Computing Genomic Distances: An Algorihtmic Viewpoint

Guillaume Fertin, Irena Rusu

CHAPTER 1

COMPUTING GENOMIC DISTANCES : AN ALGORITHMIC VIEWPOINT

by Guillaume FERTIN and Irena RUSU

1.1 INTRODUCTION

What this Chapter is About. Comparative genomics is a field of bioinformatics in which the goal is to compare several species by comparing their genomes, in order to understand how the different species under study have evolved in time. This study leads for instance to reconstructing putative ancestral genomes, building phylogenetic trees, or inferring the functionality of genes or sets of genes.

One of the main activities in comparative genomics consists in comparing *pairs* of genomes, in order to identify their common features, and thus also to determine what differentiate them. In that case, genomes are usually modeled as sequences of *genes*, where a gene is identified by a (possibly signed) label. The sign + or -, if present, indicates on which DNA strand the gene lies. In that context, the *order* of the genes in the studied genomes is the main information we are given. Note that the way this order was obtained is out of our scope here: only the order itself is taken into account.

It should also be noted that genomes may contain *several occurrences* of the same gene (possibly carrying different signs, if signs are present). In that case, we say that a genome contains *duplicates*. Indeed, genes may be duplicated during evolution, and duplicate genes actually occur frequently in all living species.

Comparing pairs of genomes on that basis can roughly be done in two different ways:

1. Compare the *structure* of the two genomes under study by computing a measure that represents the (dis)similarity between the genomes.

2. Infer the *evolution process* from one genome to another. For this, one needs to consider one or several operations (called *rearrangement(s)*) that can occur in a genome during evolution, e.g. inversions or translocations ; and the goal is to determine the most parsimonious (i.e., less costly) rearrangement scenario that leads from one genome to the other.

In this chapter, we only focus on option 1. above. This static viewpoint has the advantage to allow us to identify conserved regions between genomes, which is not the case with option 2. Note also that, although the term *distance* is very often used for option 1. (as is done in the title of this chapter), this only refers to *evolutionary distance*, i.e. the amount of changes that occurred during the evolution process. Indeed, the so-called "distances" that have been defined in the literature are rarely *mathematical distances*: they are *measures* that evaluate the differences and similarities resulting from evolution between the two genomes, either by directly counting the number of changes or, in a complementary way, by counting the conserved regions. Hence, in the following, we use the term *measure* rather than distance.

The purpose of this chapter is to present some *algorithmic aspects* of pairwise genome comparisons, when those comparisons aim at finding a (dis)similarity measure. More precisely, we present several algorithms that were proposed recently for solving (exactly or approximately) several variants of the problem. Our goal is not to survey exhaustively all the existing results on that topic, but rather to give a sample of different algorithmic ideas and techniques that have been used to answer some of the problems. Besides the fact that it presents original and non trivial concepts that we think are of interest for the reader, it also gives a flavor of the inventiveness and the richness of recent research on the subject.

Definitions and Notations. Genomes under consideration in this chapter are represented as sequences of (possibly signed) integers, built from the alphabet $\Sigma = \{1, 2, 3 \ldots, n\}$, where n is as large as necessary. When *unsigned* (resp. *signed*) genomes are considered, then their representation is a sequence of unsigned (resp. signed) integers. When a sequence contains distinct integers (that is, the corresponding genome has no duplicates), the sequence is called a *permutation*, while in the contrary case it is called a *string*. For instance, $P = (2\,{-3}\,8\,{-4}\,{-5}\,1\,7\,{-6})$ is a signed permutation, while $Q = (3\,{-4}\,{-3}\,2\,1\,2\,2)$ is a signed string.

For any genome P, its length is denoted as m_P. Moreover, for any $1 \leq i \leq j \leq m_P$, $P[i]$ denotes the i-th element of P, $|P[i]|$ is $P[i]$ whose sign has been removed, and $P[i, j] = (P[i], P[i + 1], \ldots, P[j])$ denotes the portion of

P whose extremities are given by indices i and j, both being included. For instance, if $P = (2\ -3\ 8\ -4\ -5\ 1\ 7\ -6)$, then $P[2] = -3$, $|P[2]| = 3$, and $P[2,4] = (-3\ 8\ -4)$.

A *duo* in a genome P is a set of two consecutive elements of P. For any $1 \leq i \leq m_P - 1$, the duo d_i represents $P[i]P[i+1]$, and is denoted as follows: $d_i = (P[i], P[i+1])$. Two duos $d = (a, b)$ and $d' = (a', b')$ of a signed genome are said to be *identical* if (i) $a = a'$ and $b = b'$ or (ii) $a = -b'$ and $b = -a'$. If the genome is unsigned, then two such duos are identical whenever $a = a'$ and $b = b'$ only. Given two permutations P and Q built on the same alphabet, an *adjacency* in P is a duo d for which there exists a duo d' in Q such that d and d' are identical. Whenever d is not an adjacency in P, then it is a *breakpoint*. We note that these two notions are symmetric: that is, given two permutations P and Q built on the same alphabet, the number of adjacencies (resp. breakpoints) in P is equal to the number of adjacencies (resp. breakpoints) in Q.

Organization of the Chapter. This chapter is organized as follows: in Section 1.2, we are interested in comparing pairs of genomes by finding their common or conserved intervals. In this context, three algorithms are presented. Section 1.3 is devoted to two algorithms to determine the minimum number of breakpoints between pairs of genomes containing duplicates. Section 1.4 is the conclusion.

1.2 INTERVAL-BASED CRITERIA

1.2.1 Brief introduction

Breakpoints and adjacencies are easy to compute in permutations, but do not give much insight on the genome organization in terms of genes that are close to each other in both genomes. However, these groups of genes are particularly interesting, since they show a similarity in their content between genomes, similarity which has been preserved in spite of the past evolutionary events.

Looking to *intervals* rather than to *duos* allows us to model such regions with identical content but different gene order. In this context, the location of the genes changes from one genome to the other, some genes may even get duplicated, but the new locations and the possibly duplicates still form an interval in the second genome.

Going further, that is, speaking about substrings with (almost) identical content but whose genes are not consecutive, is possible, but in this case measuring the similarity or dissimilarity between genomes becomes difficult (the 'missing' genes allow us to have strictly overlapping regions as well as long gaps). These regions are more successfully seen like *clusters of genes* sharing a potential common function. Their importance is undeniable, but

is not the point of this chapter. This is why we limit our study to measures based on intervals.

1.2.2 The context and the problems

Let P be a signed string of integers over the finite alphabet Σ, representing a linear genome.

Definition 1 [13] The **character set** of the interval $P[i, j]$ of P, with $1 \leq i \leq j \leq m_P$, is defined by

$$\mathcal{CS}(P[i, j]) = \{|P[h]| : i \leq h \leq j\}$$

The character set of an interval stores the content of the interval, regardless of the order of genes within it, their signs or their number of occurrences. When the same character set is defined by two intervals of two strings, then a strong local similarity is identified between the two strings.

Definition 2 [13] Let $\mathcal{C} \subseteq \Sigma$ be a set of integers and P, Q be two signed strings over Σ. Set $\mathcal{C}$ is a **common interval** of P and Q if there exist positions a, b, i, j, with $a \leq b$ and $i \leq j$, such that

$$\mathcal{CS}(P[a, b]) = \mathcal{CS}(Q[i, j]) = \mathcal{C}.$$

This definition allows several variants:

(1) **common intervals of two permutations**, defined in [16], when P and Q are permutations on $\{1, 2, \ldots, n\}$.
(2) **conserved intervals of two permutations**, defined in [4], when P and Q are signed permutations on $\{1, 2, \ldots, n\}$ and the common interval is required to satisfy either $P[a] = Q[i]$ and $P[b] = Q[j]$, or $P[a] = -Q[j]$ and $P[b] = -Q[i]$.
(3) **common intervals of two strings**, defined in [13], when P and Q are unsigned strings with elements in Σ.

Each variant defines a criterion to measure the **similarity** between strings P and Q as the **number of common/conserved intervals** of P and Q. In the case of common intervals and of conserved intervals in permutations, a mathematically rigorous notion of distance (satisfying the three properties of a metric) may be defined as follows. Note that the number of common intervals between a permutation P and itself is the number of intervals of P, that is $n(n + 1)/2$. The same holds for conserved intervals.

