq n-1$, then $S[j+1] = N$.
3. $s[k] \neq N$, for $k \in \{i, ..., j\}$.

Example 15. Let $\Sigma = \{a, t, c, g, N\}$ be an alphabet where we use the lower case for the nucleotides only to visualize better the N's and their positions. Let $s = Ngaaaatcg NNagNNN$ and let $threshold_N = 3$.
The substring $t = s[2, 9] = gaaaatcg$ is a maximal N-free substring of s.
The substring $t' = s[3, 9] = aaaatcg$ is not a maximal N-free substring of s because $s[2] \neq N$, i.e. it violates point 2.
The substring $t'' = s[1, 9] = Ngaaaatcg$ is not a maximal N-free substring of s because $s[1] = N$, i.e. it violates point 3.
The substring $t''' = s[12, 13] = ag$ is not a maximal N-free substring of s because $|t| = 2$, that is not greater or equal to $threshold_N$, i.e. it violates point 1.

Example 16. Let $\Sigma = \{a, t, c, g, N\}$ and let $s = atcNNacgtt$ and let $threshold_N = 3$. The length of the string is 10.
The substring $t = s[1, 3] = atc$ is a maximal N-free substring of s, even if it does not have an N on the left. In fact the point that said $s[i - 1] = N$, said also that this holds only for $i \geq 2$ and in this case $i = 1$.
The same way, the substring $t = s[6, 10] = acgtt$ is a maximal N-free substring of s, even if it does not have an N on the right.

To find all maximal N-free substrings of s we can simply start from the beginning of s and use two counters $cont_1$ and $cont_2$. The first one indicates the first character of the maximal N-free substring t with an N on the left. The second counter indicates the position of the last character in t with an N on the right. At the beginning $cont_1 = 1$ and $cont_2 = 1$. We scan the entire string s from $s[1]$ to $s[n]$, when we are at $s[i]$ we have to two possibilities:

1. $s[i] \neq N$, then we set $cont_2 = i$;
2. $s[i] = N$, then we check if $cont_2 - cont_1 \geq threshold_N$, then $s[cont_1, cont_2]$ is a maximal N-free substring of s. In any case we set $cont_1 = i + 1$.

Since $cont_2$ is always equal to $i - 1$ when we need to know its value, then we can implement the algorithm without it. In Algorithm 12 you can see the pseudocode that creates the multiset R that contains all the maximal N-free substrings of a string s, represented as an array of character S.

We remember that the string s is represented as an array, hence is numberd from 0 to $s - 1$.

Note that at line 7 the value $(i - 1)$ represents $cont_2$. From lines 12 to 14 we treat the possibility that a maximum N-free substring can finish in the end of the string and does not have an N on the right.

```
CalculateNfreeSubstring (S, threshold_N)
1  cont_1 = 0
2  R = ∅
3  for i = 0 to |S| − 1 do
4  |  if S[i] = N then
5  |  |  if i − cont_1 ≥ threshold_N then
6  |  |  |  R = R ∪ {S[cont_1, (i − 1)]}
7  |  |  end
8  |  |  cont_1 = i + 1
9  |  end
10 end
11 if |S| − cont_1 ≥ threshold_N then
12 |  R = R ∪ {S[cont_1, |s| − 1 ]}
13 end
14 return R
```

Algorithm 12: Algorithm that calculates all maximal N-free substrings of a string. The inputs are the string S, represented as an array of characters, and $threshold_N$. The output is a multiset R that contains all maximal N-free substring of S.

The time complexity is linear. In the worst case we have a string composed only of N's. Hence in the for-loop at line 4, for each i we have three steps (two for evaluating the if-clauses and one for updating $cont_1$). Hence it is $O((|s| - 1) \cdot 3)$ and the if-clause at line 12 is $O(1)$. In total it is $O((|s| - 1) \cdot 3 + 1) = O(|s|)$.

The space complexity is $O(|s|)$ because in the worst case the string s does not contain any N and the size of R is $|s|$.

Let s be a string and choose $threshold_N$. We denote by $R(s) = \{R_1, ..., R_k\}$ the multiset of all maximal N-free substrings of s.

Now we have to redefine the *threshold* q-gram distance to deal with many strings. We redefine the status of a q-gram for a set of all maximal N-free substrings.

Let Σ be an alphabet, let x in Σ^q be a q-gram, let s be a string in Σ^*. Let $R = R(s)$ be a set of k strings, $R = \{R_1, ..., R_k\}$. For $x \in \Sigma^q$, let $S(R, x)$ denote the *status* of x in R, which is defined as:

$$S(R,x) = \begin{cases} 0 & \text{if } x \text{ does not occur in any } R_i \in R \\ 1 & \text{if } x \text{ occurs in exactly one } R_i \in R \text{ and } S(R_i, x) = 1 \\ \infty & \text{if } x \text{ occurs in exactly one } R_i \in R \text{ and } S(R_i, x) = \infty \\ \infty & \text{if } x \text{ occurs in at least two strings } R_i, R_j, i \neq j \end{cases}$$

Let s and t be two strings, choose q and $threshold_N$. Now we can calculate the *threshold* q-gram distance as

$$TQD_q(s,t) = \sum_{v \in \Sigma^q} |\{v \mid S(R(s),v) \neq S(R(t),v)\}|.$$

For the implementation of the TQD we can compute $R(s)$ and $R(t)$ using Algorithm 12 in time $O(|s| + |t|)$. Then we use Algorithm 11, which calculates the *threshold* q-gram distance and we change the call of the function INSERT(v, n_string): we put $n\_string = 1$ if R_i comes from $R(s)$, we put $n\_string = 2$ if R_i comes from $R(t)$. The time complexity and the space complexity of THRESHOLDQGRAMDISTANCE remain unchanged.

This solution is useful even if we want to compare two genomes composed of many chromosomes. For example, if we have to compute the *threshold* q-gram distance between the entire Human genome and the entire genome of the chimpanzee, then we can treat the chromosomes as the segments R_i and calculate the distance with the new definition.

5.2 First case study: two strains of *E. coli*

For a first experiment we decided to compare the genomes of two strains of *Escherichia coli*. A strain is a subgroup within the species that has unique characteristics which distinguish it from other strains. *Escherichia coli* is a Gram-negative bacterium of the genus Escherichia that is commonly found in the lower intestine of virtually all mammals.

The strain *E. coli* K12 is a debilitated strain which does not normally colonize the human intestine. It survives poorly in the environment and it is widely used as model organism also due to the fact that it does not have any adverse effects. We decided to use the genome of *E. coli* K12 because it is one of the most extensively studied micro-organisms.

For the second genome we used *Escherichia coli* O157:H7, which is an anenterohemorrhagic serotype of the *Escherichia coli*. In contrast with *E. coli* K12, the strain O157 is virulent and causes serious illness or death. The virulence is imputed to a toxin called the Shiga Toxin that causes premature destruction of the red blood cells. Strains of *E. coli* that express the Shiga Toxin gained this ability due to infection with a prophage, and the strain *E. coli* K-12 may become infected and produce the toxin after incubation with *E. coli* O157 [ONM+84].

We decided to use two organism of the same species, but different strains so that they are similar enough to give an appreciable result. Moreover we can try to interpret the results with respect to the differences which are known between the two genomes.

The RefSeq of the two sequences are NC_000913.3 for *E. coli* K12 and BA000007.2 for *E. coli* O157. Sequences were downloaded from NCBI on 20

October 2014. The length of the genome of *E. coli* K12 is 4.64 Mb, while *E. coli* O157 has 5.64 Mb. Hence *E. coli* O157 has approximately 1 million more nucleotides than *E. coli* K12. Both sequences does not contains any N's.

For all this section we will refer to first string as *E. coli* K12, and to second string for *E. coli* O157.

