UNIVERSITÉ DU QUÉBEC À MONTRÉAL

# DUPLICATIONS ET COMPARAISON DE GÉNOMES : UNE APPROCHE PRAGMATIQUE.

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

KARINE ST-ONGE

NOVEMBRE 2005

Il n'faut pas chercher à
savoir où s'en va le temps
Il s'en va pareil aux glaces dans le Saint-Laurent
On fait toute la vie semblant qu'on va durer toujours
Pareil au fleuve dans son cours
Et c'est peut-être rien que pour ça qu'on fait des enfants

---

« Il était une fois des gens heureux »
de Marie-Élaine Thibert
composé par Stéphane Venne

## REMERCIEMENTS

Je voudrais remercier d'abord et avant tout, le destin, car c'est grâce à la force du destin que j'ai rencontré mes deux directeurs...

Tout à commencé à l'été 2001 où je devais choisir un(e) directeur(trice) de stage en recherche parmi une liste de professeurs que je ne connaissais pas. Mon choix s'est arrêté sur François Bergeron qui m'a suggéré quatre sujets. Étant donné que j'avais choisi un sujet touchant à la bioinformatique, François m'a référé à Anne Bergeron. C'est à cet instant que j'ai connu ma directrice Anne. Lors de ce stage, nous avons travaillé entre autres avec Cedric Chauve. C'est plus particulièrement à l'été 2002 que j'ai connu mon directeur Cedric, car il a souhaité me diriger au cours d'un deuxième stage en recherche.

Merci au destin !

Cedric et Anne ont accepté d'être mes directeurs de maîtrise, et je leur en suis très reconnaissante. J'ai apprécié votre patience, votre aide, votre support et votre travail. Tout ceci a été exceptionnel, autant que vous l'êtes.

Merci à Cedric et à Anne !

Mais avant de débuter la maîtrise, j'ai eu beaucoup d'encouragements et de soutient moral de plusieurs personnes.

Merci à ma famille et aux ami(e)s !

Finalement, tous ces « merci » ne seraient sans doute pas possible sans l'aide financière que j'ai eue tout au long de mes études.

Merci au CRSNG, au FQRNT et à mes parents !

Merci à tous !

## Table des matières

## Table des figures

## LISTE DES SYMBOLES ET NOTATIONS

| | | |
|---|---|---|
| Séquence | Liste ordonnée de nucléotides ou d'acides aminés | p. 5 |
| Suite | Liste ordonnée de signaux (gènes, protéines, domaines, etc.) | p. 14 |
| $A$, $B$, $C$, ... | Étiquettes attribuées aux signaux | p. 15 |
| $s_i$ | Séquence | p. 17 |
| Segment | Sous-suite | p. 18 |
| $g_i$ | Gène | p. 18 |
| $S_i$ | Suite | p. 20 |
| Famille de gènes | Ensemble de gènes homologues | p. 31 |
| Groupe de gènes | Ensemble de gènes de familles différentes | p. 31 |

**Résumé**

Nous présentons dans ce mémoire un travail algorithmique dans le cadre de la bioinformatique et plus précisément de l'analyse comparée de génomes entiers. Le principal résultat de notre travail est un algorithme efficace de détection de groupe de segments de génomes de contenu similaire en gènes. Notre algorithme a une complexité quadratique en temps ce qui permet de comparer plusieurs génomes en un temps acceptable. Nous avons appliqué cet algorithme à plusieurs jeux de données, incluant un ensemble de douze génomes de $\gamma$-protéobactéries.

# INTRODUCTION

L'augmentation, au cours des dernières années, du nombre de génomes séquencés disponibles dans les banques de données publiques a rendu nécessaire le développement d'algorithmes efficaces d'analyse de génomes complets permettant de comparer des ensembles de plusieurs génomes. Ce champ de recherche est actuellement l'un des plus actifs dans les domaines de la bioinformatique et de la génomique comparée. Le travail présenté dans ce mémoire s'inscrit dans cette ligne et présente un algorithme efficace de détection de groupes de segments de gènes conservés dans plusieurs génomes.

Sans entrer dans les détails, nous considérons qu'un génome est une suite ordonnée de gènes et nous nous intéressons aux regroupements de segments (sous-suite) ayant un contenu en gène similaire. Lorsqu'un segment de gènes est conservé entre les génomes des espèces, il est fort probable que c'est dû à la viabilité des espèces. C'est-à-dire qu'il existe certaines configurations des mêmes gènes qui permettent à une espèce de vivre et d'autres pas. De plus, certains gènes se retrouvent dans un génome côte à côte et dans un certain ordre pour des raisons de mécanismes cellulaire. C'est le cas, entre autres, des opérons. Plusieurs équipes de recherche se sont intéressées aux regroupements de segments, afin de proposer un modèle ayant un temps de calcul plus acceptable qu'exponentiel, ce qui est le cas du modèle général. Nous proposons dans ce mémoire un modèle permettant de calculer efficacement des regroupements de segments mais, auparavant, nous présentons les éléments nécessaires à une meilleure compréhension de nos travaux.

Dans le chapitre I, nous présentons rapidement les notions biologiques élémentaires. Nous abordons la notion de gène et de la structure du génome. Nous allons expliquer le phénomène de l'évolution des génomes et donner une introduction sur le sujet de la génomique comparée et de la détection des groupes de gènes.

Au chapitre II, nous présentons les notions à la base du regroupement de segments : la construction des familles d'homologues et le calcul des équipes de gènes. Nous détaillons également chacun des modèles qui ont été proposés dans les travaux antérieurs.

Nous présentons notre modèle dans le chapitre III. Nous allons décrire chacun des étapes de notre algorithme, dont le code source est fourni en annexe.

Les résultats de nos expérimentations sont présentés au chapitre IV. Nous avons fait des expérimentations sur trois jeux de données. Le premier jeu de données contient deux génomes, le deuxième trois et le dernier contient douze génomes.

# Chapitre I

# LA COMPARAISON DE GÉNOMES

Dans ce chapitre, nous décrivons le fonctionnement et la structure du génome, ainsi que les grands principes de l'analyse comparative des séquences génomiques qui sont à la base des problèmes algorithmiques que nous présentons dans le reste de ce mémoire. La majorité des définitions proviennent de (Cordoliani H., 1994) et (Bernot A., 2001).

## 1.1 Gènes et génome

Le *génome* d'un organisme vivant est un ensemble de molécules contenant la totalité de son information génétique. Son rôle principal consiste à contrôler l'ensemble des processus biochimiques, en particulier la synthèse de protéines, qui participent au développement et à l'évolution de l'organisme. Chaque cellule contient une copie complète du génome qui dirige l'activité spécifique de celle-ci : une cellule de peau par exemple, n'a pas le même rôle qu'une cellule du cerveau, mais chacune possède une même copie du génome, qui contrôle différents processus. Ce rôle de contrôle de processus complexes justifie la métaphore classique comparant le génome à un programme informatique, interprété par la machinerie cellulaire, dans le but de faire fonctionner un organisme.