Definition 3 Let $Common(P, Q), Conserved(P, Q)$ be respectively the number of common and conserved intervals of two permutations P and Q. The **common intervals distance** between P and Q is defined as

$$distance_{Common}(P, Q) = n(n+1) - 2Common(P, Q).$$

The **conserved intervals distance** between P and Q (assumed signed) is defined as

$$distance_{Conserved}(P, Q) = n(n+1) - 2Conserved(P, Q).$$

This notion of distance was introduced in [4] for conserved intervals, and in the more general case where each of P and Q is replaced by a set of permutations. The extension of the definitions and methods used in this section to more than two strings is briefly discussed in Section 1.2.6. In the case of common intervals in strings, the possible difference between the lengths of P and Q as well as the possibly different number of locations of each interval in each string make a neat definition much more difficult to find.

In the next three sections, we present algorithms to compute the number of common intervals between P and Q according to each of the three variants above. These three algorithms allow us to see and discuss three different ways to reach a similar goal. The reader will note that in the case of common intervals, both in permutations and in strings, the algorithms given here really compute each of the searched intervals. The number of intervals is then implicitly computed. In the case of conserved intervals, it is possible to compute directly the number of intervals, without displaying them all.

To understand the differences between the three approaches, let us start with a slightly deeper analysis of the problem. In the following, a pair (x, i) indicates the element x of Σ situated at position i in Q. Looking for a common interval of P and Q is looking for two positions i and j on Q such that the elements in the interval $Q[i, j]$ have neighboring localizations on P.

Definition 4 Given a position i, a **Max zone** of $(Q[i], i)$ is any maximal interval of P whose character set is (regardless of signs) included in $\mathcal{CS}(Q[i, m_Q])$. A **Min zone** of $(Q[i], i)$ is any maximal interval of P whose character set is (regardless of signs) included in $\mathcal{CS}(Q[1, i])$.

Recalling that the elements in the interval $Q[i, j]$ are both in $\mathcal{CS}(Q[i, m_Q])$ and in $\mathcal{CS}(Q[1, j])$, the following lemma is then easy to deduce:

Lemma 1 *The set $\mathcal{C}$ defined by $\mathcal{C} = \mathcal{CS}(Q[i, j])$, with $1 \leq i \leq j \leq m_Q$, is a common interval of P and Q if and only if there exist a Max zone of $(Q[i], i)$ and a Min zone of $(Q[j], j)$ whose intersection is $\mathcal{C}$.*

Figure 1.1 illustrates this lemma. Note that, when P and Q are permutations (with or without signs), there is exactly one Max zone and one Min zone for every element in Q. The three approaches are summarized below.

Commuting Generators. The algorithm for common intervals in permutations, presented in Section 1.2.3 and introduced in [3], identifies for every element

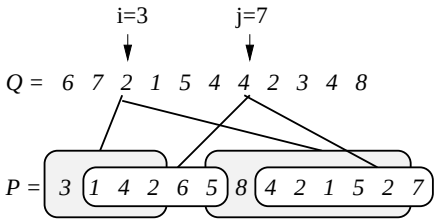

Figure 1.1 Max zones (grey boxes) of $(2,3)$ and Min zones (white boxes) of $(4,7)$ when $Q = (6\,7\,2\,1\,5\,4\,4\,2\,3\,4\,8)$ and $P = (3\,1\,4\,2\,6\,5\,8\,4\,2\,1\,5\,2\,7)$. Sets $\{1,2,4\}$, $\{5\}$ and $\{1,2,4,5\}$ may all be obtained as the intersection of a Max zone of $(2,3)$ with a Min zone of $(4,7)$, but only $\{1,2,4,5\}$ is $\mathcal{CS}(Q[3,7])$ and is thus a common interval of P and Q.

$(Q[i],i)$ in Q (renumbered such that Q is the identity permutation) a subzone of the Max zone and a subzone of the Min zone containing consecutive elements (including i) with respect to the order in Q. These subzones are called generators. A common interval is then defined by any pair i,j such that j is in the Max subzone of i and i is in the Min subzone of j. Once the subzones are computed, obtaining all the common intervals is an easy task. The resulting algorithm is in $O(n+N)$, where $n = m_P = m_Q = Card(\Sigma)$ is the common length of P and Q, and N is the number of common intervals between P and Q.

Bound-and-Drop. The algorithm for conserved intervals in (signed) permutations, presented in Section 1.2.4 and introduced in [4], considers for each j in Q (renumbered such that Q is the identity permutation) the candidates $i < j$ in Q such that $Q[i]$ is in the Min zone of $(Q[j],j)$ in P, $Q[j]$ is in the Max zone of $(Q[i],i)$ in P, and the number of elements in the interval of P with endpoints $Q[i]$ and $Q[j]$ is $j-i+1$. The bad candidates i are dropped, whereas the firstly found suitable one is validated. The result is a set of special conserved intervals $\mathcal{CS}([i,j])$, called irreducible intervals, which allows to quickly compute, in $O(n)$, the total number of conserved intervals. An algorithm to find all the conserved intervals in $O(n+N)$ time is then easily obtained (where $n = m_P = m_Q = Card(\Sigma)$).

Element Plotting. The algorithm for common intervals in strings, presented in Section 1.2.5 and introduced in [13], deals with multiple Max zones and Min zones for any fixed $(Q[i],i)$ by considering in a left to right order all the positions $j > i$ and plotting on P the elements found in $Q[i,j]$. The Max zones of $(Q[i],i)$ and the Min zones of $(Q[j],j)$ are in this way computed simultaneously but uncontiguously and incompletely (only the useful elements, those in $Q[i,j]$, are plotted). When an interval of plotted elements in P has the same number of distinct elements as $\mathcal{CS}(Q[i,j])$, a common interval is displayed. This is equivalent to saying that the intersection of a Max zone of

$(Q[i], i)$ and a Min zone of $(Q[j], j)$ is an interval which contains exactly the elements in $\mathcal{CS}(Q[i, j])$. The algorithm presented here runs in $\theta(m^2)$ (where $m = max\{m_P, m_Q\}$), but may be improved to $O(m^2)$ as shown in [13].

1.2.3 Common Intervals in Permutations and the Commuting Generators Strategy

Common intervals of two permutations were introduced in [16], together with a first (and quite complex) algorithm in $O(n + N)$ to compute them.

The algorithm we present in this section was proposed in [3]. The genomes P and Q are represented as permutations on $\Sigma = \{1, 2, \ldots, n\}$. Moreover, in order to simplify the presentation, we assume without loss of generality that Q is the identity permutation, denoted Id, and P is an arbitrary permutation. This can be easily achieved given two arbitrary permutations, by renumbering Q to obtain Id and renumbering P accordingly. As a consequence, given $i \in \{1, 2, \ldots, n\}$ we have that $Q[i] = Id[i] = i$, and that the Max zone and Min zone of (i, i) may be redefined as follows :

Definition 5 Given $i \in \{1, 2, \ldots, n\}$, define on P the following intervals:

$IMax[i]$: the largest interval containing i and elements greater than i,
$IMin[i]$: the largest interval containing i and elements smaller that i.

Let $(i..j)$ be a shorter notation for $\mathcal{CS}(Id[i, j])$, that assumes $1 \leq i \leq j \leq n$. According to Lemma 1, a set $(i..j)$ is a common interval of P and Id if, and only if, the equality $\mathcal{CS}(IMax[i]) \cap \mathcal{CS}(IMin[j]) = (i..j)$ holds. With the supplementary notation

$Sup[i]$: the largest integer such that $(i..Sup[i]) \subseteq \mathcal{CS}(IMax[i])$, and
$Inf[i]$: the smallest integer such that $(Inf[i]..i) \subseteq \mathcal{CS}(IMin[i])$,

the latter equality holds if and only if $j \leq Sup[i]$ and $Inf[j] \leq i$. Equivalently, we have that $(i..j)$ is a common interval of P and Id if and only if

$$(i..j) = (i..Sup[i]) \cap (Inf[j]..j)$$

.

The vectors Sup and Inf, both of size n, are thus sufficient to generate all the common intervals $(i..j)$ (and only the common intervals) using the preceding formula. But this pair of vectors is not necessarily unique. A general definition may be given:

Definition 6 A pair (R, L) of vectors of size n is a **generator** for the common intervals of P and Id if the following properties hold:

(i) $R[i] \geq i$ and $L[i] \leq i$, for all $1 \leq i \leq n$, and
(ii) $(i..j)$ is a common interval of P and Id if and only if
$$(i..j) = (i..R[i]) \cap (L[j]..j).$$

The pair (Sup, Inf) allows us to affirmatively answer the question whether a generator exists for every P and Id. Paper [3] deeply analyzes generators for two or more permutations, and for other families of intervals. We focus here on computing the common intervals of two permutations, which is done with Algorithm 1.1.