First of all we can calculate the *threshold* q-gram distance between the two strings. In Table 5.2 we can see the values of the distances for $q \in \{1, ..., 32\}$ in the second column. In the third column we have the number of q-grams that have a status different from zero in both sequences, i.e. the number of q-grams which occur in at least one of the two sequences. We recall that the status of a q-gram v in s is: 0, if v does not appear in s; 1, if it appears only once and 2 if v occurs at least twice in s. In the fourth column of Table 5.2 we have the total number of q-grams, i.e. $|\Sigma|^q$.

q	threshold q-gram distance	q-grams with pair status ≠ (0,0)	4^q	q	threshold q-gram distance	q-grams with pair status ≠ (0,0)	4^q
1	0	4	4	17	3,415,406	6,618,034	171.8E+8
2	0	16	16	18	3,500,252	6,666,892	687.2E+8
3	0	64	64	19	3,580,089	6,710,440	274.9E+9
4	0	256	256	20	3,658,666	6,752,447	110.0E+10
5	0	1,024	1,024	21	3,736,118	6,793,509	439.8E+10
6	0	4,096	4,096	22	3,810,119	6,832,544	175.9E+11
7	5	16,384	16,384	23	3,883,352	6,871,043	703.7E+11
8	394	65,489	65,536	24	3,955,786	6,909,030	281.5E+12
9	9,288	259,309	262,144	25	4,025,164	6,945,346	112.6E+13
10	188,176	954,919	1,048,576	26	4,093,778	6,981,205	450.4E+13
11	1,059,775	2,607,392	4,194,304	27	4,161,676	7,016,639	180.1E+14
12	2,197,970	4,553,659	16,777,216	28	4,226,917	7,050,645	720.6E+14
13	2,820,618	5,754,591	67,108,864	29	4,291,545	7,084,297	288.2E+15
14	3,085,031	6,265,359	268.4E+6	30	4,355,598	7,117,616	115.3E+16
15	3,225,803	6,466,121	107.4E+7	31	4,417,295	7,149,693	461.2E+16
16	3,326,330	6,558,755	429.5E+7	32	4,478,396	7,181,432	184.5E+17

Table 5.2: Table of the *threshold* q-gram distance between *E. coli* K-12 and *E. coli* O-157 for various values of q. In the third column we have the number of q-gram that have a status different from zero in both sequences. In the fourth column we have the maximum number of q-gram for a given q, that is 4^q.

In Figure 5.3 we represent the values of Table 5.2, and in Figure 5.4 we represent only the first three columns of the table. Using these graphs we can better understand and explain the table.

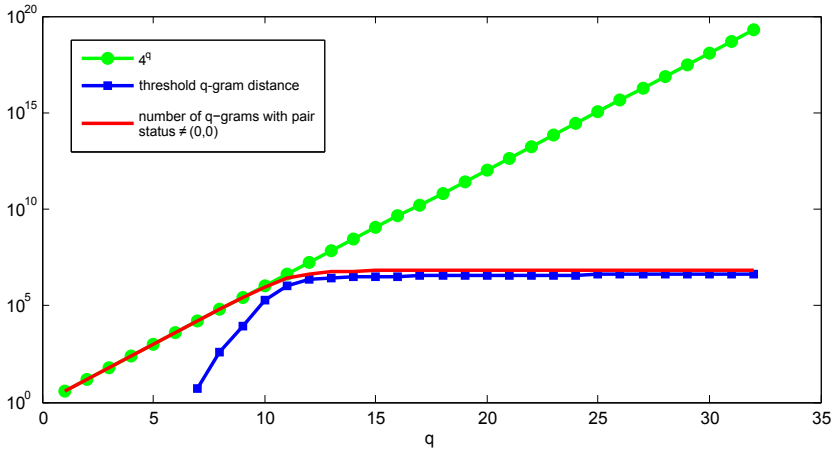


Figure 5.3: Plot that represents the *threshold* q-gram distance between *E. coli* K-12 and *E. coli* O-157 for various values of q. In abscissa we have the value of q and in ordinate we have the numbers up to 10^{20} in logarithmic scale. In blue the *threshold* q-gram distance, in red the number of q-grams that have a status different from zero in both sequences and in green 4^q. The values corresponds to Table 5.2. Note that the red and blue lines only appear close due to the logarithmic scale, see Figure 5.4.

The number of q-grams that have a status different from zero (third column in the table) is the number of q-grams that occur at least once in at least one of the two strings. The fourth column represents all possible q-grams. Since the alphabet is $\Sigma = \{A, C, T, G\}$, there are $|\Sigma|^q = 4^q$ possible q-grams.

As we can see in Figure 5.3, virtually all q-grams with $q \leq 10$ occurs in at least one of the two strings. In fact, the red line and the green line coincide. For $q > 10$ both the distance and the number of q-grams that appear in at least one of the two strings have a linear trend, see Figure 5.4. Obviously it is not possible that all q-grams appear in the strings for any value of q. In other words, 4^q grows exponentially, and the number of q-grams that compare in at lest one of the two sequences follows this increase for $q \leq 10$.

The number of q-grams in a string s, counted with multiplicities, is $|s| - q + 1$, hence we have approximately 10,000,000 q-grams for s and t (in fact the length of the two genomes together is around 10 Mb). If all these q-grams are different, then we would have 10 million q-grams with pair status different from $(0, 0)$. It is interesting that for $q = 10$ almost all possible q-grams appear in the two strings (954,919 out of 1,048,576), while for $q = 11$ only half of all q-grams appear in the two strings (2,607,392 out of 4,194,304), even if there are 10 million q-grams. In other words, the fact that not all the 11-grams appears in the two strings is not a limitation due

to the length of the two genomes. It is likely that $q = 11$ has a meaning, or contains some sort of information. However, we do not have an explanation for this.

To analyse the differences between the first two columns of Table 5.2 it is more practical to look at Figure 5.4.

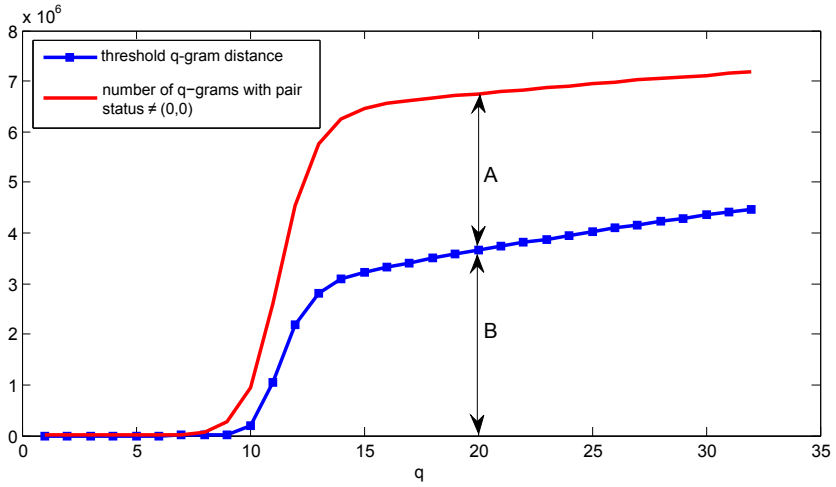


Figure 5.4: Representation of the *threshold* q-gram distance between *E. coli* K-12 and *E. coli* O-157 in blue. We plot in red the number of q-grams with pair status different from $(0, 0)$. We have highlighted two numbers: A is the number of 20-grams with the same status in the two strings, and B is the number of 20-grams with different status.

Obviously the number of q-grams that have a pair status different from $(0, 0)$ is always greater or equal to the *threshold* q-gram distance. In fact all the possible pair status different from $(0, 0)$ are:

$$\{(0,1),(0,2),(1,0),(1,1),(1,2),(2,0),(2,1),(2,2)\}$$

The *threshold* q-gram distance is equal to the number of q-grams with different status, hence the number of q-grams with pair status equal to one of the following:

$$\{(0,1),(0,2),(1,0),(1,2),(2,0),(2,1)\}$$

In Figure 5.4, we plotted two numbers for $q = 20$, they represent:

- B, is the *threshold* q-gram distance, hence the number of q-grams with different status;
- A, the number of q-grams with same status (but different from zero), i.e. $(1, 1)$ or $(2, 2)$.

In order to analyse Figure 5.4 we look at the distribution of the pair status in A and B. For example in Figure 5.4, how many of the q-grams in A have a pair status $(1, 1)$, and how many q-grams have a pair status $(2, 2)$. We will study this in the following subsection.

5.2.1 Analysis of the pair status

While we are computing the *threshold* q-gram distance, it is easy to store the total number of q-grams that have a certain pair status, for every possible pair status. We need 9 counters (one for each pair status) and, after we have created the profiles, while we are scanning the profiles to calculate the distance, we update the counters.