**La double hélice d'ADN.** Du point de vue biochimique, un génome est un ensemble de chromosomes, dont le nombre varie d'une espèce à l'autre : le génome des bactéries, par exemple, se compose généralement d'un seul chromosome, mais le génome humain

comporte 23 paires de chromosomes. Un *chromosome* est constitué de deux longues chaînes de *nucléotides* enroulées l'une autour de l'autre et formant le motif désormais célèbre de la *double hélice*. Ces deux chaînes sont parfois appelées les deux *brins* de l'acide désoxyribonucléique (ADN), et les nucléotides sont aussi appelés des *bases*. Les quatre bases qui entrent dans la composition de l'ADN sont l'Adénine (notée A), la Thymine (T), la Guanine (G) et la Cytosine (C). La double hélice est formée par l'appariement des bases des deux brins selon une structure découverte par Watson et Crick (Crick F. - Watson J., 1953) :

$$\begin{aligned} A &\rightarrow T \\ T &\rightarrow A \\ C &\rightarrow G \\ G &\rightarrow C \end{aligned}$$

Chaque couple de bases ainsi liées forme une *paire de bases* (Figure 1.1). Il s'ensuit que chaque brin peut être considéré comme le miroir de l'autre pour la relation définie par les appariements Watson-Crick : on dit que les deux brins sont *complémentaires*.

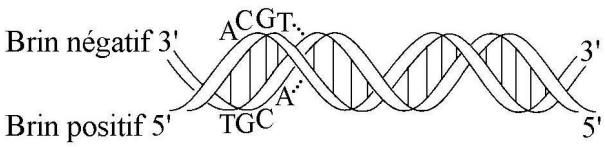



**Figure 1.1** L'ADN

**Orientation d'un chromosome.** Chaque brin d'une double hélice d'ADN comporte une extrémité initiale appelée extrémité 5', et une extrémité terminale appelée extrémité 3'. Dans l'appariement des deux brins formant la double hélice d'ADN, l'extrémité 5' d'un brin s'apparie toujours avec l'extrémité 3' de l'autre et vice-versa. Cette distinction entre ces deux extrémités induit une orientation pour les brins d'un chromosome. Dans le cadre des processus biochimiques impliquant une lecture du génome, c'est-à-dire un

déplacement le long de la double hélice, un des deux brins est lu de 5' vers 3' (on parle de brin *positif*) et le second est le brin *négatif*. Par exemple, à partir de la figure 1.1, les bases lues sur le brin positif sont : A C G T, et les bases correspondantes sur le brin négatif sont : T G C A.

**Gènes et protéines.** Le rôle principal d'un chromosome dans le fonctionnement d'une cellule est de contrôler la synthèse ou la production de protéines. Une *protéine* est constituée d'une chaîne, ou séquence, d'acides aminés, détaillée à la section 1.1.2, qui forme sa structure *primaire*. Cette chaîne adopte une conformation tridimensionnelle dénommée sa structure *tertiaire*. La chaîne d'acides aminés (structure primaire) d'une protéine est synthétisée lors de la *lecture* d'une partie d'un chromosome spécifique à cette protéine, appelée un *gène*. Pour reprendre la métaphore informatique, un gène peut être vu comme un sous-programme spécifiquement chargé de la synthèse de la séquence d'acides aminés formant la base d'une protéine. On dit qu'un tel gène *code* pour cette protéine.

Il faut mentionner ici que certains gènes ne codent pas pour des protéines. Il existe plusieurs familles de gènes impliquées dans le processus de synthèse de protéines en tant qu'auxiliaires de cette synthèse et non comme contenant les informations relatives à une protéine donnée. On les appelle *gènes non codants*. Ces familles comprennent notamment les acides ribonucléiques (ARN) dont les ARN de transfert (ARNt), les ARN ribosomaux (ARNr) constitutifs du ribosome, ainsi que de nombreuses autres familles d'ARN non codants dont le rôle est encore peu connu, mais semble être fondamental (Eddy S. R., 2001).

Finalement, il est important de noter qu'un gène, codant ou non codant, est une séquence de nucléotides et se trouve donc sur un brin du chromosome. L'information présente sur le brin complémentaire est généralement non fonctionnelle du point de vue biologique.

La synthèse de la séquence d'acides aminés à partir d'un gène codant, qui constitue la structure primaire d'une protéine, s'effectue en deux étapes principales (Figure 1.2) : la *transcription*, qui «  extrait » une copie du gène du chromosome, l'ARN messager

(ARNm), que nous décrivons à la section 1.1.1, et la *traduction*, qui transforme cette copie en une séquence d'acides aminés, détaillée à la section 1.1.2. Un gène non codant ne subit que l'étape de transcription.

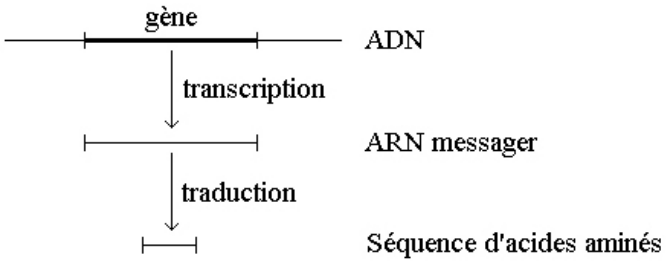



**Figure 1.2** La synthèse de la séquence d'acides aminés

**Eucaryotes et procaryotes.** Il est important de distinguer deux familles d'organismes, les eucaryotes et les procaryotes, car elles diffèrent de manière fondamentale au niveau des propriétés de leur génome. Les *eucaryotes* sont des organismes dont les cellules sont pourvues d'un vrai noyau, contenant la majorité du matériel génétique alors que, chez les procaryotes, qui n'ont pas de noyau, le matériel génétique est libre dans les cellules. Les mammifères par exemple, sont des eucaryotes, tandis que les bactéries sont des procaryotes. Parmi les différences pertinentes vis-à-vis de notre propos, c'est-à-dire la comparaison de génomes complets, on peut notamment relever que le génome des eucaryotes est beaucoup plus grand que celui des procaryotes. En effet, le génome des eucaryotes peut comprendre quelques milliards de paires de bases (3,2 milliards pour le génome humain) contre quelques millions de paires de bases pour les procaryotes. De plus, un génome de procaryote a généralement une plus grande proportion de gènes codants qu'un génome d'eucaryote, dont les gènes sont beaucoup plus dispersés (près de 98% des bases du génome humain ne codent pas pour des protéines).

### 1.1.1 La transcription

La transcription d'un gène codant pour une protéine consiste à extraire du chromosome, sous forme d'ARNm, la séquence de nucléotides qui comporte toutes les informations

nécessaires à la synthèse de la structure primaire de cette protéine. En fait, cette chaîne de nucléotides est un *pré-ARNm* qui subira possiblement certaines transformations avant d'atteindre l'état final d'ARNm et d'être prêt pour la traduction. Cette étape de transcription se décompose en trois phases : l'initiation de la transcription, l'élongation et la terminaison. Elle fait intervenir toute une machinerie fondamentale, dont un enzyme : l'*ARN polymérase*.