Algorithm 1.1 Algorithm Common_Intervals_In_Permutations [3]

Step 1. `For all` $i \in \{1, 2, \ldots, n\}$ `do compute` $IMax[i]$ `and` $IMin[i]$ `Endfor;`
Step 2. `Compute` (Sup, Inf)`;`
Step 3. `Compute common intervals.`

Consider Steps 1 to 3 one by one.

Step 1. As $IMax[i]$ and $IMin[i]$ are intervals of P, computing them just requires to compute their left and right endpoints. In order to ensure a linear complexity, all the endpoints of a given type are obtained during a single search along P. To obtain, for instance, the left endpoints $LMin[i]$ of $IMin[i]$ $(1 \leq i \leq n)$, Algorithm 1.2 is proposed, which uses a stack S to store in a convenient order the current candidates. The linear running time of this algorithm is obvious.

Algorithm 1.2 Algorithm Step 1 (LMin version) [3]

$\{S$ *is a (initially empty) stack of positions in* $P\}$
`Stack 0 on S;` $P[0] \leftarrow n+1$`;`
`For` $h = 1$ `to` n `do`
  `While` $P[top(S)] < P[h]$ `do Unstack` $top(S)$ `Endwhile;`
  $LMin[P[h]] \leftarrow top(S) + 1$`;`
  `Stack` h `on` S`;`
`Endfor.`

Step 2. Computing (Sup, Inf) when the endpoints of $IMax[i]$ and $IMin[i]$ are known is based on the following property.

Lemma 2 *Let P be a permutation. If $(i..k) \subseteq \mathcal{CS}(IMax[i])$, then $Sup[i] \geq Sup[k]$. If $(k..i) \subseteq \mathcal{CS}(IMin[i])$ then $Inf[i] \leq Inf[k]$.*

The proof of this lemma is easily done by noting that we have $(k..Sup[k]) \subseteq \mathcal{CS}(IMax[k])$, and that $IMax[k]$ is included in $IMax[i]$. Then both $(i..k)$ and $(k..Sup[k])$ are subsets of $\mathcal{CS}(IMax[i])$ and so is their union. The conclusion follows from the definition of $Sup[i]$. A similar reasoning is valid for the second part of the lemma.

Lemma 2 allows to compute $Sup[i]$ in the decreasing order of i. Assuming $Sup[i+1], \ldots, Sup[n]$ are already known, one obtains $Sup[i]$ by initializing it with i and successively updating it to $Sup[k_h]$ as long as the element k_h

defined by $k_1 = i + 1$, $k_{h+1} = Sup[k_h] + 1$ $(h \geq 1)$ is found in $IMax[i]$. Algorithm 1.3 is based on this idea and runs in $O(n)$ since the total number of updates over all i is $n - 1$.

Algorithm 1.3 Algorithm Step 2 [3]

```
{W, w are two vectors of size n}
Inf[1] ← 1; Sup[n] ← n;
For i = 1 to n do W[i] ← i; w[i] ← i Endfor;
For i = n − 1 downto 1 do
   While W[i] + 1 is in IMax[i] do W[i] ← W[W[i] + 1] Endwhile;
   Sup[i] ← W[i]
Endfor;
For i = 2 to n do
   While w[i] − 1 is in IMin[i] do w[i] ← w[w[i] − 1] Endwhile;
   Inf[i] ← w[i]
Endfor.
```

Step 3. To compute common intervals, a *commuting* generator is needed in order to ensure a minimum running time for the algorithm.

Definition 7 A generator (R, L) is **commuting** if each of the collections of sets $\{(i..R[i]) : 1 \leq i \leq n\}$ and $\{(L[i]..i) : 1 \leq i \leq n\}$ has the property that any two distinct sets of the collection are either disjoint or one of them contains the other one.

Fortunately, the generator (Sup, Inf) computed in Step 2 is commuting (easily deduced either using the definition or Algorithm 1.3); but Algorithm 1.4 works as well for an arbitrary commuting generator. It first computes for every element (and position) $i > 1$ in Id a value $Support[i]$ which gives the rightmost position $h < i$ such that $Id[h, R[h]]$ contains $Id[i, R[i]]$. Then, for each j in decreasing order, it successively identifies the positions $i = Support^q[j]$ $(q \geq 0$, with the convention that $Support^0[j] = j$, and $Support^r[j] = Support[Support^{r-1}[j]])$ such that $L[j] \leq i$, that is, $(L[j]..j)$ contains i. This is sufficient to ensure that $(i..j)$ is a common interval, since we also have, by the definition of the vector $Support$, that $R[i] \geq j$ and thus $(i..j) = (i..R[i]) \cap (L[j]..j)$.

The second *while* loop of this algorithm is executed proportionally to the number of common intervals it outputs, so that the running time of the algorithm is in $O(n + N)$.

Algorithm Common_Intervals_In_Permutations first collects the necessary information, and then builds at once all the common intervals, in contrast with the next algorithms, which use sequentially collected information to sequentially display the common intervals.

Algorithm 1.4 Algorithm Step 3 [3]

```
{S is a (initially empty) stack of positions in Id}
{R, L are two vectors of size n, representing the given generator}
Stack 1 on S
For i = 2 to n do
   While R[top(S)] < i do Unstack top(S) Endwhile;
   Support[i] ← top[S];
   Stack i on S
Endfor;
For j = n downto 1 do
   i ← j;
   While i ≥ L[j] do
     Output the common interval (i..j);
     i ← Support[i]
   Endwhile
Endfor.
```

1.2.4 Conserved Intervals in Permutations and the Bound-and-Drop Strategy

In [4], conserved intervals were introduced as a family of common intervals that should not be broken by rearrangement operations on the genome. An $O(n)$ algorithm to compute the number of conserved intervals between two (and even more, see Section 1.2.6) permutations is given in the paper, and we present it below. Displaying all the conserved interval in $O(n + N)$, where N is the total number of conserved intervals, is an easy task using Lemma 3 below and the algorithm.

In this section, genomes are *signed* permutations on $\{1, 2, \ldots, n\}$. Without loss of generality, we assume once again that one of the permutations is the identity permutation Id, and the other permutation P is an arbitrary *signed* permutation. Moreover, since singletons are known conserved intervals, they are omitted from the presentation below. All the conserved intervals considered in the remaining of this section are therefore, by definition, **non-singletons**.

Definition 8 Le P be a signed permutation. A conserved interval C of Id and P is **reducible** if there exist smaller intervals $C_1, C_2, \ldots, C_h$ $(h \geq 2)$ such that C is the union of $C_1, C_2, \ldots, C_h$. In the contrary case, C is called **irreducible**.

Note that irreducible conserved intervals are not necessarily minimal with respect to inclusion. Moreover, it is easy to prove that irreducible conserved intervals are either disjoint, or included in each other (and with different endpoints on each permutation), or overlapping on exactly one element, so that they form chains:

Definition 9 Let P be a signed permutation. A collection $C_1, C_2, \ldots, C_l$ $(l \geq 1)$ of irreducible conserved intervals of P and Id is a **chain** if C_x and

C_{x+1} have exactly one element in common, for all x, $1 \leq x \leq l-1$. A chain $C_1, C_2, \ldots, C_l$ ($l \geq 1$) is **maximal** if no irreducible conserved interval C_0 exists such that $C_0, C_1, C_2, \ldots, C_l$ or $C_1, C_2, \ldots, C_l, C_0$ is a chain.

Maximal chains partition the collection of irreducible conserved intervals. Moreover, a conserved interval is always a chain (not necessarily maximal), so that estimating the number of conserved intervals and displaying each of them when irreducible conserved intervals are known may be done using the following result:

Lemma 3 *Let P be a signed permutation. A maximal chain $C_1, C_2, \ldots, C_l$ ($l \geq 1$) of irreducible conserved intervals of P and Id generates exactly $l(l+1)/2$ conserved intervals.*

It remains to give the algorithm for finding the irreducible conserved intervals. Note that the conserved intervals with endpoints i, j, $1 \leq i \leq j \leq n$ on Id have either positive endpoints i, j (in this order from left to right) or negative endpoints $-j, -i$ (in this order from left to right) on P. The algorithm given below shows how to identify the irreducible conserved intervals with positive endpoints (called *positive irreducible intervals*). The same algorithm, applied to Id and to the result $\overline{P}$ of a complete signed reversal on P (that is, $\overline{P} = (-P[n]\ -P[n-1]\ \ldots\ -P[1])$) identifies the irreducible conserved intervals with negative endpoints (called *negative irreducible intervals*).

