An Algorithmic View of Gene Teams

Marie-Pierre Béal* Anne Bergeron † Sylvie Corteel‡ Mathieu Raffinot§

Abstract

Comparative genomics is a growing field in computational biology, and one of its typical problem is the identification of sets of orthologous genes that have virtually the same function in several genomes. Many different bioinformatics approaches have been proposed to define these groups, often based on the detection of sets of genes that are "not too far" in all genomes. In this paper, we propose a unifying concept, called *gene teams*, which can be adapted to various notions of distance. We present two algorithms for identifying gene teams formed by n genes placed on m linear chromosomes. The first one runs in $O(mn \log^2 n)$ and uses a divide and conquer approach based on the formal properties of gene teams. We next propose an optimization of the original algorithm, and, in order to better understand the complexity bound of the algorithms, we recast the problem in the Hopcroft's partition refinement framework. This allows us to analyze the complexity of the algorithms with elegant amortized techniques. Both algorithms require linear space. We also discuss extensions to circular chromosomes that achieve the same complexity.

Résumé

La comparaison des génomes est un domaine croissant en biologie computationnelle et l'un de ses problèmes typiques est l'identification d'ensembles de gènes orthologues qui ont virtuellement la même fonction dans plusieurs génomes. Plusieurs approches bio-informatiques distinctes ont été proposées pour définir ces groupes. Elles sont souvent basées sur la détection d'ensembles de gènes qui ne sont pas "trop éloignés" dans tous les génomes considérés. Dans cet article, nous proposons un concept unificateur, appelé *équipe de gènes*, qui peut être adapté à différentes notions de distances. Nous présentons deux algorithmes pour identifier les équipes de gènes formées par n gènes situés sur m chromosomes linéaires. Le premier a une complexité en temps de $O(mn \log^2 n)$ et utilise une approche "diviser pour régner" basée sur des propriétées formelles des équipes de gènes. Nous proposons ensuite une optimisation de cet algorithme, et, afin de mieux comprendre la borne sur sa complexité, nous replaçons le problème dans le cadre d'un schéma de raffinement de partitions de Hopcroft. Ceci nous permet d'analyser la complexité par des techniques plus élégantes de complexité amortie. Les deux algorithmes ont une complexité en espace linéaire. Nous considérons également des extensions au cas des chromosomes circulaires qui ont la même complexité.

*Institut Gaspard-Monge, Université de Marne-la-Vallée, Cité Descartes, Champs-sur-Marne, 77454 Marne-la-Vallée Cedex 2, France. Marie-Pierre.Beal@univ-mlv.fr.

†LaCIM, Université du Québec à Montréal, Canada. anne@lacim.uqam.ca

‡CNRS - Laboratoire PRiSM, Université de Versailles, 45 Avenue des Etats-Unis, 78035 Versailles cedex, France. E-mail: syl@prism.uvsq.fr

§CNRS - Laboratoire Génome et Informatique, Tour Evry 2, 523, Place des Terrasses de l'Agora, 91034 Evry, France. raffinot@genopole.cnrs.fr

1 Introduction

In the last few years, research in genomics science evolved rapidly. More and more complete genomes are now available due to the development of semi-automatic sequencer machines. Many of these sequences – particularly prokaryotic ones – are well annotated: the position of their genes are known, and sometimes parts of their regulation or metabolic pathways.

A new computational challenge is to extract gene or protein knowledge from high level comparison of genomes. For example, the knowledge of sets of orthologous or paralogous genes on different genomes helps to infer putative functions from one genome to the other. Many researchers have explored this avenue, trying to identify groups or clusters of orthologous genes that have virtually the same function in several genomes [1, 6, 5, 7, 9, 11, 13, 14, 15, 18]. These researches are often based on a simple, but biologically verified fact, that proteins that interact are often coded by genes closely placed in the genomes of different species. With the knowledge of the positions of genes, it becomes possible to automate the identification of groups of closely placed genes in several genomes. For a more complete biologically oriented discussion on these groups of genes, we refer the reader to [12].

From an algorithmic and combinatorial point of view, the formalizations of the concept of *closely placed genes* are still fragmentary, and sometimes confusing. The distance between genes is variously defined as differences between physical locations on a chromosome, distance from a specified target, or as a discrete count of intervening actual or predicted genes. The algorithms often lack the necessary grounds to prove their correctness, or assess their complexity. This paper contributes to a research movement of clarification of these notions. We aim to formalize, in the simplest and most comprehensive ways, the concepts underlying the notion of distance-based clusters of genes. We can then make use of these concepts, and their formal properties, to design sound and efficient algorithms.

A first step in that direction has been done in [9, 19] with the concept of *common intervals*. A common interval is a set of orthologous genes that appear consecutively, possibly in different orders, on a chromosome of two or more species. This concept covers simplest cases of sets of closely placed genes, but does not take in account the nature of the gaps between genes. Common intervals can be defined on chromosomes with paralogous genes, that is, each gene could have multiple locations on the chromosomes. However, the algorithms in [9, 19] are designed only for the case where each gene occurs once on each chromosome.

In this paper, we extend this notion by relaxing the "consecutive" constraint. We assume that each gene occurs once on each chromosome. We allow genes to be separated by gaps that do not exceed a fixed threshold. We develop a simple formal setting for these concepts, and give two polynomial algorithms that detect maximal sets of closely placed genes, called *gene teams*, in m chromosomes. Note that we focus in this paper on the algorithmic part of the gene team concept. A complete study validating this model from a biological point of view is available in [12], an the results concerning the divide-and-conquer algorithm were announced in [3].

The first algorithm refines the partitions induced by gene chains of two or more chromosomes. It uses a divide-and-conquer approach based on the existence of small classes of the partitions. The apparent simplicity hides a complex underlying problem that first appeared in the non trivial complexity of this first algorithm.

Next, in order to better understand the complexity bounds, and analysis, of this algorithm, we recast the problem in the Hopcroft's partition refinement framework [10], which covers a wide range of applications [8, 16]. We develop a new algorithm based of the first Hopcroft minimization algorithm, and show that the first algorithm described is a cleverly disguised Hopcroft-like algorithm. The close links between the two algorithms allows us to derive an elegant complexity analysis, based on amortized techniques, which is much more intuitive than the equational approach. Moreover, the fact that Hopcroft-like algorithms have been extensively studied confirms the intrinsic difficulties of the gene teams identification problem.

This paper is organized as follows. In Section 2, we formalize the concept of *gene teams* that unifies most of the current approaches, and discuss their basic properties. In Section 3 we present two algorithms that identify the gene teams of two chromosomes. The links between Hopcroft's partitioning framework and gene teams identification are explored in Section 4. Finally, in Section 5, we extend our algorithms to m chromosomes, and to circular chromosomes. An extended abstract of this paper appeared in [3].

2 Gene Teams and their Properties

Much of the following definitions refer to sets of *genes* and *chromosomes*. These are biological concepts whose definitions are outside the scope of this paper. However, we will assume some elementary formal properties relating genes and chromosomes: a chromosome is an ordering device for genes that belong to it, and a gene can belong to several chromosomes. If a gene belongs to a chromosome, we assume that its position is known, and unique.

2.1 Definitions and Examples

Let Σ be a set of n genes that belong to a chromosome C, and let P_C be a function:

$$\Sigma \xrightarrow{P_C} R$$

that associates to each gene g in Σ a real number$P_C(g)$, called its *position*.

Functions of this type are quite general, and cover a wide variety of applications. The position can be, as in [14, 15, 11], the physical location of an actual sequence of nucleotides on a chromosome. In more qualitative studies, such as [1, 13], the positions are positive integers reflecting the relative ordering of genes in a given set. In other studies [5], positions are both negative and positive numbers computed in relation to a target sequence.

The function P_C induces a permutation on any subset S of Σ, ordering the genes of S from the gene of lowest position to the gene of highest position. We will denote the permutation corresponding to the whole set Σ by π_C. If g and g' are two genes in Σ, their *distance* $\Delta_C(g, g')$ in chromosome C is given by $|P_C(g') - P_C(g)|$.