**Initiation de la transcription.** Lors de l'initiation, la machinerie moléculaire se fixe sur le chromosome, en amont du gène sur une zone appelée *promoteur* de la transcription. Il n'est pas rare, chez les procaryotes en particulier, qu'un même promoteur soit commun à plusieurs gènes consécutifs. Ce phénomène est à la base de la notion *d'opéron*. Le rôle du promoteur consiste à indiquer la présence d'un gène, ainsi qu'à fournir tout un ensemble d'informations spécifiques à la transcription de ce gène, liées par exemple à l'épissage alternatif, décrit à la section 1.2. Une fois l'initiation terminée, le processus de transcription proprement dit peut débuter.

Le promoteur est un exemple intéressant de ce que l'on appelle les *séquences fonctionnelles non codantes*, dont font partie les gènes non codants introduits dans la section précédente. Il s'agit de séquences ayant un rôle dans le fonctionnement du génome, d'où leur qualificatif de fonctionnelles, mais ne synthétisant pas de séquences d'acides aminés, d'où non codantes. Dans le génome humain par exemple, les séquences codantes représentent entre 1 et 2% des bases et les séquences fonctionnelles non codantes entre 3 et 4% (Waterston R. H. *et al.*, 2002).

**Élongation de la transcription.** La phase principale de la transcription est l'élongation, durant laquelle la machinerie moléculaire ouvre la double hélice formant le chromosome, en cassant les liens constituant les paires de bases, et lit le gène, sur le brin où il est situé, pour produire, hors du génome, la séquence de nucléotides complémentaires dans laquelle les T sont remplacés par des U :

$$A \rightarrow U$$
$$C \rightarrow G$$
$$G \rightarrow C$$
$$T \rightarrow A$$

C'est cette copie complémentaire légèrement modifiée du gène que l'on nomme ARNm. Par exemple, si la séquence d'un gène est A C T T G C T C A G A, alors l'ARNm correspondant sera U G A A C G A G U C U. Après le passage de la machinerie de transcription, la double hélice de l'ADN se reforme naturellement (Figure 1.3).

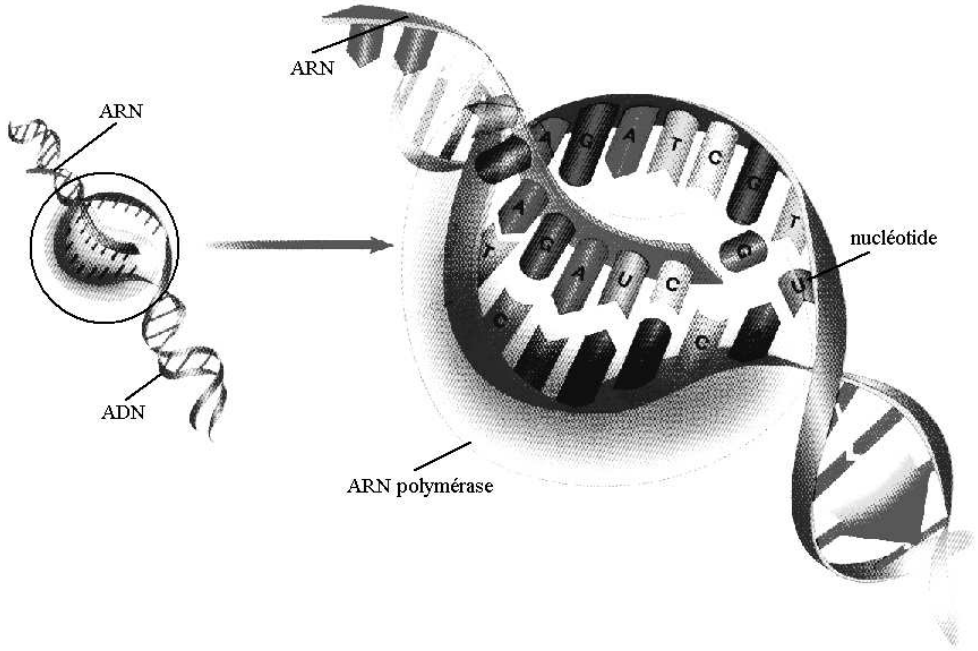



**Figure 1.3** La transcription (Lien Internet, Gresham High School)

**Terminaison de la transcription.** C'est en reconnaissant un signal particulier que la machinerie moléculaire détecte la fin d'un gène, ce qui provoque la fin de la transcription.

### 1.1.2 La traduction

Une fois un gène codant transcrit, la molécule d'ARNm résultante est une séquence de nucléotides non rattachée au chromosome. Elle doit encore être transformée en une séquence d'acides aminés, qui sera la structure primaire d'une protéine. Cette transformation constitue l'essentiel de l'étape de la traduction. Son principe est relativement simple : l'ARNm est lu par triplets de nucléotides consécutifs (nucléotides 1,2,3, puis 4,5,6, puis 7,8,9, ...), nommés *codons*, et chaque codon produit un acide aminé spécifique. On transforme une séquence de $3k$ nucléotides en une séquence de $k$ acides aminés.

Pour être complet, il faut préciser que certains codons ne produisent pas d'acide aminé, il s'agit des codons « STOP » qui indiquent la fin de la séquence de l'ARNm. De plus, comme il existe 64 codons possibles, c'est-à-dire 64 triplets sur un alphabet à 4 lettres, et seulement 20 acides aminés (Table 1.1), certains acides aminés peuvent être synthétisés par plusieurs codons.

| **Codons** | **Acide aminé** | **Symbole** |
|---|---|---|
| GCA GCC GCG GCU | Aline | *A* |
| UGC UGU | Cystéine | *C* |
| GAC GAU | Acide aspartique | *D* |
| GAA GAG | Acide glutamique | *E* |
| UUC UUU | Phénylalanine | *F* |
| GGA GGC GGG GGU | Glycine | *G* |
| CAC CAU | Histidine | *H* |
| AUA AUC AUU | Isoleucine | *I* |
| AAA AAG | Lysine | *K* |
| UUA UUG CUA CUC CUG CUU | Leucine | *L* |
| AUG | Méthionine | *M* |
| AAC AAU | Asparagine | *N* |
| CCA CCC CCG CCU | Proline | *P* |
| CAA CAG | Glutamine | *Q* |
| AGA AGG CGA CGC CGG CGU | Arginine | *R* |
| AGC AGU UCA UCC UCG UCU | Sérine | *S* |
| ACA ACC ACG ACU | Thréonine | *T* |
| GUA GUC GUG GUU | Valine | *V* |
| UGG | Tryptophane | *W* |
| UAC UAU | Tyrosine | *Y* |
| UAA UAG UGA | « stop » | |

**Table 1.1** Les vingt acides aminés et les codons correspondants (Lien Internet, Lexique EncycloBio)

L'ensemble de cette étape se déroule dans une structure appelée *ribosome*, que l'on peut considérer comme « l'usine à protéines » (Figure 1.4). Elle est constituée d'ARN ribosomaux qui s'agglomèrent en deux sous-structures, la petite sous-structure et la grande sous-structure, entre lesquelles la séquence de l'ARNm se glisse pour être lue, codon par codon. Pour chaque codon lu, un ARN de transfert spécifique vient se fixer sur le codon et synthétise l'acide aminé correspondant, qui se fixe à l'extrémité de la chaîne d'acides aminés en cours de formation.