In the algorithm, P^* is the permutation $(0\, P[1] \ldots P[n]\, n+1)$. Moreover, $LMin[i]$ is the left endpoint of the interval $IMin[i]$ defined as in Definition 5, but for P^*_+, the unsigned permutation obtained from P^* by removing the signs.

Algorithm 1.5 Algorithm Positive_Irreducible_Intervals [4]

```
{S is a (initially empty) stack of indices in P*}
{B is a vector of size n + 2}
Stack 0 on S; B[0] ← n + 1;
Compute LMin[i], i = 0,...,n + 1, using Algorithm Step 1 (LMin version);
For j = 1 to n + 1 do
   If LMin[j] > 1 then B[j] ← |P*[LMin[j] − 1]| else B[j] ← n + 1 Endif;
   While |P*[j]| < P*[top(S)] or |P*[j]| > B[top(S)] do
     Unstack top(S)
   Endwhile;
   If j − top(S) = P*[j] − P*[top(S)] and B[j] = B[top(S)] then
      Output the positive irreducible interval (P*[top(S)]..P*[j])
   Endif;
   If P*[j] > 0 then Stack j on S Endif;
Endfor.
```

It is important to note that each index j in P can be the right endpoint of at most one positive irreducible interval. Then, each j is considered, in increasing order, and the corresponding left endpoint of the possible interval

is searched for on the stack S, which contains the positions of the candidates (both the positions and their corresponding elements in P^* are in increasing order). To this end, a value $B[i]$ is computed for each i, that upper bounds the values $|P^*[j]|$ obtained at a possible right end j of a conserved interval $(i..j)$. Obviously bad candidates $top(S)$ (too large or whose bound $B[top(S)]$ is exceeded by $|P^*[j]|$) are dropped, and the next candidate either is the suitable one (the number of elements in the interval is correct, and these elements are all smaller than $P^*[j]$), or is dropped.

The running time of this algorithm is in $O(n)$, since the *While* loop will globally unstack at most n elements (each index is stacked exactly once on S).

1.2.5 Common Intervals in Strings and the Element Plotting Strategy

In this section, P and Q are unsigned strings over $\{1, 2, \ldots, n\}$, of respective lengths m_P and m_Q, which implies that every element in the alphabet may have zero, one or several occurrences in each string. Without loss of generality, it is assumed that $n \leq m_P + m_Q$ (otherwise a renumbering of the elements in the alphabet may be performed to achieve this), and that strings P and Q are extended at their left and right extremities with a new element (not in Σ), say $n + 1$. To simplify explanations, the resulting strings are still noted P and Q with lengths m_P and m_Q (which only differ by 2 from the initial lengths, thus not affecting the complexity order of the algorithm below).

The algorithm presented in this section was proposed in [13] and uses a very different strategy to display all common intervals, compared to the ones in Sections 1.2.3 and 1.2.4. To start with, note that we may limit the searches to *maximal locations* of common intervals:

Definition 10 Let Q be an unsigned string on the alphabet $\Sigma = \{1, 2, \ldots, n\}$ and $\mathcal{C} \subseteq \{1, 2, \ldots, n\}$. Interval $Q[i, j]$ is a **location** of $\mathcal{C}$ in S if $\mathcal{CS}(Q[i, j]) = \mathcal{C}$. The location $Q[i, j]$ is **left-maximal** if $i = 1$ or $Q[i-1] \notin \mathcal{C}$, **right-maximal** if $j = m_Q$ or $Q[j+1] \notin \mathcal{C}$, and **maximal** if it is both left-maximal and right-maximal.

Algorithm 1.6 uses a vector POS and a matrix NUM to store respectively the positions in P of every element $c \in \{1, 2, \ldots, n\}$, and the number of distinct elements in each interval $P[a, b]$, with $a, b \in \mathcal{CS}(P)$. For each pair of indices i, j with $1 \leq i \leq j \leq m_Q$, such that $Q[i, j]$ is a maximal location of $\mathcal{CS}(Q[i, j])$ (set stored as a vector OCC in the algorithm), the n_{OCC} elements of $\mathcal{CS}(Q[i, j])$ are plotted (or marked) on P. Intervals of plotted elements on P are then tested to see if they have all the desired elements (that is, n_{OCC} elements) and if they are maximal. In the positive case, they are maximal common intervals and are thus output by the algorithm.

It is easy to imagine examples on which this algorithm will display twice (or more) the same common interval $\mathcal{C}$, with two different locations either on

Algorithm 1.6 Algorithm Common_Intervals_In_Strings [13]

```
{OCC[c] = 1 if and only if c belongs to the current interval Q[i,j]}
Compute data structures POS and NUM for P;
For i = 1 to m_Q do
   For c = 1 to n do OCC[c] ← 0 Endfor;
   n_OCC ← 0; j ← i;
   While j ≤ m_Q and Q[i,j] is left-maximal do
     c ← Q[j];
      OCC[c] ← 1; n_OCC ← n_OCC + 1;
     While Q[i,j] is not right-maximal do j ← j + 1 Endwhile;
     For all p in POS[c] do
       Mark element c at position p in P;
       P[a,b] ← the largest interval of marked characters with a ≤ p ≤ b;
       If NUM[a,b] = n_OCC and P[a,b] is maximal then
         Output C = CS(Q[i,j]) and the pair (P[a,b], Q[i,j])
       Endif
     Endfor;
     j ← j + 1
   Endwhile
EndFor.
```

P or on Q. The algorithm may be modified to avoid redundant output, as shown in [13].

This algorithm runs in $\Theta(m^2)$, where $m = \max\{m_P, m_Q\}$, but a variant of it exists to run in $O(m^2)$ [13].

1.2.6 Variants

The notions and algorithms presented so far are either directly devised for or easily extended to an arbitrary number $K \geq 2$ of genomes (see [3, 4, 13]). In this case, the complexity becomes $O(Kn + N)$ to output all common or all conserved intervals in (signed) permutations, $O(Kn)$ to compute the number of conserved intervals in signed permutations, and $O(Kn^2)$ to output all common intervals in strings.

The case of genomes with duplicates, represented by (signed or unsigned) strings, was approached in Section 1.2.5 under the double hypothesis that (1) no distinction can be made among duplicates in P and in Q, and that (2) the locations of a common interval of P and Q may contain an arbitrary number of copies of each gene. This approach passes up the underlining biological hypothesis that the copies of a gene are obtained during speciation and duplication processes, that imply relationships between duplicates. Several ways to express this biological hypothesis exist (see [7] for detailed explanations) resulting in different hard problems to solve, for which different approaches were proposed.

In our next section, we present some of them.

1.3 CHARACTER-BASED CRITERIA

1.3.1 Introduction and Definition of the Problems

As mentioned in the previous section, computing the number of breakpoints between two genomes that do not contain duplicates is an easy task. On the contrary, when duplicates occur in genomes, an intuitive way of dealing with them is to get back to permutations, that is genomes without duplicates. For this, the goal is to establish a one-to-one correspondence between genes of both genomes, that is a *matching*, say $\mathcal{M}$. Once $\mathcal{M}$ is found, we remove from both genomes the genes which are not matched by $\mathcal{M}$ (this happens, for instance, when the number of duplicates of a given gene differs between both genomes), and, after a renaming of the genes, we obtain a permutation, on which all classical measures can be computed.

The tricky part of the process is to find an appropriate matching. Usually, the matching $\mathcal{M}$ that we look for is one that optimizes the studied measure, thus following the parsimony hypothesis, which states that nature always chooses the "shortest path" to go from one species (i.e., one genome) to another.

In that case, the problem of computing a measure between two genomes, which was just a computation problem in permutations (we are just asked to provide a number), becomes an *optimization* problem in strings, in which one wants to find the matching that optimizes the studied measure.

In the following, we are interested in genomes that contain duplicates. We first look at the particular case in which the pairs of genomes that we compare contain, for each gene g, exactly the same number of copies of g. In that case, we say that genomes are *balanced*. This restriction could seem quite strong, but genes are DNA fragments, and no two genes are constituted of the exact same DNA sequence; thus, duplicate genes are actually genes that are pairwise *similar*, i.e. their DNA sequence are sufficiently close. Hence, it is always possible to build gene clusters such that there are as many copies in the first genome as in the second (e.g., by removing from a given cluster those genes that are less similar to the others).