We define a number that represents the number of q-grams with a certain pair status. Let s and t be two strings over an alphabet Σ of size σ and choose $q \leq min(|s|, |t|)$. Let $a, b \in \{0, 1, 2\}$ and let $v = (a, b)$ be a pair status. We define the counter of v in s and t as

$$C_{s,t,q}(v) = |\{x \in \Sigma^q \mid S(s, x) = a, S(t, x) = b\}|$$

When s and t are clear from the context, we just write $C_q(v)$. We define $A_{s,t}(q) = C_{s,t,q}(1, 1) + C_{s,t,q}(2, 2)$, and $B_{s,t} = \sum_{i \neq j} C_{s,t,q}(i, j) = TQD_q(s, t)$. When s and t are clear from the context, we just write $A(q)$ and $B(q)$.

In Table 5.3, we see the counters of each pair status, calculated on *E. coli* K-12 and *E. coli* O-157 for $1 \leq q \leq 32$.

To analyse the table, we proceed in two directions:

- first we understand which are the pair status that compose $A(q)$ and $B(q)$ for $1 \leq q \leq 32$;
- second we study how the hapax and repeats are distributed.

In Figure 5.5 we can see two graphs. The graph on top represents the counter of the pair status with same status: the number of q-grams that have status 1 in both strings and the number of q-grams that have status 2 in both strings. We can see that the main component of $A(q)$ is the pair $(2, 2)$ for $q < 10$; for $q = 11$ we have $C_q(1, 1) \approx C_q(2, 2)$ and for $q > 11$ the main component is the pair $(1, 1)$. $C_q(2, 2)$ is dominant for lower q's because the chance to see a short word at least two times in a very long string, like a genome, is high. In our case, for example, if we choose $q = 9$, then there are $4^9 = 262,144$ possible q-grams. The genomes we are using have a length of about $5,000,000$; hence the probability the a q-gram appears more than once is high (in mean 20 occurrences).

In Figure 5.5, in the bottom graph, we can see the decomposition of the *threshold* q-gram distance in its components. The main components are the counters of the pair status $(0, 1)$ and $(1, 0)$. Hence the *threshold* q-gram distance is mainly due to q-grams that are hapax in one string and do not appear in the other, or vice versa.

q	$C_q(0,1)$	$C_q(0,2)$	$C_q(1,0)$	$C_q(1,1)$	$C_q(1,2)$	$C_q(2,0)$	$C_q(2,1)$	$C_q(2,2)$
1	0	0	0	0	0	0	0	4
2	0	0	0	0	0	0	0	16
3	0	0	0	0	0	0	0	64
4	0	0	0	0	0	0	0	256
5	0	0	0	0	0	0	0	1,024
6	0	0	0	0	0	0	0	4,096
7	0	1	0	0	3	0	1	16,379
8	43	86	25	70	164	12	64	65,025
9	1,647	1,135	887	2,047	3,799	305	1,515	247,974
10	37,437	19,367	22,328	67,000	71,427	5,313	32,304	699,743
11	324,953	85,578	197,672	608,304	278,842	25,066	147,664	939,313
12	948,715	125,984	590,948	1,771,596	313,338	35,613	183,372	584,093
13	1,470,143	114,086	935,086	2,696,647	168,923	27,377	105,003	237,326
14	1,734,389	98,746	1,121,053	3,090,828	68,091	19,341	43,411	89,500
15	1,857,192	91,277	1,215,376	3,199,790	28,179	15,715	18,064	40,528
16	1,926,197	88,015	1,273,264	3,206,853	14,822	14,561	9,471	25,572
17	1,978,005	86,296	1,319,610	3,181,799	10,578	14,240	6,677	20,829
18	2,024,094	85,172	1,361,997	3,147,559	9,060	14,203	5,726	19,081
19	2,066,390	84,305	1,401,358	3,112,088	8,395	14,264	5,377	18,263
20	2,107,653	83,529	1,439,953	3,076,050	7,970	14,345	5,216	17,731
21	2,148,160	82,807	1,477,967	3,040,076	7,641	14,430	5,113	17,315
22	2,186,763	82,131	1,514,312	3,005,464	7,369	14,501	5,043	16,961
23	2,224,889	81,494	1,550,292	2,971,055	7,126	14,574	4,977	16,636
24	2,262,568	80,869	1,585,904	2,936,899	6,896	14,643	4,906	16,345
25	2,298,612	80,270	1,620,032	2,904,103	6,702	14,711	4,837	16,079
26	2,334,235	79,691	1,653,783	2,871,606	6,512	14,784	4,773	15,821
27	2,369,454	79,134	1,687,187	2,839,389	6,333	14,855	4,713	15,574
28	2,403,266	78,601	1,719,302	2,808,388	6,168	14,920	4,660	15,340
29	2,436,730	78,080	1,751,130	2,777,639	6,016	14,980	4,609	15,113
30	2,469,869	77,575	1,782,672	2,747,132	5,877	15,039	4,566	14,886
31	2,501,752	77,102	1,813,095	2,717,723	5,749	15,084	4,513	14,675
32	2,533,317	76,644	1,843,225	2,688,570	5,620	15,129	4,461	14,466

Table 5.3: Table containing the number of q-grams with a certain pair status. The analysis is performed on *E. coli* K-12 and *E. coli* O-157 for various values of q.

The q-grams that are hapax in only one of the two strings $((1,0),(0,1))$ and the q-grams that are hapax in both strings, pair status equal to $(1,1)$, have a similar shape and they have higher values. On the other hand, q-grams that are a repeat in at least one of the two strings are less present (their counters have a lower value), and they all have a Gaussian-like shape. We analyse these two groups separately in Figure 5.6.

In Figure 5.6, the top graph represents the counters of the pair status without repeats, while the bottom graph represents the counters of the pair status with at least one repeat.

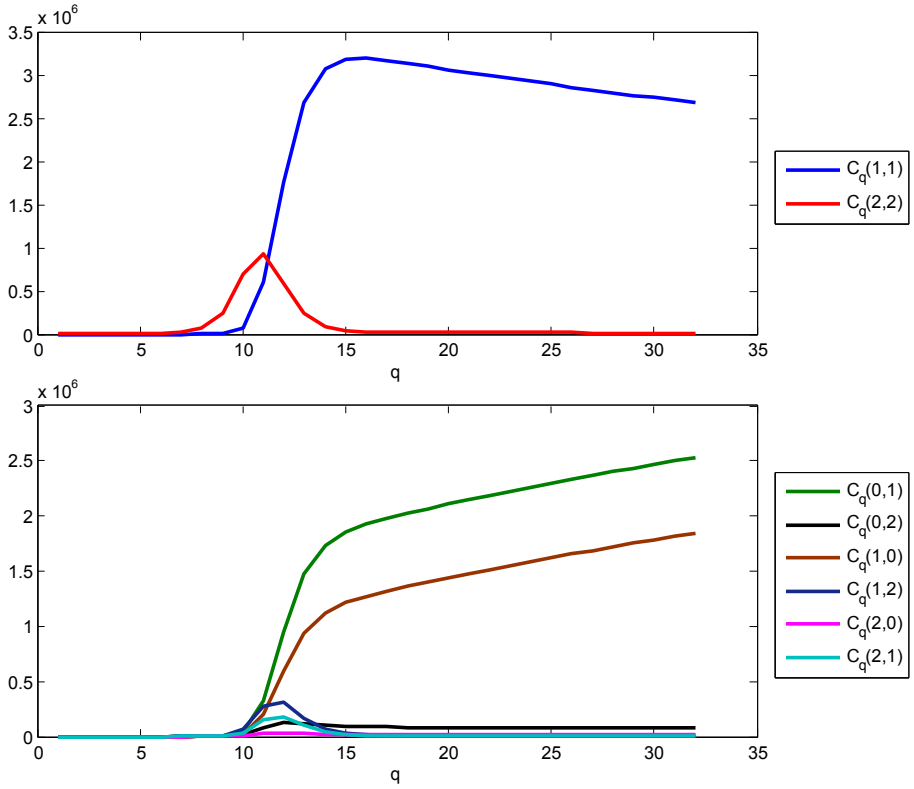


Figure 5.5: Representation of the counter of the pair status for the *threshold* q-gram distance between *E. coli* K-12 and *E. coli* O-157. The top graph represents the counter of the pair status with same status. The bottom graph represents the counter of the pair status with different status, hence it is a decomposition of the *threshold* q-gram distance.