For example, if $\Sigma = \{a, b, c, d, e\}$, consider the following chromosome X, in which genes not in Σ are identified by the star symbol:

$$X \;=\; c \;\; * \;\; * \;\; e \;\; d \;\; a \;\; * \;\; b.$$

Define $P_X(g)$ as the number of of genes appearing to the left of g. Then $\Delta_X(c, d) = |P_X(d) - P_X(c)| = 4$, $\pi_X = (c\ e\ d\ a\ b)$, and the permutation induced on the subset $\{a, c, e\}$ is $(c\ e\ a)$.

Definition 1 *Let S be a subset of Σ, and $(g_1 \ldots g_k)$ be the permutation induced on S on a given chromosome C. For $\delta > 0$, the set S is called a δ-*chain *of chromosome C if $\Delta_C(g_j, g_{j+1}) \le \delta$, for $1 \le j < k$.*

For example, if $\delta = 3$, then $\{a, c, e\}$ is a δ-chain of X, since each pair of consecutive elements in the permutation $(c\ e\ a)$ is distant by less than δ.

We will also refer to *maximal* δ-chains with respect to the partial order induced on the subsets by the inclusion relation. For example, with $\delta = 2$, the maximal δ-chains of X are $\{c\}$ and $\{a, b, d, e\}$. Note that singletons are always δ-chains, regardless of the value of δ.

Definition 2 *A subset S of Σ is a* δ-set *of chromosomes C and D if S is a δ-chain both in C and D. A* δ-team *of the chromosomes C and D is a maximal δ-set with respect to inclusion. A δ-team with only one element is called a* lonely gene.

Consider, for example, the two chromosomes:

$$\begin{array}{lcllllllll} X & = & c & * & * & e & d & a & * & b \\ Y & = & a & b & * & * & * & c & * & d & e. \end{array}$$

For $\delta = 3$ then $\{d, e\}$ and $\{c, d, e\}$ are δ-sets, but not $\{c, d\}$ since the latter is not a δ-chain in X. The δ-teams of X and Y, for values of δ from 1 to 4 are given in the following table.

δ	δ-teams	Lonely Genes
1	$\{d, e\}$	$\{a\}, \{b\}, \{c\}$
2	$\{a, b\}, \{d, e\}$	$\{c\}$
3	$\{a, b\}, \{c, d, e\}$	
4	$\{a, b, c, d, e\}$	

Note that two gene teams can overlap. For instance, if $X = a\ c\ b\ d$, $Y = a\ b\ *\ *\ c\ d$ and $\delta = 2$, then $\{a, b\}$ and $\{c, d\}$ are two overlapping gene teams.

Our goal is to develop algorithms for the efficient identification of gene teams. The main pitfalls are illustrated in the next two examples.

The intersection of δ-chains is not always a δ-set. A naive approach to construct δ-sets is to identify maximal δ-chains in each sequence, and intersect them. Although this works on some examples, the approach does not hold in the general case. For example, in the chromosomes:

$$\begin{array}{lcllllll} X & = & a & b & c \\ Y & = & a & c & * & * & b, \end{array}$$

with $\delta = 1$, the maximal δ-chain of X is $\{a, b, c\}$, and the maximal δ-chains of Y are $\{a, c\}$ and $\{b\}$. But $\{a, c\}$ is not a δ-team.

Gene teams cannot be grown from smaller δ-sets. A typical approach for constructing maximal objects is to start with initial objects that have the desired property, and cluster them with a suitable operation. For gene teams, the singletons are perfect initial objects, but there is no obvious operation that, applied to two small δ-sets, produces a bigger δ-set. Consider the following chromosomes:

$$\begin{array}{lcllll} X & = & a & b & c & d \\ Y & = & c & a & d & b & . \end{array}$$

For $\delta = 1$, the only δ-sets are the sets $\{a\}$, $\{b\}$, $\{c\}$ and $\{d\}$, and the set $\{a, b, c, d\}$. In general, it is possible to construct pairs of chromosomes with an arbitrary number of genes, such that the only δ-sets are the singletons and the whole set. For example, consider the following chromosomes, in which the genes are represented by numbers in order to illustrate the construction:

$$\begin{array}{rclllllllll} X & = & 1 & 2 & 3 & \ldots & & & & \ldots & 2k \\ Y & = & 2 & 4 & 6 & \ldots & 2k & 1 & 3 & 5 & \ldots & 2k-1. \end{array}$$

For $\delta = 1$, any δ-set larger than a singleton must contain both odd and even genes because they alternate in chromosome X, but any δ-chain in Y that contains odd and even genes must contain genes 1 and $2k$, implying that the only team with more than one gene is the whole set.

Instead of growing teams from smaller δ-sets, we will extract them from larger sets that contain only teams. This leads to the following definition:

Definition 3 *A δ-*league *of chromosomes C and D is a union of δ-teams of the chromosomes C and D.*

As the two last examples show, the combinatorial properties of δ-sets are not elementary, and we need to establish them in order to develop and prove our algorithms.

2.2 Properties of δ-sets and teams.

The first crucial property of δ-teams is that they form a partition of the set of genes Σ. It is a consequence of the following lemma:

Lemma 1 *If S and T are two δ-chains of chromosome C, and $S \cap T \neq \emptyset$, then $S \cup T$ is also a δ-chain.*

Proof: Consider the permutation induced on the set $S \cup T$, and let g and g' be two consecutive elements in the permutation. If g and g' both belong to S (or to T), then they are consecutive in the permutation induced by S (or by T), and $\Delta(g, g') \leq \delta$. If g is in S but not in T, and g' is in T but not in S, then either g is between two consecutive elements of T, or g' is between two consecutive elements of S. Otherwise, the two sets S and T would have an empty intersection. If g is between two consecutive elements of T, for example, then one of them is g', implying $\Delta(g, g') \leq \delta$. ■

We now have easily:

Proposition 1 *For a given set of genes Σ, the δ-teams of chromosomes C and D form a partition of the set Σ.*

Proof: Since any singleton of Σ is a δ-set, any gene of Σ belongs to a δ-team. If the intersection of two different δ-teams T_1 and T_2 is not empty, then the intersection of the two underlying δ-chains is not empty neither in C nor in D, therefore their union is also a δ-chain in both sequences, implying that $T_1 \cup T_2$ is a δ-set, and contradicting the maximality of T_1 and T_2. ■

Proposition 1 has the following corollary:

Corollary 1 *If a set S is both a league, and a δ-set, of chromosomes C and D, then S is a δ-team.*

Proof: Since the maximal δ-sets form a partition of Σ, any δ-set is contained in a unique δ-team. ■

The algorithms described in the next section work on leagues, splitting them while ensuring that a league is split in smaller leagues. The process stops when each league is a δ-set. Corollary 1 provides a simple proof that such an algorithm correctly identifies the teams. The next proposition gives the "initial" leagues for the first algorithm.

Proposition 2 *Any maximal δ-chain of C or of D is a league.*

Proof: First observe that the set of maximal δ-chains in a chromosome also forms a partition of Σ. Therefore, any δ-chain is included in a unique maximal δ-chain. If T is a team of C and D, then T is a δ-chain in both chromosomes, thus T is included in a single maximal chain in both chromosomes. ■

3 Algorithms to Find Gene Teams

It is quite straightforward to develop $O(n^2)$ algorithms that find gene teams in two chromosomes. In the following subsection, we present some of the pitfalls of naive approaches to partition refinement that can lead to an $O(n^2)$ worst case scenario. However, since the ultimate goal is to be able to upgrade the definitions and algorithms to more than two chromosomes, such a threshold is too high. In Section 3.2, we develop an $\mathcal{O}(n \log^2 n)$ algorithm, whose complexity is analysed in section 3.3. We then propose in Section 3.4 an optimization of the first algorithm, reducing its time complexity to $O(n \log n \log \delta')$, where δ' is, for all pratical purpose, a small constant.