Suppose that the two input genomes are balanced. It thus seems natural to ask for a one-to-one correspondence of the genes (i.e., a matching) that contains *all* the genes of both genomes. Such a matching is referred to in the following as a *full matching*.

Now, if the two input genomes are *unbalanced*, we need to define more precisely the matching that we look for. First, for any gene g, we denote by $\mathsf{occ}(g,P)$ (resp. $\mathsf{occ}(g,Q)$) the number of (positive and negative) occurrences of g in P (resp. Q). The required matching $\mathcal{M}$ thus needs to satisfy the following rule: for any gene g in P (resp. Q), $\mathcal{M}$ must contain $\min\{\mathsf{occ}(g,P),\mathsf{occ}(g,Q)\}$ one-to-one correspondences involving g. In that sense, $\mathcal{M}$ remains a *full matching*, because it contains the maximum possible number of one-to-one correspondences between genes. The difficulty here is the following: since

genomes are not balanced, some genes will remain unmatched by $\mathcal{M}$. In that case, we *prune* the genomes, i.e. we remove those unmatched genes, in order to obtain two genomes P' and Q' where each gene is covered by $\mathcal{M}$. We then compute the number of breakpoints between P' and Q', thanks to the permutation induced by $\mathcal{M}$.

We note, for sake of completeness, that there exists other types of matching that can be required:

- One can ask for an *exemplar* matching, in which we only keep one occurrence of each gene g [12]
- One can also ask for an *intermediate* matching, in which for each gene g, we keep $1 \leq x \leq \min\{\mathsf{occ}(g, P), \mathsf{occ}(g, Q)\}$ occurrences of g [1]

Coming back to the *full* matching variant, in both the balanced and unbalanced cases, finding a full matching that optimizes a given measure is NP-hard, and even APX-hard for all classical measures. This is for instance the case for minimizing the number of breakpoints [9], maximizing the number of common intervals [2] or the number of conserved intervals [2], and this hardness holds even for very restricted instances. Consequently, most of the efforts in the literature have focused on what seemed to be the "simplest" case, i.e. minimizing the number of breakpoints.

In the rest of this section, we describe two algorithms that deal with genomes P and Q, represented as strings of integers, and aim at finding a full matching $\mathcal{M}$ that minimizes the number of breakpoints in the permutation induced by $\mathcal{M}$ (possibly obtained after pruning, in case P and Q are not balanced). Let us denote this problem by `BAL-FMB`(P, Q) (`FMB` = Full Matching Breakpoints) in the balanced case, and `UNBAL-FMB`(P, Q) in the unbalanced case. The two algorithms we describe here are :

1. An approximation algorithm for `BAL-FMB` [10]
2. An exact (thus, exponential) algorithm for `UNBAL-FMB` [1], written in the form of a 0-1 linear program, the goal being to be able to handle large instances.

We want to emphasize the fact that this section does not aim at being an exhaustive survey of the results concerning `BAL-FMB` and `UNBAL-FMB`, but at providing different algorithmic techniques and results that we think can be of interest for the reader.

1.3.2 An Approximation Algorithm for `BAL-FMB`

In this section, we show the main ideas and arguments of an approximation algorithm provided by Kolman and Waleń [10]. Let P and Q be two balanced strings containing signed integers, let $n = m_P = m_Q$, and let k be the

maximum number of copies of a gene in P (resp. in Q). Note that, since a gene is represented by a signed integer, k takes into account both positive and negative occurrences of the most represented integer in P (resp. Q). The result from Kolman and Waleń [10] that we develop here is the following.

Theorem 1 *There exists an $O(k)$ approximation algorithm for solving the problem* `BAL-FMB`.

A slightly different problem: `UMCSP`. In order to make things simpler, we first develop the main arguments for the above theorem, in the specific case where the strings are *unsigned* (i.e., every integer in both strings carries the same sign, which we will always consider as positive). The algorithm can easily be adapted for instances containing signed strings, but with a loss of a factor 2 in the approximation ratio, which of course does not change the ratio of $O(k)$ given in Theorem 1.

It should be first said that the result from Kolman and Waleń [10] considers a slightly different problem than `BAL-FMB`, called `UMCSP`, which stands for Unsigned Minimum Common String Partition. This problem is the following: given two balanced unsigned strings P and Q, find a partition $\mathcal{P} = \{P_1, P_2, \dots P_t\}$ of substrings (i.e., sets of consecutive elements) of P such that:

1. $P_1 \cdot P_2 \cdot P_3 \dots P_t = P$, where $X \cdot Y$ denotes the concatenation of strings X and Y
2. there exists a permutation π on $\{1, 2, \dots t\}$ such that $Q = P_{\pi(1)} \cdot P_{\pi(2)} \cdot P_{\pi(3)} \dots P_{\pi(t)}$
3. t is minimized

In this context, each P_i, $1 \leq i \leq t$ is called a *block*. It can be seen that `BAL-FMB` and `UMCSP` are closely related in the following sense:

- any block partition of P and Q returned by `UMCSP` can be converted as a full matching between genes of P and genes of Q, and vice-versa.
- if b (resp. t) denotes the minimum number of breakpoints obtained by `BAL-FMB` (resp. the minimum number of blocks obtained by `UMCSP`), then b and t differ by 1. Thus, any approximation algorithm of ratio $O(k)$ for `UMCSP` is also an approximation algorithm of ratio $O(k)$ for `BAL-FMB`.

As a consequence, in the rest of the section, we only focus on `UMCSP`, keeping in mind that the result also applies to `BAL-FMB`.

Before going into further details, we need a few definitions: a *substring* of a string S is a set of consecutive elements of S. Recall that a *duo* in a string S is just a substring of length 2 of S, and let us denote by *duos(S)* the set of duos of S. Finally, given any solution for `UMCSP`, if a duo d from P does not appear in $\mathcal{P}$, the we say that d is *broken*.

A first approximation algorithm for UMCSP. The crucial idea behind the approximation algorithm from [10] is the following: in any solution for UMCSP, whenever a substring X appears a different number of times in P than in Q, then at least one duo in at least one occurrence of X must be broken. For any non empty string X, we denote by $\#substr(P, X)$ (resp. $\#substr(Q, X)$) the number of times X appears as a substring of P (resp. Q). Hence, the approximation algorithm ApproxUMCSP we look for could work as follows: for every X such that $\#substr(P, X) \neq \#substr(Q, X)$, cut at least one duo in each occurrence of X in P and Q, and return the partitions $\mathcal{P}$ and $\mathcal{Q}$ induced by those cuts. The correctness of ApproxUMCSP is given by the following lemma.

Lemma 4 *Algorithm* ApproxUMCSP *returns two partitions $\mathcal{P}$ and $\mathcal{Q}$ that form a common partition of P and Q.*

Note that we should try to avoid too many cuts of duos, since each cut of a duo corresponds to an increase in the number of blocks in $\mathcal{P}$ and $\mathcal{Q}$. However, minimizing the number of cuts is equivalent to the Hitting Set problem, which is known to be hard to approximate [11]. Thus, a deeper analysis is needed. Let T denote the set of all substrings $X \in \Sigma^*$ such that $\#substr(P, X) \neq \#substr(Q, X)$. Then, it can be seen that not all substrings $X \in T$ need to be considered. Indeed, if two substrings $X, Y \in T$ are such that $X \sqsubset Y$ (where $X \sqsubset Y$ here means "X is a proper substring of Y"), then any duo d that breaks an occurrence of X contained in Y will also break Y. Thus we can limit ourselves to the study of the set T_{min}, which is defined as follows:

$$T_{min} = \{X \in T | \nexists X' \in T \ s.t. \ X' \sqsubset X\}$$

In other words, T_{min} is the set of the substrings of T that are minimal with respect to the relation $\sqsubset$.

Thus, instead of going through T, going through T_{min} only in algorithm ApproxUMCSP maintains its correctness. Now, in order to determine the approximation ratio we look for, we need to analyze how T_{min} is involved in any optimal solution of UMCSP. Let the two partitions $(\mathcal{P}_{\mathcal{O}}, \mathcal{Q}_{\mathcal{O}})$ represent an optimal solution of UMCSP, and let an *optimal break* be any broken duo in this solution. The following lemma holds.

Lemma 5 *If $X \in T_{min}$, then there exists at least one occurrence of X in P or Q that contains an optimal break.*

Our goal is now to assign to each $X \in T_{min}$ an optimal break. For this, for any $X \in T_{min}$, we denote by $f(X)$ the optimal break that X contains. If X contains more than one optimal break, $f(X)$ is set arbitrarily to the leftmost one. In that case, the following lemma holds.