We see that the component of $(0,1)$ is greater than the component of $(1,0)$. We believe this is because the second string is longer than the first one. Moreover, the counter of $(1,1)$ reaches a plateau at $q = 15$ and then it starts to decrease. Let v be q-gram that is an hapax in both strings (v has pair status $(1,1)$). Its prefix of length $(q-1)$ is a $(q-1)$-gram that could have pair status:

- $(2,2), (1,2), (2,1)$ and when we add a character it becomes a q-gram with status $(1,1)$, or
- $(1,1)$, and we add the same character in position q to form a q-gram with status $(1,1)$.

Hence till $q = 15$ we have an increase of the q-grams with pair status $(1,1)$. After 15 the status related to the repeats have a counter around 0 (except for $(0,2)$), and the status $(0,1)$ and $(1,0)$ increase, while the pair status

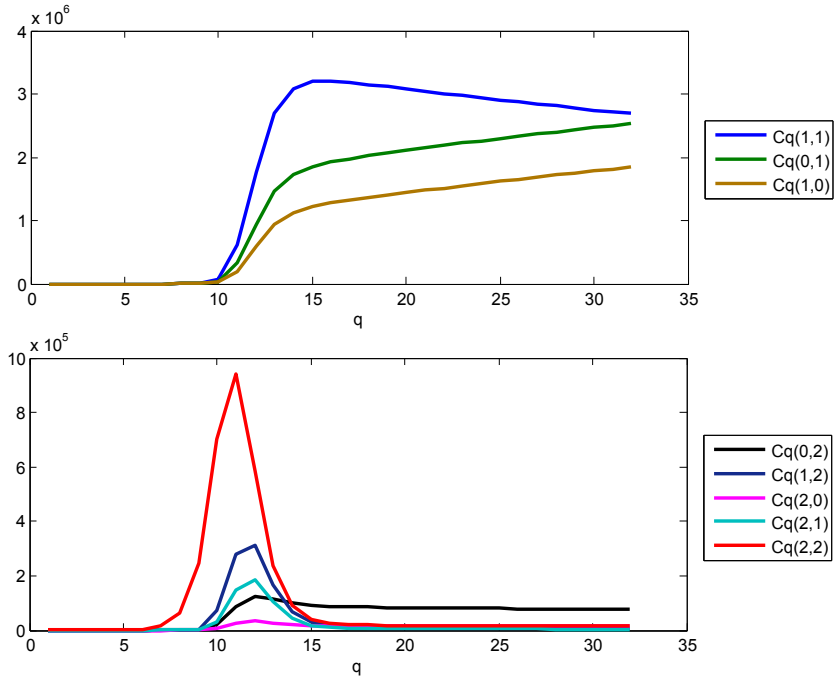


Figure 5.6: Representation of the counter of the pair status for the *threshold* q-gram distance between *E. coli* K-12 and *E. coli* O-157. The top graph represents the counter of the pair status without repeats, while the bottom graph represents the counter of the pair status with at least one repeat.

$(1,1)$ decreases. This means that a q-gram with pair status $(1,1)$ becomes two $(q+1)$-grams with pair status $(0,1)$ and $(1,0)$. This mean that for $10 \leq q \leq 15$ we have the maximum number of hapax in common. After that length, the strings start to diverge: the counters of the status $(0,1)$ and $(1,0)$ increase, and, on the other hand, the q-grams in common decrease.

All the counter of pair status related to a repeat (Figure 5.6 bottom) goes to zero after $q = 15$, except for $(0,2)$. For $q < 12$ the highest value is for the pair status $(2,2)$. This means that for $q < 12$ the sequences have more repeats in common than alone. We can see that $C_q(1,2)$ is higher than $C_q(2,1)$, and $C_q(0,2)$ is higher than $C_q(2,0)$. And we expect that because the second string is longer than the first one. The value of $C_q(2,0)$ does not go to zero as the others.

The difference between $C_q(0,1)$ and $C_q(1,0)$ and the values of $C_q(0,2)$ for $q > 15$ seems to represent the difference of length between the two genomes. The two shapes of $C_q(0,1)$ and $C_q(1,0)$ are similar, while $C_q(0,2)$ is different from the others. Looking at the counter of the pair status $(0,2)$, we think that *E. coli* O-157 might have a large repeat segment that is repeated, and is not present in *E. coli* K12.

5.3 Random genomes

In this section, we only introduce the possibility to investigate a large field of study: the study of the statistical significance of properties observed in sequences. For example, in the previous section we were studying the counter of the pair status, but we did not know if some values were high because of a biological significance or it was due to chance. Generally, an analysis of this type includes:

1. the creation of random genomes representing the 'background noise' from which we can differentiate the relevant biological information;

2. use of statistical methods to analyse the differences between the properties obtained from the genomes and the properties calculated on many random genomes.

With respect to the second point, we will simply observe the differences between the properties of the original genomes and the properties calculated as an average on a moderate number of random genomes.

There are many techniques to creates random genomes, and the best one to use depends on the investigation and on the properties that one is studying. We can create a random sequence that has the same properties of the original sequence (a shuffled sequence) or we can create a random sequence which will have similar properties in expectation [Fit83]. To this last type belongs tools such as *HMMER* in the GCG package [1], which use hidden Markov models to generate sequences. Another example is the tool *RandSeq* in ExPASy that allows one to create random protein sequences using a Bernoulli process.

A standard technique to shuffle sequences, used also in [CFM12], is to repeatedly choose two random positions and swap the respective characters. In this way we create random sequences with the same frequencies of A, T, C and G of the original sequence. However, a simple shuffling of the letters changes the frequencies of dinucleotides, trinucleotides, of q-grams in general. Often one wants to keep these properties in the random genomes. Hence, there are more complex techniques, such as *Shufflet* [JAGM08], that generates random sequences with the same q-gram profile, usually for low values of q.

Since in this thesis we are working exactly with the q-grams and their occurrences we cannot use these algorithms. Instead we use a shuffling similar to the standard one, but instead of one character we swap segments in the sequence.

Let s be a string of length n. Our algorithm of segment shuffling works as follows: we first fix a number seg_len which represents the maximum

[1] http://www.biology.wustl.edu/gcg/hmmanalysis.html

length of the segments that we will swap; and n_iter which is the number of iterations. For every iteration $i \leq n\_iter$ we randomly choose three positive integers: $l < seg\_len$, the length of segment; $p_1 < n-l+1$ and $p_2 < n-l+1$, which represent the two positions in the string s that we will exchange. Afterwards we swap the segment $s[p_1, p_1 + l]$ with $s[p_2, p_2 + l]$.

Example 17. Let $s = abcdefghil$, $n = 10$ and we choose $seg\_len = 5$. For the first iteration, the random numbers are: $l = 2$, $p_1 = 4$, $p_2 = 1$. Then the string becomes $s = decabfghil$, where ab and de have been swapped .

Note that if $|p_1 - p_2| < l$ then we have an overlap of the segments, and when we swap them, then we have a duplication of a suffix or a prefix of one of the two segments. To minimize this error we chose seg_len very smaller with respect to n so that the chance to have an overlap is near zero.

5.3.1 Comparison of random and real in *E. coli*

We did an introductory study comparing the data obtained from *E. coli* K12 and E. coli O157 with the data obtained from random genomes.