3.1 Partition Refinement

Assume that we are given two permutations on Σ, π_C and π_D, each already partitioned into maximal δ-chains of chromosomes C and D:

$$\begin{aligned}\pi_C &= (c_1 \ldots c_{k_1})(c_{k_1+1} \ldots c_{k_2}) \ldots (c_{k_s+1} \ldots c_n) \\ \pi_D &= (d_1 \ldots d_{l_1})(d_{l_1+1} \ldots d_{l_2}) \ldots (d_{l_t+1} \ldots d_n).\end{aligned}$$

Let $(c_i \ldots c_j)$ be one of the classes of the partition of π_C, by Proposition 2 $(c_i \ldots c_j)$ is a league. Our goal is to split this class in v subclasses $S_1, \ldots, S_v$ such that: a) each subclass is a league; b) each subclass is a δ-chain in C; and c) each subclass is contained in one of the classes of π_D.

Consider, for example, the following two chromosomes – in which we identified the genes as numbers, and $k \geq 1$:

$$\begin{array}{rcllllllllllll} X &=& (3 & 1 & 5 & 2 & 7 & 4 & 9 & \ldots & 2k+1 & 2k-2 & 2k+3 & 2k) \quad (2k+2) \\ Y &=& (1 & 2 & 3 & 4 & 5 & & & \ldots & & 2k+1 & 2k+2 & 2k+3). \end{array}$$

If one compares the first league of chromosome X to the first league of chromosome Y, one can observe that genes $2k+2$ and $2k+3$ must be isolated in both partitions. But the resulting problem

$$\begin{array}{lcl} X' & = & (3\ \ 1\ \ 5\ \ 2\ \ 7\ \ 4\ \ 9\ \ \ldots\ \ 2k+1\ \ 2k-2)\ \ (2k+3)\ \ \ \ (2k)\ \ \ \ (2k+2) \\ Y' & = & (1\ \ 2\ \ 3\ \ 4\ \ 5\ \ \ \ \ \ \ \ \ldots\ \ \ \ \ \ \ \ 2k+1)\ \ (2k+2)\ \ (2k+3), \end{array}$$

has the same form tas the original one, showing that a bad choice of leagues to compare can yield to $O(n)$ iterations of the process. This partition refinement approach has the drawback that big leagues must be read over and over again, in order to extract the small leagues that are buried in them. In the next section, we take the point of view of the small classes, and show that their extraction can be done efficiently.

3.2 A Divide-and-Conquer Algorithm

The following algorithm to identify teams is a divide-and-conquer algorithm that works by extracting small leagues from larger ones. Its basic principle is described in the following paragraph.

Assume that S is a league of chromosomes C and D, and that the genes of S are respectively ordered in C and D as:

$$(c_1 \ldots c_n), \text{ and } (d_1 \ldots d_n).$$

By Proposition 1, if S is a δ-set, then S is a δ-team. If S is not a δ-set, there are at least two consecutive elements, say c_i and c_{i+1} that are distant by more than δ. Therefore, both $(c_1 \ldots c_i)$ and $(c_{i+1} \ldots c_n)$ are leagues, splitting the initial problem in two sub-problems. The following two lemmas explain how to split a problem efficiently.

Lemma 2 *If S is a league, but not a team, of chromosomes C and D, then there exists a sub-league of S with at most $|S|/2$ genes.*

Proof: Let $|S| = n$, if all sub-leagues of S have more than $n/2$ genes, it follows that each team included in S has more than $\lfloor n/2 \rfloor$ genes, and the intersection of two such teams cannot be empty. ∎

The above lemma implies that if S is a league, but not a team, and if the sequences $(c_1 \ldots c_n)$ and $(d_1 \ldots d_n)$ are the corresponding permutations in chromosomes C and D, then there exist a value $p \leq n/2$ such that at least one of the following sequences is a league:

$$\begin{array}{l} (c_1 \ldots c_p), \\ (c_{n-p+1} \ldots c_n), \\ (d_1 \ldots d_p), \\ (d_{n-p+1} \ldots d_n). \end{array}$$

For example, if

$$\begin{array}{lcl} X & = & a\ \ b\ \ c\ \ *\ \ d\ \ e\ \ f\ \ g \\ Y & = & c\ \ a\ \ e\ \ d\ \ b\ \ g\ \ f \end{array}$$

and $\delta = 1$, then $(a\ b\ c)$ is easily identified as a league, since the distance between c and d is greater than 1 in chromosome X. The next problem is to extract the corresponding permutation in chromosome Y. This is taken care of the following lemma that describes the behavior of the function "Extract$((c_1 \ldots c_p), D)$":

Lemma 3 *Assume that π_C and π_D, and their inverse, are known. If $(c_1 \dots c_p)$ is a set of genes ordered in increasing position in chromosome C, then the corresponding permutation $(d'_1 \dots d'_p)$ on chromosome D can be obtained in time $\mathcal{O}(p \log p)$.*

Proof: Given $(c_1 \dots c_p)$, we first construct the array $A = (\pi_D^{-1}(c_1), \dots, \pi_D^{-1}(c_p))$. Sorting A requires $\mathcal{O}(p \log p)$ operations, yielding the array A'. The sequence $(d'_1 \dots d'_p)$ is given by $(\pi_D(A'_1) \dots \pi_D(A'_p))$. ■

The last operation needed to split a league is to construct the ordered complement of an ordered league. For example, for the league $\pi_Y = (c\ a\ e\ d\ b\ g\ f)$, the complement of the league $(c\ a\ b)$ is the league $(e\ d\ g\ f)$.

More formally, if $(d'_1 \dots d'_p)$ is a subsequence of $(d_1 \dots d_n)$, we will denote by

$$(d_1 \dots d_n) \setminus (d'_1 \dots d'_p)$$

the subsequence of $(d_1 \dots d_n)$ obtained by deleting the elements of $(d'_1 \dots d'_p)$. In our particular context, this operation can be done in $\mathcal{O}(p)$ steps. Indeed, once a problem is split in two sub-problems, there is no need to backtrack in the former problems. Therefore, at any point in the algorithm, each gene belongs to exactly two ordered leagues, one in each chromosome. If the gene data structure contains pointers to the previous and the following gene – if any – in both leagues, the structure can be updated in constant time as soon as an extracted gene is identified. Since p genes are extracted, the operation can be done in $\mathcal{O}(p)$ steps. An example of such an "extraction" operation is shown in Fig. 1.

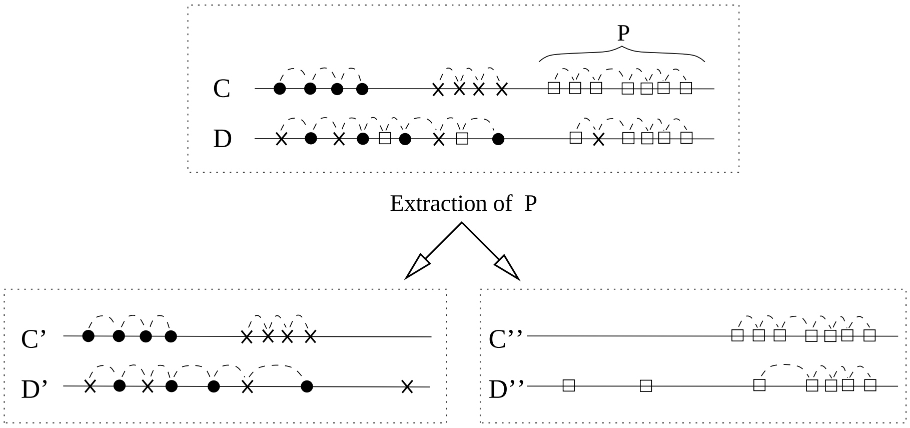

Figure 1: Extraction of a league P out of D. The initial problem on (C, D) is split in two sub-problems on (C', D') and (C'', D'').

Fig. 2 contains the formal description of the algorithm FindTeams. The three cases that are not shown correspond to the tests $\Delta_C(c_{n-p}, c_{n-p+1}) > \delta$, $\Delta_D(d_p, d_{p+1}) > \delta$ and $\Delta_D(d_{n-p}, d_{n-p+1}) > \delta$, and are duplications of the first case, up to indices.