Lemma 6 *Let $X = X[1]X[2]\dots X[l]$ and Y be two strings from T_{min} such that $f(X) = f(Y)$. In that case, $duos(Y) \cap \{X[1]X[2], X[l-1]X[l]\} \neq \emptyset$.*

The above lemma leads us directly to the following modification of algorithm ApproxUMCSP: for each $X \in T_{min}$, we cut the first and last duo in all occurrences of X in P and Q. Algorithm ApproxUMCSP is now complete, and is summarized in Algorithm 1.7.

Algorithm 1.7 Algorithm ApproxUMCSP [10]

Input: Two balanced unsigned strings of integers, P and Q, each of length $n = m_P = m_Q$

1. Compute the set T_{min} defined in the text above
2. $\Phi = \emptyset$
3. $\mathcal{P} = \{P\}$, $\mathcal{Q} = \{Q\}$
4. **For each** $X \in T_{min}$ **do**
6. **If** $duos(X) \cap \Phi = \emptyset$ **then**
7. Add the first and last duo of X in Φ
8. Cut all occurrences of those two duos in the partitions $\mathcal{P}$ and $\mathcal{Q}$
9. **End If**
10. **End For**

Output: Partitions $\mathcal{P}$ and $\mathcal{Q}$

It can be seen that Algorithm ApproxUMCSP remains correct, because each occurrence of any $X \in T_{min}$ is cut by at least one duo, and Lemma 4 still holds. Moreover, the following theorem holds.

Theorem 2 *Algorithm* ApproxUMCSP *is a* $4k$*-approximation algorithm for* UMCSP.

The proof is as follows: suppose that X_1 and X_2 are two distinct strings of T_{min} that contributed to increasing the cardinality of the set Φ during the execution of ApproxUMCSP. Then, by Lemma 6, $f(X_1) \neq f(X_2)$. Since, on the whole, there are $Card(\mathcal{P}_\mathcal{O}) + Card(\mathcal{Q}_\mathcal{O}) - 2$ optimal breaks, this means that $Card(\Phi) \leq 2\,Card(\mathcal{P}_\mathcal{O}) + 2\,Card(\mathcal{Q}_\mathcal{O}) - 4$. Here, we consider instances where a given integer appears at most k times, thus each duo from Φ induces at most k cuts. Let $\mathcal{P}$ and $\mathcal{Q}$ be the partitions returned by ApproxUMCSP, and recall that, by definition, $Card(\mathcal{P}) = Card(\mathcal{Q})$ and $Card(\mathcal{P}_\mathcal{O}) = Card(\mathcal{Q}_\mathcal{O})$. We have $Card(\mathcal{P}) \leq k\,Card(\Phi) + 1$, thus $Card(\mathcal{P}) \leq 4k(Card(\mathcal{P}_\mathcal{O}) - 1) + 1$, that is $Card(\mathcal{P}) \leq 4k\,Card(\mathcal{P}_\mathcal{O})$.

About the time complexity of ApproxUMCSP. Kolman and Waleń have presented different tricks for achieving a time complexity of $O(n)$ for ApproxUMCSP, where $n = m_P = m_Q$.

First, instead of computing T_{min}, it is sufficient to compute a set T' of strings satisfying the three following properties:

1. $Card(T')$ is in $O(n)$ and can be computed in $O(n)$ time
2. $T_{min} \subseteq T' \subseteq T$

3. If a string $X \in T$ passes the test of Line **6.** of algorithm ApproxUMCSP (i.e., $duos(X) \cap \Phi = \emptyset$), then $X \in T_{min}$

In other words, T' is just a set that is easier to compute than T_{min}. Property 1. ensures it is not too large, and that it can be found efficiently. Properties 2. and 3. ensure that ApproxUMCSP remains correct using T' instead of T_{min}.

T' can actually be computed in $O(n)$ time using a suffix tree: let P and Q be the two balanced genomes from the instance. Then we build the (compact) suffix tree $\mathcal{T}$ of string $S = P\$_P Q\$_Q$, where $\$_P$ and $\$_Q$ are characters not appearing in P and Q. Such a suffix tree can be constructed in $O(n)$ time [15]. Let r be the root of $\mathcal{T}$, and let $v \neq r$ be any node of $\mathcal{T}$. Let $parent(v)$ be the father of v in $\mathcal{T}$, let $s(v)$ be the string represented by the path from r to $parent(v)$, and let $s'(v) = s(v) \cdot c$, where c is the first character of the string represented by the edge $\{parent(v), v\}$. Finally, we say that v is a *proper* node of $\mathcal{T}$ when $s'(v)$ contains neither $\$_P$ or $\$_Q$. Now we can define T': T' is the set of the strings $s'(v)$, for any proper node v in $\mathcal{T}$ for which $\#substr(P, s'(v)) \neq \#substr(Q, s'(v))$. One can see that $Card(T')$ is in $O(n)$, because $\mathcal{T}$ contains $O(n)$ nodes. Besides, $T_{min} \subseteq T'$ by definition. Finally, using the suffix tree, $\#substr(P, s'(v))$ (resp. $\#substr(Q, s'(v))$) can be computed in $O(n)$ time, and T' can thus be computed in $O(n)$ time as well.

Second, Kolman and Waleń describe how to maintain the set Φ, test the condition of Line **6.** of algorithm ApproxUMCSP, and realize the cuts in $O(1)$ time, leading to $O(n)$ overall. For this, they use a data structure from Gabow and Tarjan [8] that ensures an amortized $O(1)$ time for each such operation.

From `UMCSP` *to the signed case* `SMCSP`. The above description was focused on `UMCSP`, that is the unsigned case. If we want to adapt ApproxUMCSP to the signed case (a problem that we call `SMCSP`), a few adaptations need to be made. For any string S, let $-S$ denote its reversed string, both in order and sign. Then three main changes need to be done:

- $\#substr(P, X)$ should now count the number of occurrences of X and $-X$ in P
- The set T' should be computed using the suffix tree $\mathcal{T}'$ of string $S' = P\$_P Q\$_Q(-P)\$_P(-Q)\$_Q$ (where the brackets are here just to delimit strings)
- Whenever a duo ab should be cut, all duos $-b-a$ should be cut too

None of these adaptations changes the time complexity or the correctness of the algorithm. However, the last one increases the approximation ratio of ApproxUMCSP by a factor 2.

Remarks. First, we note that the above approximation algorithm ApproxUMCSP is actually a $\Theta(k)$ approximation algorithm, because there exist instances for which the optimum number of blocks is $O(1)$, whereas the number of blocks returned by the algorithm is $O(k)$. Strings $P = ba\{ab\}^{k-1}$ and $Q = ab^k$ [10] form such an instance: there exists a partition of P (resp. Q) containing 3 blocks, but ApproxUMCSP will return a solution containing $k+1$ blocks.

It should also be noted that, strangely enough, if instead of trying to minimize the number of breakpoints, we aim at finding a full matching that *maximizes the number of adjacencies*, then there exists a 4-approximation algorithm [2], i.e. an approximation ratio that does not depend on k. However, each problem is in some sense the dual of the other. This raises the question whether we can do better than an $O(k)$-approximation algorithm for `BAL-FMB`; and more precisely, is `BAL-FMB` $O(1)$-approximable ?

1.3.3 An Exact Algorithm for `UNBAL-FMB`

In this section, we focus on problem `UNBAL-FMB`, where the goal is to find a full matching $\mathcal{M}$ between two *unbalanced* signed genomes, in such a way that the permutation induced by $\mathcal{M}$ minimizes the number of breakpoints.

Here, we give the main elements of an exact algorithm that solves `UNBAL-FMB`, published in [1]. The problem is known to be APX-hard, even for instances in which P does not contain duplicates and occ(g,Q)≤ 2 for any gene g from Q [2]. Thus, the algorithm we give here is exponential; our approach is to express `UNBAL-FMB` into a 0-1 linear program, that is a series of inequalities implying boolean variables only, together with an objective function on boolean variables, that we wish to maximize.

The main interest in such an approach lies in the fact that there has been many efforts in the past to develop softwares that are able to handle such programs, even if they contain a large number of variables and inequalities. Thus, our hope is that powerful enough solvers (such as minisat+ or CPLEX) will be able to provide optimal solutions on real data in reasonable time. If this is the case, then we have two options:

- we can solve exactly those instances for themselves. However, even if this works for some data, one can easily imagine that there exists instances which will never be solved in reasonable time;

- or, thanks to exact results obtained by this method, we can analyze and evaluate one or several heuristic(s), as was done e.g. in [1].