We did twenty times the following experiment: take the sequences of *E. coli* K12 and E. coli O157, let's call them A and B. We did 1 million swap of segments, whose maximum length was 100, in A, yielding A'. We did 1 million swap of segments, whose maximum length was 100, in B, yielding B'. Afterwards, we compute the *threshold* q-gram distance of A' and B'.

q	threshold q-gram distance	SD	q-grams with pair status ≠ (0,0)	SD	q	threshold q-gram distance	SD	q-grams with pair status ≠ (0,0)	SD
1	0.00	0.00	4.00	0.00	17	10,135,117.80	123.72	10,136,743.85	82.61
2	0.00	0.00	16.00	0.00	18	10,138,821.35	51.16	10,139,226.05	36.81
3	0.00	0.00	64.00	0.00	19	10,139,750.05	23.08	10,139,849.80	17.19
4	0.00	0.00	256.00	0.00	20	10,139,984.00	10.81	10,140,007.70	8.84
5	0.00	0.00	1,024.00	0.00	21	10,140,039.35	9.78	10,140,045.10	8.61
6	0.00	0.00	4,096.00	0.00	22	10,140,052.40	7.00	10,140,053.40	6.83
7	0.00	0.00	16,384.00	0.00	23	10,140,053.90	6.22	10,140,054.10	6.13
8	0.00	0.00	65,536.00	0.00	24	10,140,053.00	5.43	10,140,053.05	5.44
9	7.65	2.23	262,144.00	0.00	25	10,140,051.75	4.96	10,140,051.75	4.96
10	131,680.15	383.81	1,048,117.30	22.83	26	10,140,050.10	4.49	10,140,050.10	4.49
11	2,720,880.45	1,103.27	3,751,462.95	576.23	27	10,140,048.40	4.03	10,140,048.40	4.03
12	6,634,410.30	1,825.84	7,493,417.65	1,206.54	28	10,140,046.65	3.57	10,140,046.65	3.57
13	9,014,727.30	1,664.68	9,359,734.95	1,019.36	29	10,140,044.90	3.14	10,140,044.90	3.14
14	9,836,600.00	900.79	9,935,181.70	516.20	30	10,140,043.10	2.73	10,140,043.10	2.73
15	10,062,197.55	344.88	10,087,849.20	211.05	31	10,140,041.30	2.36	10,140,041.30	2.36
16	10,120,382.85	235.21	10,126,877.55	154.63	32	10,140,039.40	2.09	10,140,039.40	2.09

Table 5.4: Table that represents the mean and the standard deviation of the *threshold* q-gram distance between 20 random genomes of *E. coli* K-12 and 20 random genomes of *E. coli* O-157. We represent the number of q-grams with pair status different from $(0, 0)$, with mean and standard deviation.

We give in Table 5.4 the mean of the 20 values and their standard deviation for $1 \leq q \leq 32$. In the fourth column, we give the mean of the number of q-grams which have a pair status different from $(0, 0)$ and their standard deviation.

For each value of q, the standard deviation is less than the $\frac{1}{1000}$ of the *mean* (except for $q = 9$ where the value of the distance is really small). Hence the variation of the threshold q-gram distance between the random genomes is really low, and therefore, we can reasonably assume that an experiment using 20 random genomes or 100 random genomes would not change the results very much.

In Figure 5.7 we plot the *threshold* q-gram distance between the original genomes of *E. coli* on the left, and on the right the *threshold* q-gram distance obtained from random sequences.

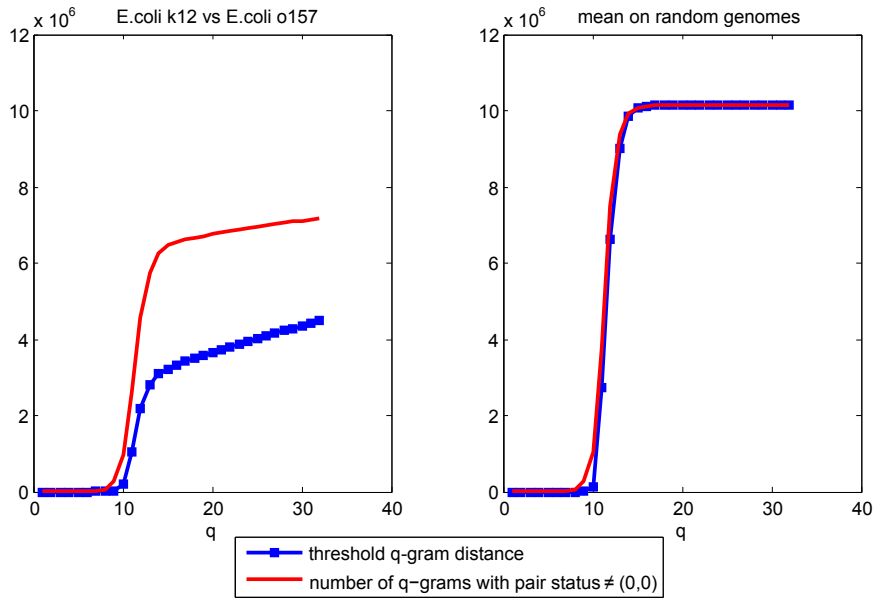


Figure 5.7: On the left the *threshold* q-gram distance between *E. coli* K-12 and *E. coli* O-157, and the relative number of q-grams with pair status different from $(0, 0)$. On the right we plot the same values calculated on a mean of 40 random genomes obtained from a permutation of the genomes of *E. coli* K-12 and *E. coli* O-157.

Note that in random genomes, for $q \geq 15$, the distance is fixed around $10,140,000$, which is exactly the sum of the length of the two genomes: $4,641,652 + 5,498,450 = 10,140,102$. This means that all q-grams appear as hapax in one of the two strings and not in the other. Hence virtually all q-grams have pair status equal to $(0, 1)$ or $(1, 0)$, for $q \geq 15$.

Now we analyse how the pair status are distributed in the random genomes, and we compare these values with the values obtained in Subsection 5.2.1.

In Table 5.5 we can see the counter of the pair status for the random genomes. As before the values inside are obtained as a mean of the 20 values calculated on random genomes.

q	$C_q(0,1)$	$C_q(0,2)$	$C_q(1,0)$	$C_q(1,1)$	$C_q(1,2)$	$C_q(2,0)$	$C_q(2,1)$	$C_q(2,2)$
1	0	0	0	0	0	0	0	4
2	0	0	0	0	0	0	0	16
3	0	0	0	0	0	0	0	64
4	0	0	0	0	0	0	0	256
5	0	0	0	0	0	0	0	1,024
6	0	0	0	0	0	0	0	4,096
7	0	0	0	0	0	0	0	16,384
8	0	0	0	0	0	0	0	65,536
9	0	0	0	0	5	0	2	262,136
10	1,531	17,193	1,367	4,865	63,648	9,515	38,427	911,572
11	499,331	491,499	429,160	513,527	551,369	325,929	423,592	517,055
12	2,910,907	557,979	2,469,190	828,058	170,556	386,506	139,272	30,949
13	4,672,912	212,372	3,950,461	344,281	16,842	148,251	13,890	726
14	5,276,840	59,546	4,456,461	98,569	1,207	41,547	999	13
15	5,441,584	15,472	4,594,236	25,651	79	10,761	66	0
16	5,484,060	3,931	4,629,637	6,495	6	2,745	4	0
17	5,494,818	994	4,638,601	1,626	0	704	1	0
18	5,497,518	255	4,640,866	405	0	182	0	0
19	5,498,197	68	4,641,437	100	0	49	0	0
20	5,498,369	19	4,641,582	24	0	14	0	0
21	5,498,411	7	4,641,617	6	0	5	0	0
22	5,498,422	3	4,641,625	1	0	2	0	0
23	5,498,424	2	4,641,627	0	0	2	0	0
24	5,498,424	2	4,641,626	0	0	1	0	0
25	5,498,424	1	4,641,626	0	0	1	0	0
26	5,498,423	1	4,641,625	0	0	1	0	0
27	5,498,423	1	4,641,624	0	0	1	0	0
28	5,498,422	1	4,641,623	0	0	1	0	0
29	5,498,421	0	4,641,623	0	0	1	0	0
30	5,498,420	0	4,641,622	0	0	1	0	0
31	5,498,420	0	4,641,621	0	0	1	0	0
32	5,498,419	0	4,641,620	0	0	0	0	0

Table 5.5: Table that represents, for each pair status, the number of q-grams with that status. The analysis is performed on 20 random genomes of *E. coli* K-12 and 20 random genomes of *E. coli* O-157 for various values of q.

We can see in the table that for $q \geq 15$, practically all q-grams are in $(0,1)$ or $(1,0)$. In Figure 5.8 we compare the counters of the pair status on original genomes with the random ones.

Just by comparing the plots of the original shapes and the randomized ones we can say that:

- $(0,1)$, $(1,0)$ original are lower than the randomized ones. In fact, we have seen that virtually all q-grams have pair status equal to $(0,1)$ or $(1,0)$, for $q \geq 15$. This is due to the fact that, for $q \geq 15$, the probability that a q-gram appear in a random sequence is low. Hence, the probability that a q-gram v compares in both strings is low. On the other hand, two real genomes that are closely related (like the two we are studying, remember that they are of the same species) have many similar genes, i.e. many segments of the sequences that are similar.