Theorem 1 *On input π_C and π_D, algorithm* **FindTeams** *correctly identifies the δ-teams of chromosomes C and D.*

Proof: Since Σ is a league, the first input to FindTeams will be a league. The correctness of the algorithm comes from the fact that if a league S is supplied to the algorithm, then either

```
FindTeams((c_1 ... c_n), (d_1 ... d_n))
1.    SubLeagueFound ← False
2.    p ← 1
3.    While (NOT SubLeagueFound) AND p ≤ ⌊n/2⌋ Do
4.        If Δ_C(c_p, c_{p+1}) > δ OR Δ_C(c_{n−p}, c_{n−p+1}) > δ OR
5.           Δ_D(d_p, d_{p+1}) > δ OR Δ_D(d_{n−p}, d_{n−p+1}) > δ        Then
6.            SubLeagueFound ← True
7.        Else p ← p + 1
8.        End of if
9.    End of while
10.   If SubLeagueFound Then
11.       If Δ_C(c_p, c_{p+1}) > δ Then
12.           (d'_1 ... d'_p) ← Extract((c_1 ... c_p), D))
13.           FindTeams((c_1 ... c_p), (d'_1 ... d'_p))
14.           FindTeams((c_{p+1} ... c_n), (d_1 ... d_n) \ (d'_1 ... d'_p))
15.       Else If ...
16.               /* The three other cases are similar */
17.       End of if
18.   Else  (c_1 ... c_n) is a Team
19.   End of if
```

Figure 2: Fast recursive algorithm for gene teams identification.

S is a δ-team, which is the condition tested by the four tests within the loop of line 3, or it has a "small" sub-league, whose complement is also a league. ■

The space needed to execute algorithm **FindTeams** is easily seen to be $\mathcal{O}(n)$ since it needs the four arrays containing π_C, π_D, π_C^{-1}, π_D^{-1}, and the n genes, each with four pointers coding implicitly for the ordered leagues.

3.3 Time Complexity of Algorithm FindTeams

In the last section, we saw that algorithm **FindTeams** splits a problem of size n in two similar problems of size p and $n-p$, with $p \leq n/2$. The number of operations needed to split the problem is $\mathcal{O}(p \log p)$, but the value of p is not fixed from one iteration to the other. In order to keep the formalism manageable, we will "*... neglect certain technical details when we state and solve recurrences. A good example of a detail that is glossed over is the assumption of integer arguments to functions.*", [17] p. 53.

Assume that the number of operation needed to split the problem is bounded by $\alpha p \log p$, and let $F(n)$ denote the number of operations needed to solve a problem of size n. Then $F(n)$ is bounded by the function $T(n)$ described by the following equation:

$$T(n) = \max_{1 \leq p \leq \lfloor n/2 \rfloor} \{\alpha p \log p + T(p) + T(n-p)\}. \tag{1}$$

with T(1) = 1.

Surprisingly, the worst case scenario of the above equation is when the input is always split in half. Indeed, we will show that $T(n)$ is equal to the function:

$$T_2(n) \quad = \quad \frac{\alpha n}{2} \log \frac{n}{2} + 2T_2\left(\frac{n}{2}\right), \tag{2}$$

with $T_2(1) = 1$. One direction is easy:

Lemma 4 *$T(n) \geq T_2(n)$.*

Proof: Suppose that $T(i) \geq T_2(i)$ for all $i < n$, then

$$\begin{aligned} T(n) &\geq \max_{1 \leq p \leq n/2} \{\alpha p \log p + T_2(p) + T_2(n-p)\} \\ &\geq (\alpha n/2) \log(n/2) + T_2(n/2) + T_2(n - n/2) \\ &= T_2(n). \end{aligned}$$

■

In order to show the converse, we first obtain a closed form for $T_2(n)$.

Lemma 5 *$T_2(n) = n - (\alpha n/4) \log n + (\alpha n/4) \log^2 n$.*

Proof: Substituting the value $T_2(n/2)$ in the left side of Equation 2, and using the identity $\log(n/2) = (\log n) - 1$ yields:

$$\begin{aligned} T_2(n) &= (\alpha n/2) \log(n/2) + 2[n/2 - (\alpha n/8) \log(n/2) + (\alpha n/8) \log^2(n/2)] \\ &= n - (\alpha n/4) \log n + (\alpha n/4) \log^2 n. \end{aligned}$$

■

We use this relation to show the following remarkable property of $T_2(n)$. It says that when a problem is split in two, the more unequal the parts, the better.

Proposition 3 *If $x < y$ then $T_2(x) + T_2(y) + \alpha x \log x < T_2(x + y)$.*

Proof: Consider the variable $z = y/x$. The following identities are easy to derive:

$$\begin{aligned} &\log(x + y) - \log x = \log(1 + z) \\ &\log(x + y) - \log y = \log(1 + 1/z) \\ &\log^2(x + y) - \log^2 x = [2 \log x + \log(1 + z)] \log(1 + z) \\ &\log^2(x + y) - \log^2 y = [2 \log x + \log(1 + z) + \log z] \log(1 + 1/z). \end{aligned}$$

Define $H(z) = \log(1 + z) + z \log(1 + 1/z)$. Its value for $z = 1$ is 2, and its derivative is $\log(1 + 1/z)$, implying that the $H(z)$ is strictly increasing. We will show that $[T_2(x + y) - T_2(x) - T_2(y)]/(\alpha x) > \log x$. Using the closed form for T_2, we have:

$$\begin{aligned} & [T_2(x + y) - T_2(x) - T_2(y)]/(\alpha x) \\ = \; & (1/4)[\log^2(x + y) - \log^2 x] + (y/4x)[\log^2(x + y) - \log^2 y] \\ & -(1/4)[\log(x + y) - \log x] - (y/4x)[\log(x + y) - \log y]. \end{aligned}$$

Substituting y/x by z, the last expression becomes:

$$\begin{aligned}
& (H(z)/4)[2\log x + \log(1+z) - 1] + (1/4)z\log z\log(1+1/z) \\
\geq\ & (H(z)/2)\log x \\
>\ & \log x,\ \text{ since } H(z) > 2, \text{ when } z > 1.
\end{aligned}$$

■

Using Proposition 3, we get:

Proposition 4 $T(n) \leq T_2(n)$.

Proof: Suppose that $T(i) \leq T_2(i)$ for all $i < n$, then

$$\begin{aligned}
T(n) &= \max_{1\leq p\leq \lfloor n/2\rfloor} \{\alpha p\log p + T(p) + T(n-p)\} \\
&\leq \max_{1\leq p\leq \lfloor n/2\rfloor} \{\alpha p\log p + T_2(p) + T_2(n-p)\} \\
&\leq \max_{1\leq p\leq \lfloor n/2\rfloor} \{T_2(p+n-p)\} \\
&\leq \max_{1\leq p\leq \lfloor n/2\rfloor} \{T_2(n)\} \\
&= T_2(n).
\end{aligned}$$

■

We thus have:

Theorem 2 *The time complexity of algorithm* **FindTeams** *is* $\mathcal{O}(n\log^2 n)$.

Theorem 2 is truly a worst case behavior. It is easy to construct examples in which its behavior will be linear, taking, for example, an input in which one chromosome has only singletons as maximal δ-chains.

3.4 A faster algorithm

Algorithm **FindTeams** can be optimized by using a parameter δ' that depends on gene density and the value of δ:

Definition 4 *Let δ' be the maximal number of genes contained in moving window of size δ, over all the chromosomes.*

The optimization focuses on how to extract the small league P, or the *pivot* of Hopcroft's framework (see Section 4). Assume P to be of size p. The extraction algorithm will run in $\mathcal{O}(p\log\delta')$ instead of $\mathcal{O}(p\log p)$. The idea is to locally sort the genes in small *zones*, and then consider consecutive zones to find the maximal δ-teams. These consecutive zones are built by extending the neighborhood of each zone, without sorting the zones.

3.4.1 Associating a zone to each gene

Each chromosome is cut in at most $2n$ zones Z_i of length δ, and each gene on this chromosome is associated with a specific zone. A table $Z = Z_1 \dots Z_h$ is built for each chromosome to insure a direct access to a zone.