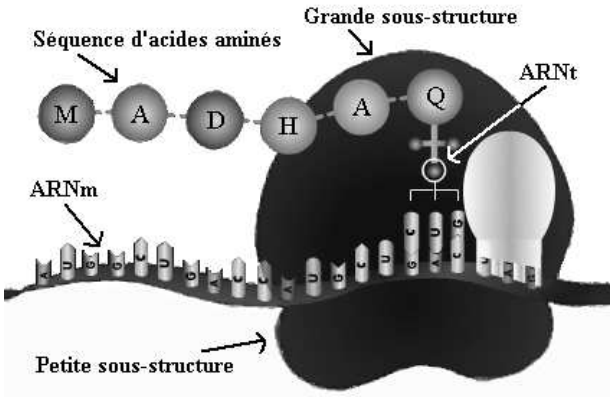



**Figure 1.4** La traduction (Lien Internet, Lexique EncycloBio)

## 1.2 La structure du génome

Dans cette section, nous reprenons la plupart des informations introduites précédemment sur la structure du génome pour montrer que cette structure peut être considérée selon différents aspects biologiques, tout en conservant certaines notions combinatoires constantes qui serviront de base à la modélisation algorithmique que nous utiliserons. Nous considérerons essentiellement deux aspects principaux : la séquence de nucléotides (Section 1.2.1) et les protéines synthétisées (Section 1.2.2).

### 1.2.1 Le génome comme séquence de nucléotides

**Structure d'un gène.** Un gène codant est une séquence de nucléotides qui sera transcrite en un ARNm. Si l'on veut détailler un peu plus précisément la structure de cette séquence, on remarque alors qu'elle est beaucoup plus longue que l'ARNm correspondant, notamment chez les eucaryotes. Cela est dû au fait qu'un gène est composé de 3 types de séquences : le promoteur, déjà mentionné, suivi d'un ensemble d'*introns* et d'*exons* qui alternent : un intron est suivi d'un exon et vice-versa, sauf pour l'exon terminant un gène.

**Introns et exons.** Les exons contiennent les séquences codantes, c'est-à-dire les futurs

codons, et seront transcrits en un ARNm. Les introns ne contiennent aucune séquence codante pour des acides aminés et ne seront pas inclus dans l'ARNm. Cependant ils peuvent contenir certaines séquences non codantes fonctionnelles, entre autres des gènes, comme des ARN non codants par exemple.

Les gènes des procaryotes et des eucaryotes sont différents en termes de la structure décrite ci-dessus : un gène de procaryote ne comporte en général qu'un exon et aucun intron, alors qu'un gène d'eucaryote peut être très complexe et contenir plusieurs exons.

**Épissage alternatif.** Dans le cas des gènes d'eucaryotes, il existe un phénomène appelé *épissage alternatif*, qui, utilisant la structure complexe en introns et exons de ces gènes, permet de produire différents ARNm à partir d'un même gène. En effet, lors de la transcription, dépendamment de l'activation ou l'inhibition de certaines séquences non codantes fonctionnelles en amont du gène, appelées *facteurs de régulation de la transcription*, certains exons ne seront pas inclus dans l'ARNm, et ne synthétiseront donc pas d'acides aminés. Cette activation/inhibition peut être automatique ou dépendre de conditions extérieures, comme la température par exemple. Cette variété dans l'assemblage des exons disponibles dans un gène est nommée *épissage alternatif* et est responsable de la grande diversité des protéines synthétisées relativement au nombre de gènes présents dans le génome (une vingtaine de milliers chez l'humain, nombre comparable au nombre de gènes du poisson Tetraodon par exemple (Jaillon O. *et al.*, 2004)). Ce phénomène d'épissage alternatif, absent chez les procaryotes explique en partie la différence apparente de complexité entre certains organismes dont les génomes sont de tailles comparables. Par exemple, les données actuelles semblent indiquer que près de 40% des gènes humains subissent un épissage (Modrek B. - Lee C., 2002).

La figure 1.5 représente le gène $\alpha$-*Tropomyosine* ($\alpha$-TM) du rat et illustre l'organisation de ce gène et des ARNm associés. Sur la première ligne, on retrouve le gène lui-même, où les rectangles représentent des exons. Les six lignes suivantes proposent différents épissages, c'est-à-dire différents choix d'exons, selon le type de cellule. Un trait plein entre deux exons représente un choix confirmé, tandis que les traits pointillées illustrent

un épissage alternatif probable, c'est-à-dire qu'aucune confirmation biologique n'existe. Les différents ARNm produits dépendent de la nature de la cellule, un gène provenant d'une cellule de muscle ne produit pas le même ARNm que s'il provenait d'une cellule de cerveau.