As we will see later, we have tested our program on a set of 12 genomes of bacteria, for which all 66 pairwise comparisons were achieved rapidly.

We first present below the complete 0-1 linear program in itself (referred hereafter as `FMA` for Full Matching Adjacencies, a name that will be justified later), together with an explanation of the different variables we have defined

and used. Next, a few data reduction rules are provided, that aim at reducing the input size, and hence at speeding-up the program.

Program `FMA` takes as input two genomes P and Q with duplicates, of respective lengths m_P and m_Q, and solves problem `UNBAL-FMB`. Recall that Σ denotes the set of integers (representing genes) on which P and Q have been built. The objective function, the variables and the constraints involved are now discussed.

Variables:

- Variables $adj(i,j,k,\ell)$, $1 \leq i < j \leq m_P$ and $1 \leq k < \ell \leq m_Q$, represent *adjacencies* according to $\mathcal{M}$. Our initial problem is to try minimize the number of breakpoints; however, maximizing the number of adjacencies makes the writing of our 0-1 linear program more simple. Besides, it can been easily seen that since $Card(\mathcal{M})$ is given by the input, the full matching that minimizes the number of breakpoints also maximizes the number of adjacencies. Thus we only focus on adjacencies here, and $adj(i,j,k,\ell) = 1$ iff the three following propoerties are satisfied:
  1. One of the two following cases occur:
     - $(P[i], Q[k])$ and $(P[j], Q[\ell])$ belong to $\mathcal{M}$, $P[i] = Q[k]$ and $P[j] = Q[\ell]$, or
     - $(P[i], Q[\ell])$ and $(P[j], Q[k])$ belong to $\mathcal{M}$, $P[i] = -Q[\ell]$ and $P[j] = -Q[k]$
  2. $P[i]$ and $P[j]$ are consecutive in P according to $\mathcal{M}$
  3. $Q[k]$ and $Q[\ell]$ are consecutive in Q according to $\mathcal{M}$.
- Variables $a(i,k)$, $1 \leq i \leq m_P$ and $1 \leq k \leq m_Q$, define a matching $\mathcal{M}$: $a_{i,k} = 1$ iff $P[i]$ is matched with $Q[k]$ in $\mathcal{M}$.
- Variables $b_X(i)$, $X \in \{P, Q\}$ and $1 \leq i \leq m_X$, define whether the gene appearing at position i of X is covered by the matching $\mathcal{M}$. More precisely, $b_X(i) = 1$ iff $X[i]$ is covered by $\mathcal{M}$. Clearly, $\sum_{1 \leq i \leq m_P} b_P(i) = \sum_{1 \leq k \leq m_Q} b_Q(k)$, and this is precisely the size of $\mathcal{M}$.
- Variables $c_X(i,j)$, $X \in \{P, Q\}$ and $1 \leq i < j \leq m_X$, determine whether genes at positions i and j in X are *consecutive genes* according to $\mathcal{M}$: $c_X(i,j) = 1$ iff $X[i]$ and $X[j]$ are both covered by $\mathcal{M}$ and no gene $X[p]$, $i < p < j$, is covered by $\mathcal{M}$.

Constraints:

Assume $1 \leq i < j \leq m_P$ and $1 \leq k < \ell \leq m_Q$.

- Constraint `C.01` ensures that each gene of P and of Q is matched at most once, i.e. $b_P(i) = 1$ (resp. $b_Q(k) = 1$) iff gene i (resp. k) is matched in P (resp. Q). Observe that in any matching, any two genes that are mapped together necessarily have the same label (except maybe for the

Program FMA

Objective :

Maximize $\sum_{1\le i<m_P}\ \sum_{i<j\le m_P}\ \sum_{1\le k<m_Q}\ \sum_{k<\ell\le m_Q} adj(i,j,k,\ell)$

Constraints :

C.01 $\forall\ 1 \le i \le m_P,\ \sum_{\substack{1\le k\le m_Q \\ |P[i]|=|Q[k]|}} a(i,k) = b_P(i)$

$\forall\ 1 \le k \le m_Q,\ \sum_{\substack{1\le i\le m_P \\ |P[i]|=|Q[k]|}} a(i,k) = b_Q(k)$

C.02 $\forall\ X \in \{P,Q\}, \forall\ g \in \Sigma,\ \sum_{\substack{1\le i\le m_X \\ |X[i]|=|g|}} b_X(i) = \min(\mathsf{occ}(g,P), \mathsf{occ}(g,Q))$

C.03 $\forall\ X \in \{P,Q\}, \forall\ 1 \le i \le j-1 < m_X,\ c_X(i,j) + \sum_{i<s<j} b_X(s) \ge 1$

C.04 $\forall\ X \in \{P,Q\}, \forall\ 1 \le i < s < j \le m_X,\ c_X(i,j) + b_X(s) \le 1$

C.05 $\forall\ 1 \le i < j \le m_P, \forall\ 1 \le k < \ell \le m_Q,$
such that $P[i] = Q[k]$ and $P[j] = Q[\ell]$,
$a(i,k) + a(j,\ell) + c_1(i,j) + c_2(k,\ell) - adj(i,j,k,\ell) \le 3$

C.06 $\forall\ 1 \le i < j \le m_P, \forall\ 1 \le k < \ell \le m_Q,$
such that $P[i] = Q[k]$ and $P[j] = Q[\ell]$,
$a(i,k) - adj(i,j,k,\ell) \ge 0$
$a(j,\ell) - adj(i,j,k,\ell) \ge 0$
$c_1(i,j) - adj(i,j,k,\ell) \ge 0$
$c_2(k,\ell) - adj(i,j,k,\ell) \ge 0$

C.07 $\forall\ 1 \le i < j \le m_P, \forall\ 1 \le k < \ell \le m_Q,$
such that $P[i] = -Q[\ell]$ and $P[j] = -Q[k]$,
$a(i,\ell) + a(j,k) + c_1(i,j) + c_2(k,\ell) - adj(i,j,k,\ell) \le 3$

C.08 $\forall\ 1 \le i < j \le m_P,\ \forall\ 1 \le k < \ell \le m_Q,$
such that $P[i] = -Q[\ell]$ and $P[j] = -Q[k]$,
$a(i,\ell) - adj(i,j,k,\ell) \ge 0$
$a(j,k) - adj(i,j,k,\ell) \ge 0$
$c_1(i,j) - adj(i,j,k,\ell) \ge 0$
$c_2(k,\ell) - adj(i,j,k,\ell) \ge 0$

C.09 $\forall\ 1 \le i < j \le m_P,\ \forall\ 1 \le k < \ell \le m_Q,$
such that $\{|P[i]|, |P[j]|\} \ne \{|Q[k]|, |Q[\ell]|\}$ or $P[i] - P[j] \ne Q[k] - Q[\ell]$,
$adj(i,j,k,\ell) = 0$

C.10 $\forall\ 1 \le i < j \le m_P,$
$\sum_{1\le k<m_Q}\ \sum_{k<\ell\le m_Q} adj(i,j,k,\ell) \le 1$

Domains :

$\forall\ 1 \le i < j \le m_P, \forall\ 1 \le k < \ell \le m_Q,$
$a(i,k),\ b_P(i),\ b_Q(i),\ c_P(i,k),\ c_Q(i,k),\ adj(i,j,k,\ell) \in \{0,1\}$

Figure 1.2 Program FMA solves exactly UNBAL-FMB

sign), and hence we do not have to explicitly ask for $a(i,k) = 0$ in case $P[i]$ and $Q[k]$ are two different genes.

- Constraint `C.02` actually defines the fact that we ask for a full matching. For each gene g, $\min(\mathsf{occ}(g,P),\mathsf{occ}(g,Q))$ occurrences of g must be covered by $\mathcal{M}$ in both P and Q.

- Constraints in `C.03` and `C.04` are concerned with our definition of consecutive genes. Variable $c_X(i,j)$ is equal to 1 iff there exists no p such that $i < p < j$ and $b_X(p) = 1$. It is worth noticing here that, according to these constraints, one may have $c_X(i,j) = 1$ even if one of the genes $X[i]$ or $X[j]$ is *not* covered by $\mathcal{M}$.