The zone building algorithm for a chromosome is given in Fig 3. The genes are scanned from left to right (line 2), the current position is initialized with the position of the first gene, the initial gene to the first gene, and the zone number to 1 (line 1). Then, if the distance between the current gene and the initial gene is greater than 2δ, we build two zones and reset the process. If this distance is between δ and 2δ, it means that we entered a consecutive zone and we also reset the process, but increment the number of zones only by one. Finally, if the distance is smaller than δ, we stay in the same zone.

```
Build_zones((c_1 ... c_n))
1.    CurrentZone ← 1  ;  InitGene ← c_1
2.    For i = 1...n Do
3.        If Δ_C(InitGene, c_i) > 2δ Then
4.            CurrentZone ← CurrentZone+2  ;  InitGene ← c_i
5.        Else
6.            If Δ_C(InitGene, c_i) > δ Then
7.                CurrentZone ← CurrentZone+1  ;  InitGene ← c_i
8.            End of if
9.        End of if
10.       Zone_C(c_i) ← CurrentZone
11.   End of for
```

Figure 3: Algorithm for assigning a zone to each gene of a chromosome C.

The h zones $Z_1, \dots, Z_h$ computed with **Build_zones** have some obvious properties. There are at most δ' genes associated with the same zone. The total number h of zones is less than or equal to $2n$, since a gene creates at most 2 zones (line 4).

3.4.2 Sorting all zones

Assume now that we want to extract a league P of size p out of a chromosome C. We first group together the genes of P that are associated to the same zone of the table Z of C. Suppose we considered l zones $Z_{i_1}, \dots, Z_{i_l}$ of size z_j, $i_1 \leq j \leq i_l$. This takes time proportional to p. We now sort each such zone using a classical optimal sort algorithm. Sorting Z_{i_j} requires $\mathcal{O}(z_j \log z_j)$ time, which is, as $z_j \leq \delta'$, less or equal than $\mathcal{O}(z_j \log \delta')$. The total complexity is then less or equal to $\mathcal{O}(\sum_{j=1}^{l} z_j \log \delta') = O(p \log \delta')$.

Note that for the rest of the extraction algorithm, we keep track, for each non empty zone Z_{i_j}, of the minimal and maximal position of the genes in Z_{i_j}. This is given by the sorting procedure without additional cost.

3.4.3 Extracting maximal δ-chains

At this point, we have a list of l sorted zones $Z_{i_1}, \dots, Z_{i_l}$ of genes, in a table $Z = Z_1 \dots Z_h$. The zones are not sorted among each other, in the sense that we cannot address the zones of $Z_{i_1}, \ldots, Z_{i_l}$ according to their order in the table Z. We show now that even without this information we can extract P in C. The idea is simply to consider for each zone Z_{i_j}, $1 \leq j \leq l$, the zone to its left in the table Z, that is Z_{i_j-1} (if it exists), and chain Z_{i_j} with Z_{i_j-1} if necessary. The zone Z_{i_j-1} is accessible in constant time through the table Z. The order in which the zones Z_{i_j} are considered is irrelevant. There are three main cases:

1. Zone Z_{i_j-1} does not exist ($i_j = 1$). Zone Z_{i_j} is directly marked as an initial zone.
2. Zone Z_{i_j-1} is empty. Then, the way zones are built by algorithm **Build_zones** (Fig. 3) insures that the genes in Z_{i_j} cannot be δ-connected to other genes to the left, since an empty zone means a distance greater than δ to any preceding gene. The zone Z_{i_j} is then marked as an initial zone.
3. Zone Z_{i_j-1}, is not empty. Then, if the distance between the last element of Z_{i_j-1} and the first element of Z_{i_j} is less or equal to δ, then Z_{i_j} is chained to Z_{i_j-1} as a following zone. Otherwise, we apply a process similar to case 2.

At the end of that process, after having considered all zones in which at least one element of P was found, all zones are either chained to the zone to their left, or initial. To finish the process, for all the initial zones, we follow the links of chained zones and concatenate the genes. This forms the maximal δ-chains, since: (a) inside a zone, the genes are δ-connected; (b) if two zones Z_{i_j-1} and Z_{i_j} are chained, the genes of these two zones are δ-connected, since we test whether the maximal gene of Z_{i_j-1} is connected to the minimal gene of Z_{i_j} or not; (c) if the δ-chain was not maximal, another zone (to the left or to the right) would have been chained.

3.4.4 Complexity

Proposition 5 *Splitting a league P of size p can be done in $\mathcal{O}(p \log \delta')$ worst case time.*

Using the analysis of Section 3.3 or the amortized techniques of Hopcroft's framework (see Section 4), we get a new algorithm with $\mathcal{O}(n \log n \log \delta')$ worst case time complexity. The optimization still requires $\mathcal{O}(n)$ space, since there are at most $2n$ zones per chromosome. The complexity analysis extends to the case of m chromosomes, yielding an $\mathcal{O}(mn \log n \log \delta')$ algorithm.

4 Hopcroft's partitioning framework

Partition refinement with pivots is a widely used technique to solve a large class of problems on graphs, strings, etc [4, 8]. The first designer was Hopcroft who used it to minimize deterministic automata [10]. We propose another version of the faster algorithm, based on partition refinement with pivots, for the computation of the δ-teams of two chromosomes. The algorithms extends to an arbitrary number m of chromosomes.

4.1 Gene teams and Hopcroft's partitioning framework

Refining a partition can be done by splitting its classes into smaller ones, according to a subset of Σ called the *pivot set*: each class X of L is replaced by $X \cap S$ and $X \setminus S$. We say that the pivot set S *splits* the partition L into a new partition. In the computation of δ-teams, pivots will always be δ-chains of one of the chromosomes.

Let L_C and L_D be the two initial partitions induced by maximal δ-chains of chromosomes C and D. We distinguish two types of pivots, called type C and type D. Pivots of type C split the partition L_D while pivots of type D split the partition L_C. Partitions are implemented by sorted lists. Therefore partitions are implicitly ordered. A partition Q is *compatible* with a partition P if every class of Q is included in a class of P and if the ordering in P respects the ordering in Q (i.e if in P the class X is before the class Y, then any class $X' \subseteq X$ of Q is before any class $Y' \subseteq Y$). A pivot splits a partition into a compatible one. Moreover, and this point differs slightly from general partition refinement schemes, each class of a partition also is implemented by a sorted list. Each class of the partition L_C is sorted according to the gene order given by chromosome C, and each class of the partition L_D is sorted accordingly to the order given by D.

Definition 5 *We say that a class X overlaps a set S if $X \not\subset S$ and $X \cap S \neq \emptyset$. Given a subset S of Σ, a partition L of Σ is said to be S-*stable *when no class of L overlaps S.*

Note that after a refinement step of L by S, the new partition is S-stable.

The **PartitionRefinement** algorithm is described in Fig. 4. While Hopcroft's original algorithm processes the "small half", we process several "small parts": initially, the stack *pivots* contains all classes of the two partitions. Then, each class in the stack is either replaced by smaller ones, or new small subclasses are stacked. The algorithm calls **Sort_zones(P)**, a procedure which computes a decomposition of the pivot P of type C (resp. D) into an union of maximal δ-chains of D (resp. C). This procedure is described in Sections 3.4.2 and 3.4.3.

Procedure **Split(X, P)**, Fig. 5, is the main part of the algorithm. If a class X properly overlaps the pivot set, the pivot splits the class X of L_C (resp. L_D) into at least two classes according to the pivot set. The obtained subclasses are still δ-chains of C (resp. D). The sizes of the subclasses are computed in parallel during the process, in order to avoid parsing an eventual – unique – large subclass. The code uses the following functions. If X is δ-chain of the chromosome C, let $(g_1, \ldots, g_k)$ be the permutation of X induced by C. We denote by $\mathrm{next}(g_i, X)$ the gene g_{i+1} when it exists, in which case $\mathrm{hasnext}(g_i, X)$ is true. If it does not exist, $\mathrm{hasnext}(g_i, X)$ is false.

The correctness of Algorithm **PartitionRefinement** is obtained with the following invariants of the **while** loop (line 6).