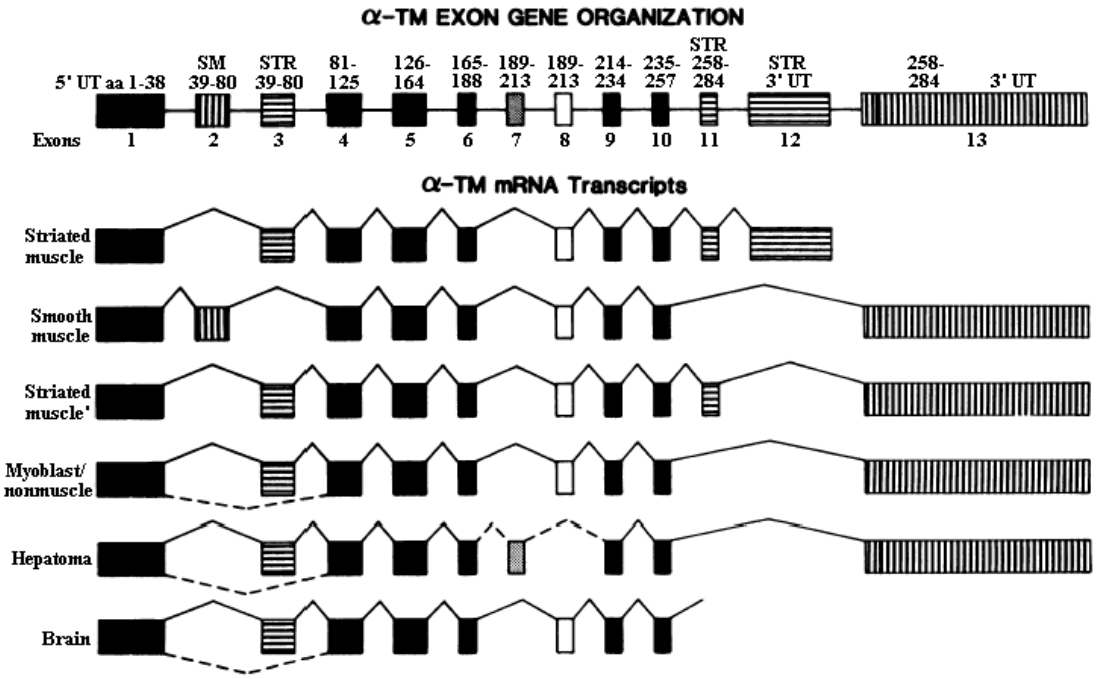



**Figure 1.5** L'épissage alternatif du gène $\alpha$-TM du rat (Wieczorek D. *et al.*, 1988)

**Génome et séquences fonctionnelles.** Un point important dans la structure du génome, et plus particulièrement des séquences fonctionnelles, réside dans la présence fréquente de plusieurs copies similaires d'un segment donné, dans un sens que nous préciserons dans la section 1.3, d'une même séquence fonctionnelle en différents endroits du génome. Ce phénomène vaut autant pour les exons de gènes que pour tout autre type de séquences fonctionnelles : gènes non codants, sites de liaison des facteurs de transcription, etc. Il est aussi important de mentionner qu'en général deux séquences fonctionnelles ne se chevauchent pas : une base donnée d'un génome appartient à au plus une séquence fonctionnelle. Une exception connue consiste en la présence, au milieu d'exons, de sites de liaison, voir (Hastings M. - Krainer A., 2001; Blanchette M., 2003),

par exemple, mais ce phénomène est rare et nous n'en tiendrons pas compte dans notre travail. Finalement, il faut aussi rappeler que toute séquence fonctionnelle est présente sur un brin du génome, ce qui lui confère une orientation par rapport aux séquences situées sur le brin opposé.

En considérant toutes ces caractéristiques, on peut proposer la modélisation combinatoire suivante : un chromosome peut être vu comme une *suite ordonnée de séquences fonctionnelles orientées*, que nous appelons *signaux*. Selon la précision désirée, ces signaux peuvent être plus ou moins fins. Par exemple, on peut se contenter de ne considérer que les gènes codants ou au contraire, descendre au niveau de la décomposition en introns et exons, et considérer toutes les séquences fonctionnelles identifiées, incluant donc des gènes non codants dans des introns.

### 1.2.2 Le génome comme une suite de protéines

Le rôle principal du génome consistant en la synthèse de protéines, il est naturel de considérer ce génome aussi sous l'angle de ces protéines. De cette façon, la modélisation la plus naturelle consiste à associer à chaque gène la protéine correspondante, ce qui revient à considérer un chromosome comme une *suite de protéines*. Il est important de noter qu'en raison de l'épissage alternatif, cette séquence n'est pas équivalente, à renommage près, à la suite des gènes.

Cependant, de la même façon que la structure d'un gène fait apparaître plusieurs types de sous-suites aux propriétés et rôles très différents, c'est-à-dire introns, exons, facteurs de transcription, etc., la séquence d'acides aminés, qui constitue la structure primaire d'une protéine, se décompose en *domaines*. Un domaine est une séquence d'acides aminés associée à une fonction biologique précise. La figure 1.6 illustre un exemple d'une protéine ayant plusieurs domaines, la protéine *trpGD* chez la bactérie *Escherichia coli*. Les numéros indiquent différents domaines attribués par la base de données PFAM, que nous détaillons à la section 2.2.4 (Bateman A. *et al.*, 2000).


**Figure 1.6** Une protéine à plusieurs domaines (Pasek S. *et al.*, 2005)

Étant donné cette décomposition de la structure primaire d'une protéine en domaines, on peut voir un chromosome comme une *suite de domaines orientés*. L'orientation étant en fait celle du gène ayant produit l'ARNm, dont la traduction a créé le domaine en question.

Les domaines de protéines sont relativement bien étudiés et il existe plusieurs bases de données les recensant, dont PFAM (Bateman A. *et al.*, 2000) est l'une des plus utilisées.

### 1.2.3 Modélisation

En considérant toutes les propriétés de la structure d'un génome que nous venons d'introduire, une caractéristique commune se dégage :

> Un génome est une suite de *séquences orientées*
> sur un alphabet de *signaux génomiques* :
> séquences fonctionnelles, gènes, protéines, domaines, etc.

La notion d'orientation est modélisée en attribuant un signe à chaque signal, soit le signe « + » (par exemple $A$) lorsque le signal est dans la direction 5' vers 3', soit le signe « - » (par exemple $\overline{A}$) lorsque le signal est dans l'autre direction.

La figure 1.7 illustre un segment d'un chromosome fictif, composé des deux brins. La suite ainsi obtenue est :

$$A\ B\ \overline{A}\ C\ \overline{C}\ \overline{D}\ E\ F\ \overline{E}\ \overline{D}.$$

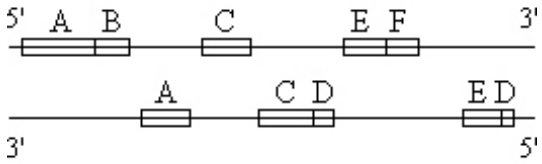



**Figure 1.7** Un exemple de suite signée

Il est important de préciser que les étiquettes attribuées aux signaux ne reflètent pas l'identité exacte entre eux, mais une certaine notion de similarité, expliquée à la section 2.2.1.

Dans la suite de ce mémoire, nous utiliserons le terme gène en lieu et place de « signal génomique », pour des raisons de concision.

Il est cependant important de conserver la propriété de non chevauchement de ces gènes. L'exemple de la figure 1.7 illustre bien que les gènes ne se chevauchent pas. On ne peut pas par exemple, considérer en même temps les gènes codants et certains signaux liés à l'épissage, ceux-ci résidant souvent dans les introns si on veut modéliser l'épissage, il faut alors raffiner le modèle au niveau des exons.

## 1.3 Évolution et génome

L'évolution des espèces et de leur génome comporte deux types de mécanismes : des mutations dans la séquence d'ADN et des réarrangements de l'ordre des gènes.

**Mutations de séquence.** À l'échelle des nucléotides, l'évolution se traduit par des mutations de nucléotides dans la séquence d'ADN, ou des insertions et des suppressions de segments de nucléotides. On visualise souvent ces mutations entre deux séquences d'ADN à l'aide d'un *alignement* de ces deux séquences, comme illustré par la figure 1.8.

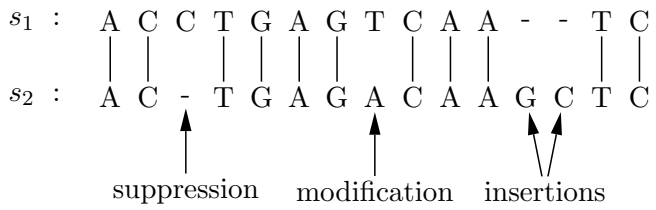



**Figure 1.8** Un alignement

Si nous comparons la séquence $s_2$ à la séquence $s_1$, le troisième nucléotide a été supprimé, le huitième a été modifié et les douzième et treizième ont été insérés.