- Constraints in `C.05` to `C.10` define variables adj. In the case where $P[i] = Q[k]$ and $P[j] = Q[\ell]$, Constraints `C.05` and `C.06` ensure that we have $adj(i,j,k,\ell) = 1$ if and only if all variables $a(i,k)$, $a(j,\ell)$, $c_1(i,j)$ and $c_2(k,\ell)$ are equal to 1. In the case where $P[i] = -Q[\ell]$ and $P[j] = -Q[k]$, Constraints `C.07` and `C.08` ensure that we have $adj(i,j,k,\ell) = 1$ iff all variables $a(i,\ell)$, $a(j,k)$, $c_1(i,j)$ and $c_2(k,\ell)$ are equal to 1. Constraint `C.09` sets variable $adj(i,j,k,\ell)$ to 0 if none of the two above cases holds. Finally, thanks to Constraint `C.10`, one must have at most one adjacency for every pair (i,j).

The objective of Program `FMA` is to maximize the number of adjacencies between the two considered genomes. According to the above, this objective thus reduces in our model to maximizing the sum of all variables $adj(i,j,k,\ell)$.

Speeding-up the program. Program `FMA` has $O((m_P\, m_Q)^2)$ variables and $O((m_P\, m_Q)^2)$ constraints. In order to speed-up the execution of the program, there are some simple rules to apply for reducing the number of variables and constraints in `FMA`.

First, the genomes are pairwise pre-processed to delete all genes that do not appear in both genomes, since we know that no full matching will contain them.

Second, for any gene g for which $\mathsf{occ}(g,P) = \mathsf{occ}(g,Q) = 1$ and $|P[i]| = |Q[k]| = g$, the corresponding variable $a_{i,k}$ is directly set to 1, as well as the two variables $b_P(i)$ and $b_Q(k)$.

Also, if two genes appearing only once in each genome occur consecutively or in reverse order with opposite signs, the corresponding variable adj is directly set to 1, and the related constraints are discarded.

Finally, if for two genes, say occurring at positions i and j in P, at least one gene g occurring between position i and j in P must be covered in any matching $\mathcal{M}$ (for example if all occurrences of g appear between i and j in P), then the corresponding variable $c_P(i,j)$ and the variables $adj(i,j,k,\ell)$ for all $1 \leq k < \ell \leq m_Q$ are set directly to 0 and the related constraints are discarded. Of course, the same reasoning applies for two positions k and ℓ in Q and the variables $c_Q(k,\ell)$ and $adj(i,j,k,\ell)$ for all $1 \leq i < j \leq m_P$.

Remarks. FMA has been tested on 12 genomes of γ-Proteobacteria (a subfamily of bacteria), which contain from 564 to 5540 genes (3104 on average). This led to 66 pairwise comparisons, that were achieved within 2 minutes (leading to an average of 1.7 s per comparison) using the solver CPLEX [1].

FMA works for unbalanced genomes, and can greatly be simplified in order to be adapted to balanced ones: more precisely, some variables and thus some constraints do not need to exist anymore. For instance, if P and Q are balanced and of length n, $b_P(i)$ (resp. $b_Q(i)$) is set to 1 for any $1 \leq i \leq n$, and $c_P(i,j)$ (resp. $c_Q(i,j)$) are unnecessary. The same goes for some constraints such as (C.03) and (C.04).

1.3.4 Other Results and Open Problems

Balanced Case. Apart from the approximation algorithm given in Section 1.3.2, the main recent result is a Fixed-Parameter Tractability algorithm for UMCSP by Damaschke [6]. More precisely, the main result from [6] is the following: there exists a Fixed-Parameter Tractability algorithm for UMCSP on P and Q, whose exponential running time involves only parameters b and r, where:

- b is the minimum number of blocks in an optimal solution for UMCSP on P and Q, and
- r is the *repetition number* of P, that is the maximum i such that $P = X \cdot Y^i \cdot Z$ for some strings X, Y and Z, where Y is non empty.

Two main open problems remain:

1. Does there exist an approximation algorithm of ratio $O(1)$ for SMCSP ?
2. Is UMCSP (resp. SMCSP) Fixed-Parameter Tractable on b only ?

Unbalanced Case. In this case, to our knowledge, no positive result (Polynomial Time Approximation Scheme, approximation algorithm or FPT algorithm) is known for UNBAL-FMB, even for restricted cases. In that sense, the field is totally open.

We note though, that a related problem, called ZMBD, has been shown to be polynomial in [2]. ZMBD is the following decision problem: given two signed unbalanced genomes, determine whether there exists a full matching $\mathcal{M}$ such that the number of breakpoints in the permutation induced by $\mathcal{M}$ is equal to zero.

1.4 CONCLUSION

In this chapter, we have presented different algorithmic techniques that deal with comparing pairs of genomes in order to infer (dis)similarity measures

between them. The two main types of measures that have been studied were: (i) common and conserved intervals in the first part, and (ii) breakpoints and adjacencies in the second part.

We intentionally did not provide a survey of all existing results on the topic, but we have chosen to focus only on few algorithms, in order to show the different techniques and ideas that lie behind those algorithms. We think and hope this could be of interest for the reader. It also allowed us to show a sample of the ideas that have been developed in the algorithmic field of comparative genomics.

As can be seen in this chapter, all of the above mentioned measures can be computed in polynomial time whenever genomes are permutations. On the contrary, when genomes contain duplicates and in case a matching is required, all measures are hard to compute, and even hard to approximate, even in very restricted cases. It can be seen, however, that when genomes are balanced, some positive results exist, in the form of approximation and FPT algorithms, in the full matching case.

The most challenging remaining open questions in this domain are probably those that ask for positive results for comparing *unbalanced* genomes, using a matching and any of the above mentioned measures.

REFERENCES

1. S. Angibaud, G. Fertin, I. Rusu, A. Thévenin, S. Vialette - Efficient Tools for Computing the Number of Breakpoints and the Number of Adjacencies between two Genomes with Duplicate Genes, *Journal of Computational Biology* 15(8): 1093-1115 (2008).
2. S. Angibaud, G. Fertin, I. Rusu, A. Thévenin, S. Vialette - On the Approximability of Comparing Genomes with Duplicates, *Journal of Graph Algorithms and Applications* 13(1): 19-53 (2009).
3. A. Bergeron, C. Chauve, F. de Montgolfier, M. Raffinot - Computing Common Intervals of K Permutations, with Applications to Modular Decomposition of Graphs, *SIAM J. Discrete Math.* 22(3): 1022-1039 (2008).
4. A. Bergeron, J. Stoye - On the Similarity of Sets of Permutations and Its Applications to Genome Comparison, *J. Comput. Biol.* 13(7): 1340-1354 (2006).
5. X. Chen, J. Zheng, Z. Fu, P. Nan, Y. Zhong, S. Lonardi - Assignment of orthologous genes via genome rearrangement, *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 2(4): 302-315 (2005).
6. P. Damaschke - Minimum Common String Partition Parameterized, *Proceedings of WABI 2008*, LNCS 5251, 87-98 (2008).
7. G. Fertin, A. Labarre, I. Rusu, E. Tannier, S. Vialette - Combinatorics of Genome Rearrangements, MIT Press, 2009.
8. H.N. Gabow, R.E. Tarjan - A Linear-Time Algorithm for a Special Case of Disjoint Set Union, *Journal of Computer and System Sciences* 30(2): 209-221 (1985).

9. A. Goldstein, P. Kolman, J. Zheng - Minimum Common String Partition Problem: Hardness and Approximations, *The Electronic Journal of Combinatorics* 12(1): paper R50 (2005).

10. P. Kolman, T. Waleń - Reversal Distance for Strings with Duplicates: Linear Time Approximation using Hitting Set, *The Electronic Journal of Combinatorics* 14(1): R50 (2007).

11. R. Raz, S. Safra - A sub-constant error-probability low-degree test, and sub-constant error-probability PCP characterization of NP, *Proceedings of STOC 97*, ACM, 475-484 (1997).

12. D. Sankoff - Genome rearrangement with gene families, *Bioinformatics* 15(11):909-917 (1999).

13. T. Schmidt, J. Stoye - Quadratic Time Algorithms for Finding Common Intervals in Two and More Sequences, *Proceedings of CPM 2004*, LNCS 3109, 347-358 (2004).

14. A. Schrijver - Theory of Linear and Integer Programming, *John Wiley and Sons* (1998).

15. E. Ukkonen - On-line construction of suffix trees, *Algorithmica* 14(3):249-260 (1995).

16. T. Uno and M. Yagiura - Fast Algorithms to Enumerate All Common Intervals of Two Permutations, *Algorithmica* 26, 290-309 (2000).