Proposition 6 *Partitions L_C and L_D always verify:*

1. *Each class of L_C (resp. L_D) is a δ-chain of C (resp. D).*

2. *The union of two distinct classes of L_C (resp. L_D) is not a δ-set.*

Proof. During the initialization of Algorithm **PartitionRefinement**, the classes of L_C and L_D are δ-chains of C and D respectively, and Procedure **Split** transforms a collection of δ-chains into a collection of δ-chains.

The conservation of the property 2 follows from the following property 2': for any pivot P, any element g of P and any element $g' \notin P$, g and g' cannot be in a same maximal δ-set. Properties 2' and 2 are true after the initialization step. Let us assume that they are both satisfied at some time. Then, after a splitting of a class X under a pivot, any two elements

```
PartitionRefinement(chromosomes C, D)
1.    Initializations
2.        L_C (resp. L_D) ← the collection of maximal δ-chains of C (resp. D),
          (each class of L_C (resp. L_D) is ordered by C (resp. D)).
3.        Let pivots be an empty stack of pivots.
4.        Add each class of L_C (resp. L_D) in pivots as a pivot of type C (resp. D).
5.    Refinements
6.        While (pivots is not empty) Do
7.            Pick a pivot P in pivots.
8.            Sort_zones(P)
9.            If P has type D (the case type C is similar) Then
10.               If L_C is not P-stable Then
11.                   Let M be the set of classes of L_C properly overlapping P.
12.                   For each class X ∈ M Do
13.                       Let (X_1, X_2, ..., X_r) = Split(X, P)
14.                       If (X is contained in the stack pivots) Then
15.                           Remove X from pivots and add X_1, X_2, ..., X_r
16.                           as pivots of type C.
17.                       Else
18.                           For each class X_i such that size[X_i] ≤ size[X]/2 Do
19.                               Add X_i in pivots as a pivot of type C.
20.                           End of for
21.                       End of if
22.                   End of for
23.               End of if
24.           End of if
25.       End of while
```

Figure 4: Hopcroft-like algorithm for gene teams identification.

of two distinct subclasses cannot belong to a same maximal δ-set, by construction. Thus the new pivots of the stack obtained from lines 15-16 of Algorithm **PartitionRefinement** or from lines 18-19 of Algorithm **PartitionRefinement** still verify 2', and the refined partition still verifies 2. □

Proposition 6 implies that no δ-team will be split during the process. The next proposition insures that there is always enough pivots in the stack to properly identify all δ-teams.

Proposition 7 *If the partition L_C is not Y-stable for every class $Y \in L_D$, (or if the partition L_D is not X-stable for every class $X \in L_C$), then some pivot of type D (resp. C) in the stack pivots will strictly refine this partition.*

In the case of more than two chromosomes, at the end of the execution of the algorithm, each partition of one chromosome is X-stable for each class X of a partition of another chromosome.

Proof. We show that if the partition L_C is not Y-stable for every class $Y \in L_D$, then some pivot in *pivots* will strictly refine the partition L_D. Let us assume that there is a class $X \in L_C$ such that X properly overlaps a class $Y \in L_D$. Let $g \in Y \cap X$, and $f \in (\Sigma \setminus Y) \cap X$. Consider the first time g and g' are split apart into two different classes Z_1 and Z_2 of L_D. If these

```
Split(class X ∈ L_C, pivot P of type D)
ouputs a list of classes L with their sizes
    Let L be the empty list.
    Extract maximal δ-chains X_1, ..., X_r of elements from X ∩ P
    Extract maximal δ-chains X'_1, ..., X'_s of elements from X ∩ (Σ \ P)
    For (each chain X_i) Do
        Compute size[X_i] with an exploration of the chain X_i.
        Add X_i to L.
        size[X] ← size[X] − size[X_i]
    End of for
    Let L' = (X'_1, ..., X'_s)
    For (each chain X' ∈ L') Do
        Set g(X') as the first element of X'.
        size[X'] ← 1.
    End of for
    While (L' contains more than one chain) Do
        While (hasnext(g(X'), X') for each X' ∈ L') Do
            For (each X' ∈ L') Do
                g(X') ← next(g(X'), X').
                size[X'] ← size[X'] + 1.
            End of for
        End of while
        For (each X' ∈ L such that NOT hasnext(g(X'), X')) Do
            Add X' to L.
            Remove X' from L'.
            size[X] ← size[X] − size[X'].
        End of for
    End of while
    If (L' is nonempty, and hence contains a unique chain X') Then
        Add X' to L.
        size[X'] ← size[X].
    End of if
    return L.
```

Figure 5: Splitting a class under a pivot.

classes are classes of the initial partition L_D, then Z_1 is an initial pivot. Otherwise, there is a splitting of a class $Z \ni g, g'$ into $Z_1 \ni g, Z_2 \ni g', \ldots, Z_r$. Then either Z was already in the stack of pivots, and all subclasses Z_i have been added as pivots (lines 15-16 of Algorithm **PartitionRefinement**), or Z was not in the stack, and all subclasses Z_i but at most one have been added as pivots (lines 18-19 of Algorithm **PartitionRefinement**). This produces a pivot either containing g and not g', or g' and not g. Such a pivot cannot go out of the stack since pivoting on it would split X into at least two classes. If it is split himself inside the stack (lines 15-16 of Algorithm **PartitionRefinement**), another pivot seperating g and g' still remains in the stack. Thus the stack contains a pivot able to strictly refine L_C. □

As a consequence, at the end of the execution of the process, L_C is Y-stable for every class $Y \in L_D$, and L_D is X-stable for every class $X \in L_C$. Thus L_C and L_D are collections of the same δ-sets. It follows from Proposition 6, property 2 that these δ-sets are maximal. We obtain the expected δ-teams as L_C or L_D.

4.2 Complexity

To achieve a good complexity, we use the following data structures. Any class of L_C (resp. L_D) is stored in a doubly linked list, ordered by C (resp. D). All the classes of a partition are stored in a doubly linked list. Each element of a class has a pointer to its class. Moreover, each gene can be accessed directly in L_C and in L_D, by the use of a table. This data structure is illustrated by Figure 6 which represents the initial partition L_C for the two following chromosomes C, D with $\delta = 2$.

$$\begin{array}{rcccccccccc} C & = & c & * & * & e & d & a & * & b & \\ D & = & a & b & * & * & * & c & * & d & e. \end{array}$$

The initializations are performed in a linear time $O(n)$ for two chromosomes.

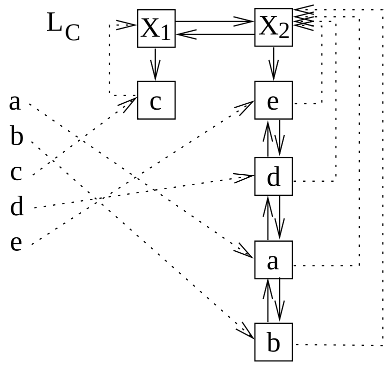

Figure 6: The initial partition L_C.

The complexity analysis uses amortized techniques, especially the *pointed parts* technique used in [4] or [2, p. 331]. We consider pairs (P, g) made of a pivot P going out of the stack of pivots (line 7 of the algorithm **PartitionRefinement**), and an element of g in P. The basic result is the following:

Proposition 8 *Each gene g appears at most $2 \log n$ times in a pivot P going out of the stack.*

Proof. If a pivot P containing an element g is going out of a stack and has size p, a pivot containing g which enters the stack later is included in P, and has size at most $p/2$. Thus, it will have a size at most $p/2$ while going out of the stack also. A gene g belongs initially to two pivots, one of type C and one of type D. $\Box$

Let $c(P, g)$ be the amortized cost of processing the pointed pair (P, g). Then, by Proposition 8 the global cost of the algorithm will be given by $2n\ c(P, g) \log n$. We establish, in the next proposition that $c(P, g)$ is $\mathcal{O}(\log \delta')$.

[Note that the complexity analysis assumes the following data structures. Any class of L_C (resp. L_D) is stored in a doubly linked list, ordered by C (resp. D). All the classes of a partition are stored in a doubly linked list. Each element of a class has a pointer to its class. Moreover, each gene can be accessed directly in L_C and in L_D, by the use of a table.]