- $(1,1)$ is the one that shows more difference between random and original. Many hapax in common between the two strains of *E. coli* are obviously related to genes. In fact 87.8% of the genome of *E. coli* K12 is composed of protein-coding genes [B$^+$97]. The genes are highly conserved, hence if we compare two strains of the same species they will have many genes in common. When we shuffle the genome we lose the information of the genes in common, and $(1,1)$ becomes $(0,1)$ or $(1,0)$ in the random genomes;

- $(1,2)$, $(2,1)$ are similar between random and original, but the original ones have a lower peaks which are moved by one position to the right. We have not an explanation for this.

- $(2,2)$ is similar between random and original, except that it seems that the original one is shift on the right. It is probable that the repeats of low length are due to chance, while repeats of length greater than 11 have a biological significance;

- $(2,0)$ and $(0,2)$ have a similar behaviour. The peak is in the same position, but for $q \geq 16$ the original value are higher than the random ones. This is probably due to the duplicated genes in the genomes, which disappear in the random ones.

Even though this analysis is only a simple study, it shows that a study of this kind can lead interesting results.

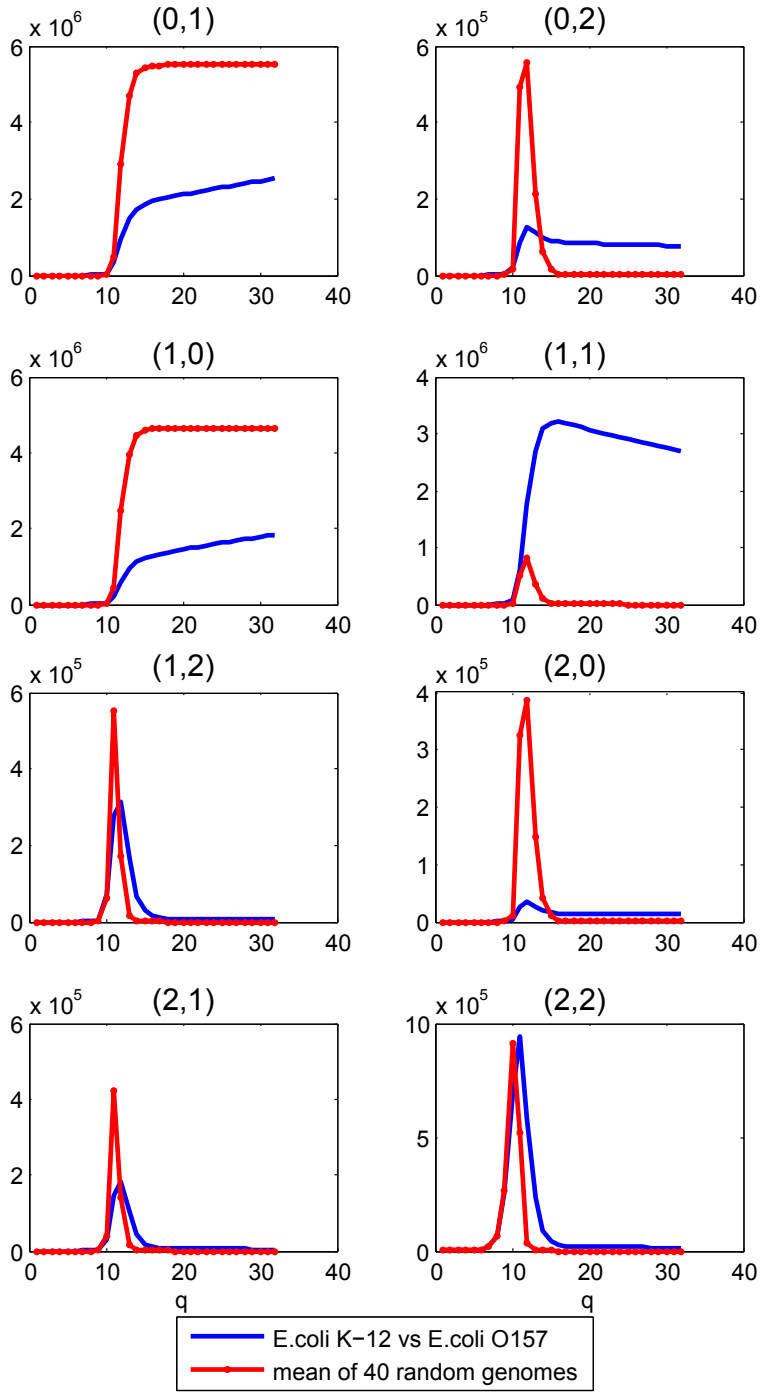


Figure 5.8: We represent the counter of the pair status on *E. coli* K-12 and *E. coli* O-157 in blue. In red we represent the mean of the counter of the pair status calculated on a mean of 40 random genomes obtained from a permutation of the genomes of *E. coli* K-12 and *E. coli* O-157. The pair status to which the graph refers is given at the top of every graph.

5.4 Four cases studies

In this section we will try to understand with a simple example how the length of the genomes influences the *threshold* q-gram distance. To choose the organisms we use the Figure 5.9 adapted from an article by Brown et al. [BDI+01].

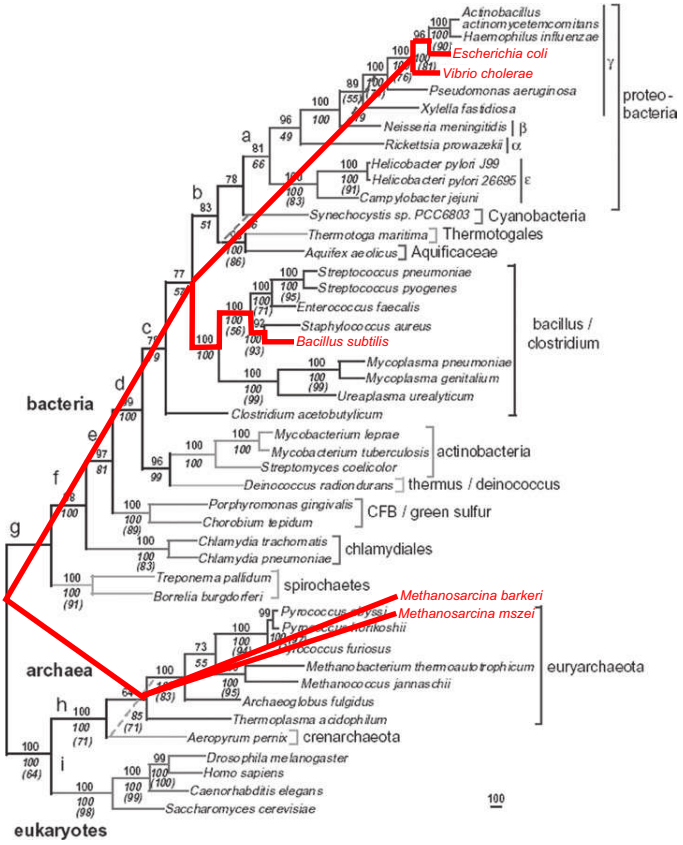


Figure 5.9: Figure adapted from the article *Universal trees based on large combined protein sequence data sets* by James R. Brown et al. [BDI+01] . In red we have evidenced the five species that we choose to analyse. *M. barkei* and *M. mazei* are not present in the study, but since they are Euryarcheota we have roughly placed them in the tree in the correspondent position.

Three of the genomes we chose are inside the tree, while the last two archaebacteria were chosen for the size of their genome. We have placed them roughly in the position where they should be. In Table 5.6 there is a summary of the properties of these five genomes. Sequences were downloaded January 20, 2015.

genome	Kingdom	Phylum	accession number	length (Mb)
Escherichia coli	Bacteria	Proteobacteria	NC_000913.3	4.64
Vibrio cholerae	Bacteria	Proteobacteria	NC_002505.1	4.03
			NC_002506.1	
Bacillus subtilis	Bacteria	Firmicutes	NC_000964.3	4.22
Methanosarcina barkeri	Archea	Euryarchaeota	CP000099.1	4.84
Methanosarcina mazei	Archea	Euryarchaeota	NC_003901.1	4.09

Table 5.6: Information and properties of the five genomes that we study.