Dans le cas où des mutations se produisent dans des séquences non fonctionnelles du génome, elles n'induisent, selon l'état actuel des connaissances, aucun changement dans le fonctionnement du génome. Par contre, des mutations se produisant dans des séquences fonctionnelles peuvent modifier la fonction de cette séquence. Une première modification est de perdre son caractère fonctionnel, ce qui arrive si la séquence est très courte et très spécifique, comme un site de liaison par exemple. Une mutation peut aussi altérer la fonction associée à cette séquence. Ce phénomène explique que les séquences fonctionnelles sont beaucoup plus conservées entre différentes espèces que les séquences non fonctionnelles : pour un même gène de l'humain et de la souris par exemple, les exons seront très similaires alors que les introns pourront être très différents. On dit que les séquences fonctionnelles sont sous *pression évolutive* pour les protéger des mutations. En d'autres termes, les mutations dans ces séquences sont très souvent non viables et ne se propagent donc pas.

Étant donné l'avancement de nombreux programmes de séquençages (Margulies E. H. *et al.*, 2003), l'analyse comparative des séquences fonctionnelles est un sujet en plein développement. Cependant nous ne nous placerons pas sous cet angle dans ce travail, et nous nous intéresserons plutôt à l'évolution par réarrangements génomiques.

**Réarrangements génomiques.** Les mécanismes d'évolution que nous prenons en compte dans ce travail ne se situent pas au niveau d'une séquence individuelle, mais à l'échelle de *segments de gènes*. Si on rappelle que l'on peut modéliser un génome comme une suite de gènes signés, on peut alors décrire un réarrangement génomique comme une action sur un segment de ces signaux qui modifie la structure combinatoire de la suite, soit en modifiant l'ordre et/ou les signes des gènes qui composent cette suite, soit en modifiant le contenu en gènes du génome. Notons qu'il existe aussi des réarrangements échangeant du matériel entre deux chromosomes (translocations), mais elles ne sont pas pertinentes dans le cas présent, car nous nous intéressons à des génomes de procaryotes, qui n'ont en général qu'un seul chromosome.

Les réarrangements les plus courants qui ne modifient pas le contenu en gènes sont :

- Les inversions : un segment signé de gènes $g_i\ g_{i+1}\ ...\ g_{j-1}\ g_j$ devient $\overline{g_j}\ \overline{g_{j-1}}\ ...\ \overline{g_{i+1}}\ \overline{g_i}$, le signe de chaque gène du segment étant changé.
- Les transpositions : un segment signé de gènes se déplace dans le génome en conservant ordre et signe des gènes du segment.
- Les transpositions inverses : une transposition d'un segment, accompagnée d'une inversion de ce même segment.

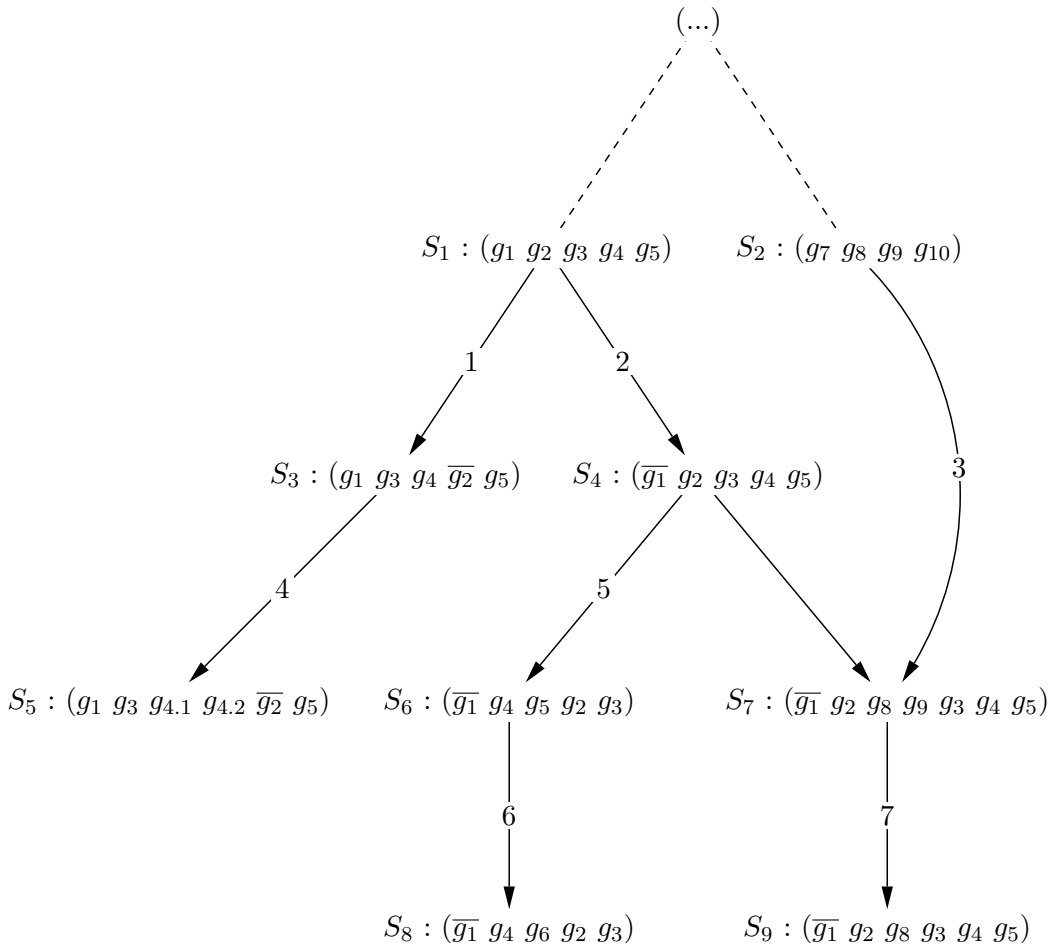
Il existe des modifications qui changent le contenu en gènes, que, par abus de langage, nous appelons aussi des réarrangements :

- Les suppressions : un segment de gènes disparaît. Du point de vue biologique, la suppression d'un gène peut être le résultat, non pas d'une disparition, mais d'une perte de la fonction de la séquence d'ADN associée suite à des mutations locales à cette séquence.
- Les mutations : un gène change d'étiquette. La mutation d'un gène est en fait un changement de fonction du gène en question, suite à des mutations de sa séquence d'ADN.
- Les duplications : un segment de gènes se duplique. La duplication est un mécanisme évolutionnaire fondamental et peut se dérouler à grande échelle. Le génome du blé

par exemple, au cours de son évolution, a subi plusieurs duplications complètes qui résultent en six copies de chaque chromosome. Les nouveaux gènes ainsi « créés » offrent à l'organisme de nouvelles facultés d'adaptation en « autorisant » le changement de fonction, c'est-à-dire l'évènement de mutation mentionné précédemment, de ces gènes pour s'adapter à de nouvelles conditions.