Proposition 9 *The amortized cost $c(P, g) = c_1 \log \delta' + c_2$, where c_1 and c_2 are constants.*

Proof. Let us assume that we pick a pivot P of type D, and of size p, in the stack. This pivot is first processed by **Sort_zones** in time $\mathcal{O}(p \log \delta')$. We assign to each (P, g) a cost $\log \delta'$, so that the sum of these costs for all g in P equals the cost of the sorting operation. The computation of the set M of lines 10-11 of the algorithm is done in time $\mathcal{O}(p)$ by exploring P and using the direct links from a gene to its position in a class. This increments the cost of each (P, g) by a constant.

We now consider the cost induced by Procedure **Split**. Let h be the size of the class X to be split. We claim that the extractions of lines 2-3 also are performed in time $\mathcal{O}(p)$. Indeed, one extracts a δ-chain X_i of elements of $X \cap P$ by exploring the list P, and by checking the δ-connection for the order induced by C. More precisely, when an element, candidate to be added in X_i, is not δ-connected to the previous ones for the order C, one builds a new class X_{i+1}. If it is δ-connected, it is removed from X in constant time. If X is no longer δ-connected, we cut it into a δ-chain X'_j of elements in $X \cap (\Sigma \setminus P)$, and a new δ-chain X. This increments the cost of each (P, g) only with another constant. Remark that this implies that there are at most p subclasses X_i. Note also that, at this time, the sizes of the subclasses, and the pointers from each element in a class to its class, have not been updated.

We next consider the cost of the computation of the sizes of the subclasses. The computation of the sizes of the subclasses X_i is performed lines 4-8 of Procedure **Split** in time $\mathcal{O}(p)$, since the sum of the sizes of these subclasses is at most p. This charges (P, g) with a constant again. The computation of the sizes s'_j of the subclasses X'_j is done in lines 14-30. Recall that a small subclass has a size less than or equal to $h/2$. Since L' in lines 14-26 has at least two subclasses, the subclasses removed in line 23 are small. At line 26, all subclasses that have been read completely are small, and the beginning of an eventual unique large subclass Y may have been explored. Nevertheless, the maximal number of elements of Y read is the maximal size of all other subclasses. The pointers from each element in a class to its class are recomputed for all subclasses but Y. Thus the cost of the computation of the sizes and pointers of all subclasses is at most $2 \sum_{j \in J} s'_j$, where J is the index set of all subclasses but Y. Since all subclasses but Y are at some time contained in the stack of pivots, and can go out of it by being removed in line 14, one charges again each (P, g) with one more constant, in order to count the cost of these operations. □

Proposition 10 *The time complexity of the algorithm* **PartitionRefinement** *is* $\mathcal{O}(n \log n \log \delta')$ *for two chromosomes and* $\mathcal{O}(mn \log n \log \delta')$ *for* m *chromosomes.*

4.3 From Hopcroft like algorithm to FindTeams

The two algorithms **PartitionRefinement** and **FindTeams** are very close. The algorithm **FindTeams** is in fact a recursive simplification of the Hopcroft like one. The simplication is based on the two following remarks.

First, the stack *pivots* of lines 6-8 of Algorithm **PartitionRefinement** is simulated in **FindTeams** by the recursive calls to itself of lines 13-14. This uses a property of the problem that is not valid for all Hopcroft like algorithms, and allows to divide the original problem in two subproblems. Indeed, assume that in line 11 of **PartitionRefinement** a pivot P (say of L_D) splits the set of classes M of L_C whose alphabet intersects that of P. The split is performed using **Split**, which partitions the resulting classes of L_C in two sets, those that contains elements of P and the others. Some of these classes will be reintegrated in the list *pivots* in lines 18-19 of **PartitionRefinement** and reused later to split other classes. A simple observation is that the classes of L_C built with elements of P after **Split**, if reused as pivots, would only cut classes built with elements of P of L_D. This property allows us to derive two sub-problems after a **Split**, on one hand all classes of L_C built of elements of P together with P on L_D, and, on the other hand, all the classes remaining on L_C and L_D. This is used in **FindTeams** to recursively call the same algorithm on these two sets in lines 13-14 of Algorithm **FindTeams**.

A second remark concerns the computation of the sizes of the classes. In the Hopcroft-like algorithm, when splitting a class X with a pivot P, the sizes of the resulting classes of size less than or equal to size$[X]/2$ are computed in lines 14-30 of **Split**. After the split, in lines 18-19 of algorithm **PartitionRefinement**, the classes are kept as potential pivots. Algorithm **FindTeams** simplifies this step lines by finding a small class of size p (if it exists) in $\mathcal{O}(p)$ and considering it as a pivot.

5 Extensions

5.1 Multiple Chromosomes

The most natural extension of the definition of δ-teams to a set $\{C_1, \ldots, C_m\}$ of chromosomes, is to define a δ-set S as a δ-chain in each chromosome C_1 to C_m, and consider maximal δ-sets as in Definition 2. For example, with $\delta = 2$, the only δ-team of chromosomes:

$$\begin{array}{lcllllllll} X & = & c & * & * & e & d & a & * & b \\ Y & = & a & b & * & * & c & * & d & e \\ Z & = & b & a & e & * & * & c & * & d, \end{array}$$

that is not a lonely gene is the set $\{a, b\}$.

All the definitions and results of Section 2 apply directly to this new context, replacing C and D by the m chromosomes.

Algorithm **Findteams** can be readily adapted to m chromosomes by modifying its two main tasks of finding and extracting small leagues. Identifying a small league in m partitions can be done in $O(mp)$. This small league must then be extracted from $m-1$ chromosomes, yielding two sub-problems, one of which is of size p. The analysis of Section 3.3 yields directly an $\mathcal{O}(mn \log^2 n)$ time bound for this algorithm, since the parameter α in Equation 1 was arbitrary.

5.2 Extension to Circular Chromosomes

In the case of circular chromosomes, we first modify slightly the assumptions and definitions. The positions of genes are given here as values on a finite interval:

$$\Sigma \xrightarrow{P_C} [0..L],$$

in which position L is equivalent to position 0. The distance between two genes g and g' such that $P_C(g) < P_C(g')$ is given by:

$$\Delta_C(g, g') = \min \left\{ \begin{array}{l} P_C(g') - P_C(g) \\ P_C(g) + L - P_C(g'). \end{array} \right.$$

The permutation $\pi_C = (g_1 \ldots g_n)$ is still well defined for circular chromosomes, but so are the permutations, for $1 < m \leq n$:

$$\pi_C^{(m)} = (g_m \ldots g_n g_1 \ldots g_{m-1}).$$

A δ-chain in a circular chromosome is any δ-chain of at least one of these permutations. A *circular* δ-chain is a δ-chain $(g_1 \ldots g_k)$ such that $\Delta_C(g_k, g_1) \leq \delta$: it goes all around the chromosome. All other definitions of Section 2 apply without modifications.

Adapting algorithm FindTeams to circular chromosomes requires a special case for the treatment of circular δ-chains. Indeed, in Section 3.2, the beginning and end of a chromosome provided obvious starting places to detect leagues. In the case of circular chromosomes, assume that S is a league of chromosomes C and D, and that the genes of S are respectively ordered in C and D, from arbitrary starting points, as:

$$(c_1 \ldots c_n) \text{ and } (d_1 \ldots d_n).$$

If none of these sequences is a circular δ-chain, then there is a gap of length greater than δ on each chromosome, and the problem is reduced to a problem of linear chromosomes. If both are circular δ-chains, then S is a δ-team. Thus, the only special case is when one is a circular δ-chain, and the other, say $(c_1 \ldots c_n)$ has a gap greater than δ between two consecutive elements, or between the last one and the first one. Without loss of generality, we can assume that the gap is between c_n and c_1. Then, if S is not a team, there exists a value $p \leq n/2$ such that one of the following sequence is a league:

$$\begin{array}{l}(c_1 \ldots c_p) \\ (c_{n-p+1} \ldots c_n.)\end{array}$$

The extraction procedure is similar to the one in Section 3.2, but both the extracted leagues can again be circular δ-chains, as illustrated in Fig. 7.