We chose five genomes with comparable lengths. We chose *E. coli* as the reference genome. We chose *V. cholerae* because it is shorter than *E. coli*, but still has similar length, and especially because it is closely related to *E. coli*. They are both Gammaproteobacteria.

We chose *B. subtilis* because it is less closely related to *E. coli* and *V. cholerae*. It is a Firmicutes, a bacteria of a different phylum with respect to *E. coli*.

We chose two Euryarchaeotas because they are distantly related to *E. coli*, indeed they belong to different kingdoms, but they are related to each other. And we select these two genomes because one has a length greater than the reference, *M. barkeri* with 4.84 Mb; the other one (*M. mazei*) has a smaller genome, 4.09 Mb, and has a length similar to *V. cholerae*.

We measured the *threshold* q-gram distance between *E. coli* and the others genomes, in Table 5.7 there are the results.

In Figure 5.10 we can see a representation of the distances in the table.

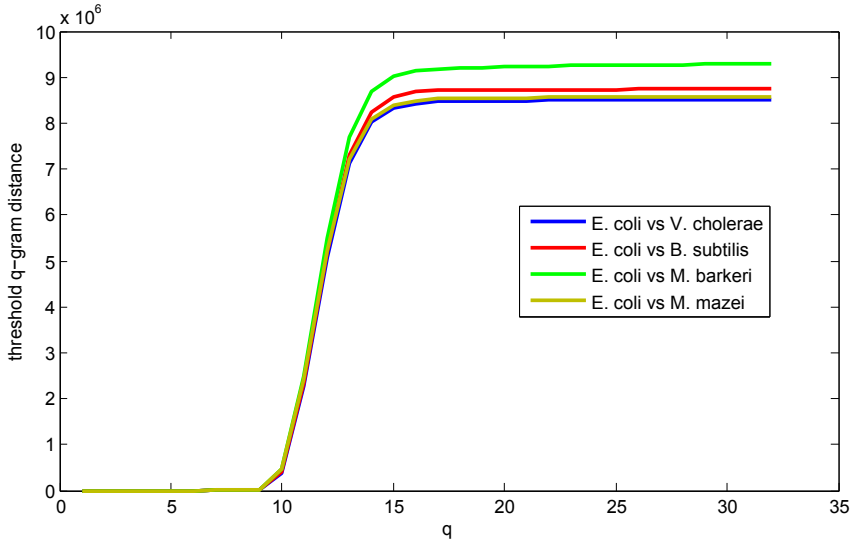


Figure 5.10: We represent the *threshold* q-gram distance between *E. coli* and 4 other genomes.

	threshold q-gram distance between E.coli and			
q	V. cholerae	B. subtilis	M. barkeri	M. mazei
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0
4	0	0	0	0
5	0	0	0	0
6	0	0	0	0
7	4	4	5	4
8	456	445	477	558
9	17,146	18,107	20,434	23,486
10	390,465	416,622	455,448	469,704
11	2,262,847	2,327,419	2,491,361	2,420,035
12	5,084,168	5,201,425	5,510,144	5,222,065
13	7,111,076	7,298,336	7,695,327	7,208,478
14	8,020,255	8,251,780	8,681,527	8,092,115
15	8,334,023	8,578,995	9,028,308	8,396,814
16	8,432,372	8,678,812	9,141,952	8,492,993
17	8,464,812	8,709,637	9,183,594	8,524,413
18	8,477,974	8,720,299	9,203,359	8,536,972
19	8,484,704	8,724,708	9,215,669	8,543,457
20	8,489,414	8,727,295	9,225,350	8,548,041
21	8,493,386	8,729,260	9,233,759	8,551,869
22	8,496,608	8,730,874	9,241,244	8,555,189
23	8,499,564	8,732,340	9,248,128	8,558,237
24	8,502,345	8,733,693	9,254,467	8,561,059
25	8,504,841	8,734,927	9,260,286	8,563,648
26	8,507,209	8,736,075	9,265,716	8,566,057
27	8,509,461	8,737,159	9,270,826	8,568,344
28	8,511,534	8,738,173	9,275,575	8,570,468
29	8,513,531	8,739,163	9,280,084	8,572,498
30	8,515,458	8,740,122	9,284,357	8,574,432
31	8,517,288	8,741,046	9,288,397	8,576,275
32	8,519,056	8,741,921	9,292,252	8,578,033

Table 5.7: Distances between *E. coli* and 4 genomes.

We can see that for $q \geq 15$ the distance depends only on the length of the genomes. In fact *M. mazei*, which belongs to a different kingdom from *E. coli*, has a lower distance then *B. subtilis*. Moreover if we look at $q = 32$, we can see that the distance of two genomes is equal to the sum of the length of the two genomes. For example, the length of *E. coli* is 4.64 Mb, the length of *B. subtilis* is 4.22 Mb, and their sum is 8.86 Mb, which is similar to 8,741,921 of the *threshold* 32-distance of the two genomes. This results is similar to the experiment we did with random genomes, hence, if we use a value of q too high, then we loose the biological information. In fact, the lower is the length of a genome, the lower is its distance from another genome, independently of the repeats/hapax content of the two genomes.

In Figure 5.11 we present the distances for $10 \leq q \leq 14$ in a more

detailed form. For $10 \leq q \leq 12$ the two Archea have a higher value of the distance with respect to the two Bacteria. For q greater than 12 the distance between *E. coli* and *M. mazei* becomes similar to the distance with *V. cholerae*. Hence for $q = 11$ or $q = 12$ the distance is proportional to the relatedness between the species. On the other hand, for $q > 12$ the length of the genome becomes prominent in the *threshold* q-gram distance.

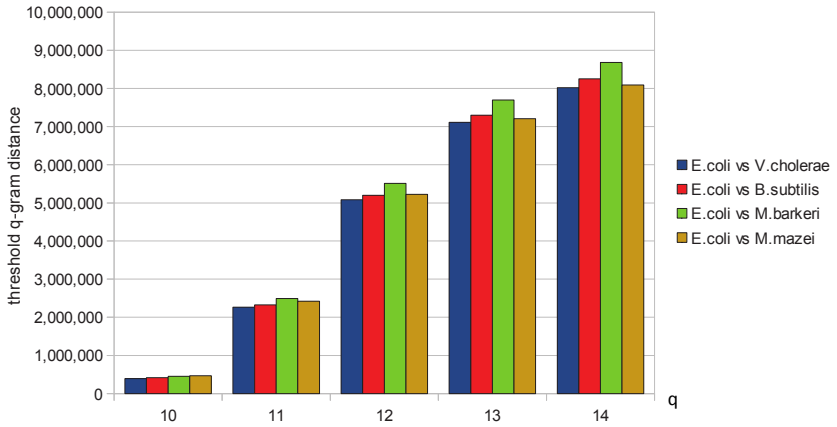


Figure 5.11: Histogram that represents the *threshold* q-gram distance between *E. coli* and the other 4 genomes studied in Table 5.7. We use only a value of q between 10 and 14 to analyse what happens in this interval.

We have to choose carefully the value of q. Since we measure hapax and repeats, one possible approach is to choose a value for q that makes the expected number of times the a q-gram will appear equal to 1, assuming that the q-grams are equally probable.

In this way we do not choose a value of q too low where we see only repeats because 4^q is much smaller than the size of the genome. Furthermore, we avoid the situation where q is high and 4^q is much greater than the size of the genomes and then we see all q-grams with pair status $(0, 1)$ or $(1, 0)$. This is what happens for $q > 12$ in Table 5.7. For high values of q, the *threshold* q-gram distance is practically equal to the sum of the lengths of the two genomes.

Let s be a string of length n over an alphabet Σ, where $|\Sigma| = \sigma$. Let x be a q-gram over Σ. We assume that the probability that x occurs in a fixed position in s is independent of the other q-grams in s. Even though the occurrence of q-grams in nearby positions in the same string are not independent events, it has been argued that modelling it as independent events (ball and urn model) does lead to a good approximation [Nic03]. So we adopt this simple (and simplistic) model here: assume that q-grams occur in positions independently of each other. Assume that the characters of the string s are chosen uniformly at random (i.e. with probability $1/\sigma$). Then,

the expected number of occurrences of a q-gram x in s is $(n-q+1)/\sigma^q$. This is because there are $(n-q+1)$ positions in s, and the probability that x starts at a fixed position in s is $1/\sigma^q$. If we want to have a frequency of x equal to 1 then we have to put:

$$\frac{(n-q+1)}{\sigma^q} = 1$$
$$(n-q+1) = \sigma^q$$

Since n is much greater than $q+1$ we can write:

$$n = \sigma^q$$
$$q = \log_\sigma(n) = \frac{\ln(n)}{\ln(\sigma)}$$

Applied to the previous example we have that the mean of the length of the genomes is $4,36$ Mb. Hence $q = \log_4(4{,}360{,}000) = 11.03$. If we take the floor of this number we choose $q = 11$, which is agrees with previous experiments.