- Les insertions : un segment de gènes est inséré dans le génome. L'insertion d'un segment de gènes est en général dû à un *transfert latéral* (Philippe H. - Douady C. J., 2003), c'est-à-dire à un transfert de matériel génétique d'un organisme vers un autre. Par exemple, le génome des mitochondries des eucaryotes provient de l'absorption d'un génome de bactérie au cours des premiers stades de l'évolution. Les gènes ainsi insérés sont, tout comme dans le cas précédent, très susceptibles de changer de fonction pour permettre au génome de s'adapter à de nouvelles conditions.

Nous illustrons ces mécanismes d'évolution à l'aide d'un exemple simple contenant un exemple de chacun des types de réarrangements présentés à la figure 1.9.




| Évènement | Numéro de l'arête | Espèces impliquées | Gène(s) impliqué(s) |
|---|---|---|---|
| Inversion | 2 | $S_1\ S_4$ | $g_1$ |
| Transposition | 5 | $S_4\ S_6$ | $g_2\ g_3$ ou $g_4\ g_5$ |
| Transposition inverse | 1 | $S_1\ S_3$ | $g_2$ |
| Suppression | 7 | $S_7\ S_9$ | $g_9$ |
| Duplication | 4 | $S_3\ S_5$ | $g_4 \rightarrow g_{4.1}\ g_{4.2}$ |
| Transfert latéral | 3 | $S_2\ S_4\ S_7$ | $g_8\ g_9$ |
| Mutation | 6 | $S_6\ S_8$ | $g_5 \rightarrow g_6$ |

**Figure 1.9** Les évènements d'évolution

**Comparaison de gènes.** Nous concluons notre rapide survol des mécanismes d'évolution en abordant les différentes façons de définir la relation évolutionnaire entre gènes.

Étant donné deux gènes, présents dans un même génome ou dans des génomes différents, on les considère reliés évolutionnairement s'ils sont descendants d'un gène ancestral commun : on dit alors qu'ils sont *homologues*. Cependant, on peut distinguer deux types de mécanismes évolutionnaires produisant deux gènes à partir d'un gène ancestral : la *duplication* et la *spéciation*.

Les relations d'orthologie et de paralogie sont expliquées à l'aide de la figure 1.10. Les relations d'évolution entre trois espèces $S_1$, $S_2$ et $S_3$ provenant d'un même ancêtre commun sont illustrées. Il y a trois évènements de spéciation $Sp_1$, $Sp_2$ et $Sp_3$, apparaissant aux jonctions d'un « Y » inversé. Il y a aussi deux évènements de duplication $Dp_1$ et $Dp_2$, identifiés par un trait horizontal. Deux gènes dont l'ancêtre commun le plus proche se situe à la jonction d'un « Y » inversé (spéciation) sont dits *orthologues*. Deux gènes dont l'ancêtre commun le plus proche se situe sur une ligne horizontale (duplication) sont dits *paralogues*. Nous avons les exemples suivants :

$g_5$ et $g_6$ sont paralogues,
$g_4$ et $g_5$ sont paralogues,
$g_4$ et $g_6$ sont paralogues,
$g_3$ et $g_5$ sont orthologues,
$g_3$ et $g_6$ sont orthologues,
$g_2$ et $g_5$ sont paralogues,
$g_2$ et $g_6$ sont paralogues,
$g_1$ et $g_5$ sont orthologues,
$g_1$ et $g_6$ sont orthologues.

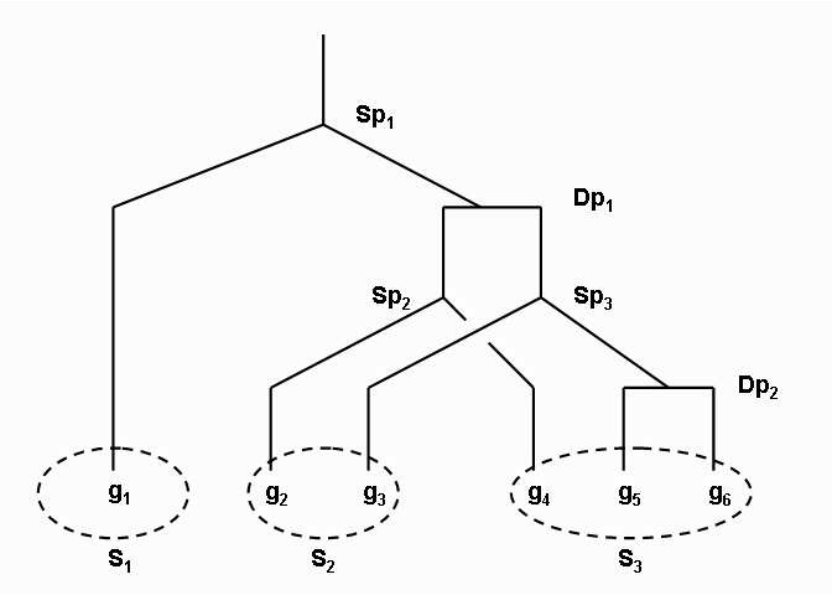



**Figure 1.10** Orthologie et paralogie (Fitch W. M., 2000)

Tel qu'illustré ci-dessus, ces relations sont de type *plusieurs à plusieurs*, c'est-à-dire qu'un gène peut être orthologue ou paralogue à plusieurs gènes.

Les relations d'orthologie et de paralogie sont réflexives, mais ne sont pas transitives (Fitch W. M., 2000). Par exemple :

$$g_5 \text{ est orthologue à } g_1,$$
$$g_1 \text{ est orthologue à } g_6,$$
$$g_5 \text{ n'est pas orthologue à } g_6.$$

L'union des relations d'orthologie et de paralogie forme la relation d'homologie. Deux gènes sont homologues s'ils sont soit orthologues ou paralogues. Cette relation d'homologie est une relation d'équivalence étant donné qu'elle est réflexive, symétrique et transitive.

## 1.4 Génomique comparée et détection de groupes de gènes

### 1.4.1 Introduction à la génomique comparée

La complétion, ces dernières années, de plusieurs programmes de séquençage de génomes, tant chez les eucaryotes (humain, souris, rat, poulet, arabette des dames, riz, mouche à fruit et moustique) que chez les procaryotes (le site du NCBI recense plus de 225 génomes bactériens complets en 2005) a ouvert la voie à un nouveau paradigme d'analyse des génomes basé sur la comparaison de génomes : la *génomique comparée*.

En effet, il est primordial de remarquer que le séquençage d'un génome n'est que la première étape de son analyse : le but final est de comprendre le fonctionnement de celui-ci. Cette analyse du génome consiste principalement à associer à toute séquence impliquée dans une fonction biologique, les gènes en particulier, une description de cette fonction. Cette description constitue *l'annotation fonctionnelle* de la séquence du génome.

Ce processus d'annotation est en fait la source des principales problématiques bioinformatiques, dont un des buts est de fournir des *prédictions* d'annotation fonctionnelle qui permettront de guider les expérimentations biologiques qui les valideront ou les infirmeront.