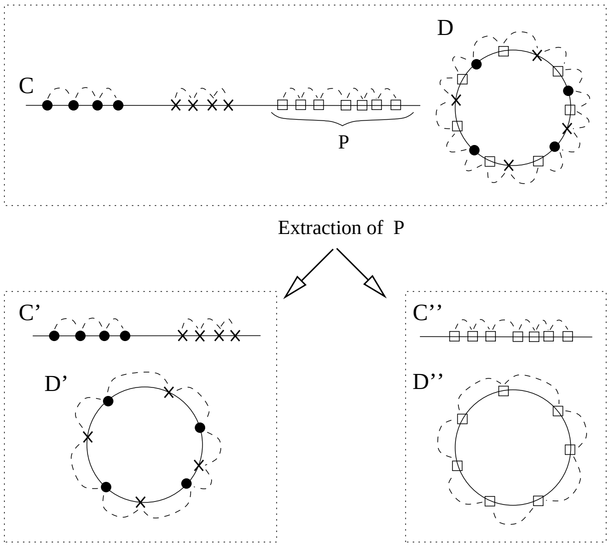

Figure 7: Special case that might occur when extracting the league p out of a circular league of D. Both extracted leagues are again circular δ-chains of D' and D''.

The circularity can be detected in $\mathcal{O}(p)$ steps, since the property is destroyed if and only if an extracted gene creates a gap of length greater than δ between its two neighbors.

5.3 Teams With a Designated Member

A particular case of the team problem is to find, for various values of δ, all δ-teams that contain a designated gene g. Clearly, the output of algorithm FindTeams can be filtered for the designated gene, but it is possible to do better. In lines 13 and 14 of Fig. 1, the original problem is split in two subproblems. Consider the first case, in which the sub-league $(c_1 \ldots c_p)$ is identified:

1. If gene g belongs to $(c_1 \ldots c_p)$, then the second recursive call is unnecessary.

2. If gene g does not belong to $(c_1 \ldots c_p)$, then the extraction of (d'_1, d'_p), and the first recursive call, are not necessary.

These observations lead to a simpler recurrence for the time complexity of this problem, since roughly half of the work can be skipped at each iteration. With arguments similar to those in Section 3.3, we get that the number of operations is bounded by a function of the form:

$$T(n) = \alpha(n/2)log(n/2) + T(n/2),$$

where $T(1) = 1$, and whose solution is: $T(n) = \alpha n \log n - 2\alpha n + 2\alpha + 1$.

6 Conclusions and perspectives

We defined the unifying notion of *gene teams* and we constructed two distinct identification algorithms for n genes belonging to two or more chromosomes, the faster one achieving $O(mn \log n \log \delta')$ time for m linear or circular chromosomes. Both algorithms require only linear space.

The gene team identification problem is more complex than one could think in view of the simplicity of the first recursive algorithm. We showed in a second part that this algorithm is in fact a nice simplification of a full Hopcroft partitioning algorithm. However, instead of leading to a faster algorithm, this strong link reinforces our estimation of the intrinsic complexity of the gene team identification problem. In some particular Hopcroft like algorithms, a clever pivot choice can reduce the complexity from $\mathcal{O}(n \log n)$ to $\mathcal{O}(n)$ [8]. Obtaining faster algorithms or lower bounds for the gene team identification problem remains open.

We intend to extend our work in two directions that will further clarify and simplify the concepts and algorithms used in comparative genomics. The first is to relax some aspect of the definition of gene teams. For large values of m, the constraint that a set S be a δ-chain in all m chromosomes might be too strong. Sets that are δ-chains in a quorum of the m chromosomes could have biological significance as well. We also assumed, in this paper, that each gene in the set Σ had a unique position in each chromosome. Biological reality can be more complex. Genes can go missing in a certain species – their function being taken over by others, and genes can have duplicates.

In a second phase, we plan to extend our notions and algorithms to combine distance with other relations between genes. For example, interactions between proteins are often studied through metabolic or regulatory pathways, and these graphs impose further constraints on teams.

A complete implementation handling multiple linear or circular chromosomes is available at `http://www-igm.univ-mlv.fr/~raffinot/geneteam.html`.

Acknowledgment

We would like to thank Marie-France Sagot for interesting discussions, Laure Vescovo and Nicolas Luc for a careful reading, and Myles Tierney for suggestions on the terminology. A special thanks goes to Laurent Labarre for his bibliography work.

References

[1] A.K. Bansal. An automated comparative analysis of 17 complete microbial genomes. *Bioinformatics*, 15(11):900–908, 1999.

[2] D. Beauquier and J. Berstel an P. Chrétienne, editors. *Eléments d'algorithmique*. Masson, Paris, 1992.

[3] A. Bergeron, S. Corteel, and M. Raffinot. The algorithmic of gene teams. In *Workshop on Algorithms in Bioinformatics (WABI)*, number 2452 in Lecture Notes in Computer Science, pages 464–476. Springer-Verlag, Berlin, 2002.

[4] A. Cardon and M. Crochemore. Partitioning a graph in $O(|A| \log_2 |V|)$. *Theoretical Computer Science*, 19(1):85–98, 1982.

[5] T. Colombo, A. Guénoche, and Y. Quentin. Inférence fonctionnelle par l'analyse du contexte génétique: une application aux transporteurs ABC, 2002. Journées ALBIO, Montpellier, March.

[6] T. Dandekar, B. Snel, M. Huynen, and P. Bork. Conservation of gene order: a fingerprint of proteins that physically interact. *Trends Biochem. Sci.*, 23(9):324–328, 1998.

[7] W. Fujibuchi, H. Ogata, H. Matsuda, and M. Kanehisa. Automatic detection of conserved gene clusters in multiple genomes by graph comparison and p-quasi grouping. *Nucleic Acids Research*, 28(20):4029–4036, 2000.

[8] M. Habib, C. Paul, and L. Viennot. Partition refinement techniques: an interesting algorithmic tool kit. *International Journal of Foundations of Computer Science*, 10(2):147–170, 1999.

[9] S. Heber and J. Stoye. Finding all common intervals of k permutations. In *Combinatorial Pattern Matching (CPM)*, number 2089 in Lecture Notes in Computer Science, pages 207–218. Springer-Verlag, Berlin, 2001.

[10] J. E. Hopcroft. An $n \log n$ algorithm for minimizing the states in a finite automaton. In Z. Kohavi, editor, *The Theory of Machines and Computations*, pages 189–196. Academic Press, 1971.

[11] M. Huynen, B. Snel, W. Lathe, and P. Bork. Predicting Protein Function by Genomic Context: Quantitative Evaluation and Qualitative Inferences. *Genome Research*, 10:1024–1210, 2000.

[12] N. Luc, J.-L. Risler, A. Bergeron, and M. Raffinot. Gene Teams: A New Formalization of Gene Clusters For Comparative Genomics. *Computational Biology and Chemistry (ex. Computer and Chemistry)*, 27(1):59–67, 2002.

[13] A. Morgat. Synténies bactériennes, 2001. Entretiens Jacques Cartier on Comparative Genomics, Lyon, December.

[14] H. Ogata, W. Fujibuchi, S. Goto, and M. Kanehisa. A heuristic graph comparison algorithm and its application to detect functionally related enzyme clusters. *Nucl. Acids. Res.*, 28(20):4021–4028, 2000.

[15] R. Overbeek, M. Fonstein, M. D'Souza, G. D. Pusch, and N. Maltsev. The use of gene clusters to infer functional coupling. *Proc. Natl. Acad. Sci. USA*, 96(6):2896–2901, 1999.

[16] Robert Paige and Robert E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.

[17] C. Leiserson T. Cormen and R. Rivest, editors. *Introduction to Algorithms*. MIT Press, 1992.

[18] R. L. Tatusov, M. Y. Galperin, D. A. Natale, and E. V. Koonin. The COG database: a tool for genome-scale analysis of protein functions and evolution. *Nucl. Acids. Res.*, 28(1):33–36, 2000.

[19] T. Uno and M. Yagiura. Fast algorithms to enumerate all common intervals of two permutations. *Algorithmica*, 26(2):290–309, 2000.