Chapter 6

Conclusions

We have studied the *threshold* q-gram distance and some of its properties. We have proven that there are five transformations that leave the *threshold* q-gram profile unchanged: two were introduced by Ukkonen [Ukk92] and three are introduced in this thesis. If we apply one of these five operations to a string s to obtain s', then s and s' have the same *threshold* q-gram distance. The two Ukkonen transformations allow you to find all strings with a given balanced de Bruijn subgraph. The three transformations we described can be visualized as transformations between balanced de Bruijn subgraphs without changing the underling protograph. Whether it is possible to create all Eulerian realizations of a protograph using only these three operations is an open problem.

In Chapter 4 we have studied several implementations of the *threshold* q-gram distance. One possible implementation we have not looked at is using suffix automata [Ukk92].

All implementations use an integer code to represent the q-grams, but this could be a limit since the number of q-grams grows exponentially with q: it equals σ^q, where σ is the size of the alphabet. To represent such a number we need $\log_2(\sigma^q) = q \cdot \log_2(\sigma)$ bits. It is preferable for us if the size of the integer not exceed the register size of the CPU. Most processors today compute with 64 bits: this justifies why we used in our implementation an integer of 64 bits. If we need larger values of q then we could divide each q-gram in segments and work on them as grams of lower size. In this way, we do not need to define a new data type, since in many programming language the maximum integer has size 64 bits.

Another interesting question is how the *threshold* q-gram profiles relate to each other for different values of q, and analogously for the *threshold* q-gram distance. For example what can we say about $TQD_{q+1}(s,t)$ if we know $TQD_q(s,t)$, and what about the relative *threshold* q-gram profiles.

If we answer questions of this type, we may be even be able to compute $TQD_q(s,t)$ for high values of q using the *threshold* q-gram distance on lower

values of q.

In Chapter 5 we studied three possible applications of the *threshold* q-gram distance, and each one can be improved:

1. Studying the repeat and hapax content of two genomes. This study is related to [FM13] and [CFM12], which contain studies of hapax and repeat content of one genome at time (*intra-genome* study). With the *threshold* q-gram distance, we can study the repeats/hapax that are present in both genomes and their distribution among different organisms, so that it is an *inter-genomes* study.

   In [CFM14], a repeat sharing gene network was introduced, permitting the study of paralogous genes. Paralogous genes are genes within the same genome which results from a gene duplication. Now we can define a new type of gene network that permit to study orthologous genes. These are genes in different species that evolved from a common ancestral gene by speciation. We can define a new gene network where the nodes are the genes of two organisms, and each node has a label that identifies the organism. An undirected edge connects two genes with different labels if the two genes have a q-gram in common.

2. We can use random genomes to evaluate how much the properties we are studying are due to chance or have a biological significance. We have studied how to create a random genome, while the statistical method to use has to be investigated.

3. The *threshold* q-gram distance can be used to evaluate the similarity between genomes. A study with many other genomes should be done.

Bibliography

[AAJdCLV09] Hanson AD, Pribat A, Waller JC, and de Crcy-Lagard V. 'unknown' proteins and 'orphan' enzymes: the missing half of the engineering parts list–and how to find it. *Biochem J.*, 425(1):1–11, 2009.

[B+97] Frederick R. Blattner et al. The complete genome sequence of escherichia coli k-12. *Science*, 277(5331):1453 – 1462, 1997.

[Bar09] Janet Heine Barnett. *Resource for teaching discrete mathematics*, chapter Early Writings on Graph Theory: Euler Circuits and The Königsberg Bridge Problem, pages 197 – 208. Mathematical Association of America, 1st edition, 2009.

[BDI+01] JR Brown, CJ Douady, MJ Italia, WE Marshall, and MJ Stanhope. Universal trees based on large combined protein sequence data sets. *Nat. Genet.*, 28:281 – 285, 2001.

[CFM12] Alberto Castellini, Giuditta Franco, and Vincenzo Manca. A dictionary based informational genome analysis. *BMC Genomics*, 13(485), 2012.

[CFM14] Alberto Castellini, Giuditta Franco, and Alessio Milanese. A genome analysis based on repeat sharing gene networks. *Natural computing*, 2014.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[Con01] International Human Genome Sequencing Consortium. Initial sequencing and analysis of the human genome. *Nature*, 409:860 – 921, 2001.

[D+83] R. F. Doolittle et al. Simian sarcoma virus onc gene, v-sis, is derived from the gene (or genes) encoding a platelet-derived growth factor. *Science*, 221(4607):275–277, 1983.

[dB46] Nicolaas de Bruijn. A combinatorial problem. *Koninklijke Nederlandse Akademie v. Wetenschappen*, 48:758 – 764, 1946.

[Dic06] Allen Dickson. Introduction to graph theory. Lecture notes, downloaded from `http://www.math.utah.edu/mathcircle/notes/MC_Graph_Theory.pdf`, October 2006.

[Die12] Reinhard Diestel. *Graph Theory, 4th Edition*, volume 173 of *Graduate texts in mathematics*. Springer, 2012.

[dKGC+11] A. P. Jason de Koning, Wanjun Gu, Todd A. Castoe, Mark A. Batzer, and David D. Pollock. Repetitive elements may comprise over two-thirds of the human genome. *PLoS Genet*, 7(12):e1002384, 12 2011.

[Eve05] Daniel L. Everett. Cultural constraints on grammar and cognition in pirahã. *Current Anthropology*, 46(4):621–646, 2005.

[Fit83] Walter M. Fitch. Random sequences. *Journal of Molecular Biology*, 163:171 – 176, 1983.

[FM13] Giuditta Franco and Alessio Milanese. An investigation on genomic repeats. In *The Nature of Computation. Logic, Algorithms, Applications - 9th Conference on Computability in Europe, CiE 2013, Milan, Italy, July 1-5, 2013. Proceedings*, pages 149–160, 2013.

[Fra13] Giufitta Franco. personal communication, 2013.

[Hie73] Carl Hierholzer. Ueber die möglichkeit, einen linienzug ohne wiederholung und ohne unterbrechung zu umfahren. *Math. ann.*, 6:30 – 32, 1873.

[JAGM08] Minghui Jiang, James Anderson, Joel Gillespie, and Martin Mayne. ushuffle: A useful tool for shuffling biological sequences while preserving the k-let counts. *BMC Bioinformatics*, 9:192, 2008.

[KR87] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. pages 249–260, 1987.

[Lev66] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10:707, February 1966.

[Nic03] Pierre Nicodme. q-gram analysis and urn models. *Proc. of Discrete Random Walks DRW2003*, pages 243–258, 2003.

[NW70] Saul B. Needleman and Christian D. Wunsch. Ia general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–53, 1970.

[ONM+84] AD O'Brien, JW Newland, SF Miller, RK Holmes, HW Smith, and SB Formal. Shiga-like toxin-converting phages from escherichia coli strains that cause hemorrhagic colitis or infantile diarrhea. *Science*, 226(4675):694 – 696, 1984.

[Pev89] Pavel A. Pevzner. 1-tuple DNA sequencing: computer analysis. *J Biomol Struct Dyn.*, 7:63–73, 1989.

[Pev95] Pavel A. Pevzner. DNA physical mapping and alternating eulerian cycles in colored graphs. *Algorithmica*, 13(1/2):77–105, 1995.

[SGK+12] Jens Stoye, Robert Giegerich, Stefan Kurtz, Enno Ohlebuch, Elys Willing, Peter Husemann, Roland Witter, and Katharina Westerholt. Sequence analysis i + ii. Lecture notes of Bielefeld University, Faculty of Technology., Summer 2012.

[Sha48] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423, 1948.

[Ukk92] Esko Ukkonen. Approximate string matching with q-grams and maximal matches. *Theor. Comput. Sci.*, 92(1):191–211, 1992.