Le principe de l'analyse comparative des génomes repose sur le postulat fondamental suivant, que nous avons déjà rapidement mentionné dans la section précédente (Roskin K. M. *et al.*, 2003) :

> Les régions fonctionnelles d'un génome sont soumises à une pression,
> la pression évolutive, qui les protège plus des mutations
> que les séquences non fonctionnelles.

Ce principe vaut autant pour l'évolution au niveau des séquences de nucléotides que pour l'organisation des groupes de gènes.

Par exemple, considérons le gène *RPL18A*, présent chez plusieurs vertébrés. La figure 1.11, obtenue sur le navigateur de génome de l'université de Californie à Santa Cruz (Lien Internet, Le navigateur de génome de l'université de Californie de Santa Cruz), présente la structure en exons et introns de ce gène chez l'humain ainsi qu'une mesure de la conservation de la composition en nucléotides de ce gène chez d'autres vertébrés (chimpanzé, souris, rat, poulet, fugu et poisson zèbre). Les rectangles noirs des trois premières lignes (RPL18A, CCDS12367.1 et « RefSeq Genes »), illustrent les régions codantes du gène *RPL18A*. La ligne « Spliced ESTs » illustre des séquences codantes chez l'humain provenant de l'épissage alternatif. Les dernière lignes (« chimp », « dog », « mouse », « rat », « chicken », « fugu » et« zebrafish ») sont les génomes de ces espèces alignés au gène *RPL18A*. L'histogramme de la ligne « Conservation » illustre la conservation des régions des génomes des espèces. Lorsque des séquences des génomes sont très conservées, elles correspondent aux régions codantes du gène *RPL18A*.

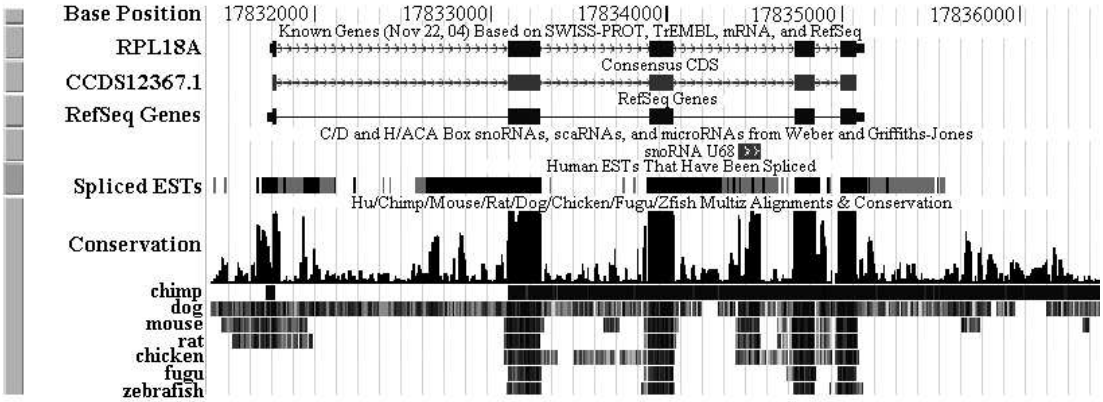



**Figure 1.11** Conservation de séquences codantes

L'analyse de cette conservation permet de faire nettement la distinction entre les exons, séquences codantes et donc soumises à une pression évolutive, et les introns non codants. La seule séquence très conservée mais présente dans un intron, au milieu du troisième intron, est une séquence fonctionnelle, à savoir le gène non codant du *snoRNA U68*, présent aussi chez le chimpanzé, la souris, le rat et le poulet. Elle est représentée par un rectangle noir, qui se situe entre les lignes « RefSeq Genes » et « Spliced ESTs ».

Cet exemple illustre bien l'effet de la pression évolutive sur les séquences fonctionnelles, contrairement aux séquences non fonctionnelles, et comment l'analyse comparative est un outil fondamental dans l'annotation de génomes (Margulies E. H. *et al.*, 2003; Blanchette M., 2003; Boffelli D. *et al.*, 2004; Blanchette M. - Tompa M., 2002).

Le postulat de conservation grâce à la pression évolutive ne vaut pas que pour l'évolution par mutation de séquences, mais aussi pour l'évolution par réarrangements génomiques : si un ensemble de gènes proches, en termes de position physique sur un génome, sont impliqués dans un même processus biologique, et que leur proximité physique est importante vis-à-vis de cette fonction, alors il est probable que ces gènes seront proches dans les autres génomes dans lesquels ils ont la même fonction.

Ce phénomène se constate, par exemple, par la présence *d'opérons* dans les génomes de procaryotes. Un opéron est une suite de gènes consécutifs dont la transcription est régulée par un unique promoteur, du fait de leur implication dans un même ensemble de processus biochimiques. Par exemple, l'opéron *Tryptophane* est composé de 6 gènes consécutifs impliqués dans le processus de transformation de l'acide chorismique en tryptophane. Ces 6 gènes, se retrouvent dans cet ordre chez *E. coli* :

$$\overline{trpA},\ \overline{trpB},\ \overline{trpC},\ \overline{trpD},\ \overline{trpE} \text{ et } \overline{trpL},$$

mais renversés chez *S. typhimurium* :

$$trpL,\ trpE,\ trpD,\ trpC,\ trpB \text{ et } trpA,$$

une bactérie proche, du point de vue évolutionnaire, de *E. coli*. Par contre, chez *H. influenza*, une bactérie éloignée, on ne retrouve que cinq gènes de l'opéron *Tryptophane*, répartis en deux groupes :

$$trpE,\ trpG,\ \text{HI1388.1},\ trpD,\ trpC,\ \overbrace{\cdot\ \cdot\ \cdot}^{environ\ 40\ gènes}\ trpB,\ trpA,$$

où *HI1388.1* est l'identificateur d'un protéine hypothétique. Cet exemple permet d'illustrer que la pression évolutive tend à conserver des gènes fonctionnellement reliés ensemble, même si ce groupement peut être plus ou moins bien conservé dans le cadre d'organismes distants. Le phénomène d'opérons est limité aux génomes de procaryotes, mais la notion de groupes de gènes conservés s'applique aussi aux génomes d'eucaryotes (Singer G. at al., 2004).

### 1.4.2 Identification de groupes de gènes

Le cadre général de notre travail est l'étude comparative de génomes complets, considérés sous l'angle de suites de gènes signés, à des fins d'analyse de l'évolution ou d'assistance à l'annotation fonctionnelle de ces génomes. Comme il l'a été décrit dans la section précédente, un des aspects importants dans une telle analyse comparative à l'échelle de génomes complets réside dans la détection de groupes de gènes apparaissant comme sous-suites compactes dans plusieurs des génomes étudiés. En effet, un tel regroupement conservé à travers l'évolution indique un possible lien fonctionnel entre les gènes du groupe. De plus, l'analyse de la structure combinatoire d'un groupe de gènes que l'on retrouve dans plusieurs génomes peut fournir des informations importantes sur les réarrangements que ce groupe a subi durant l'évolution (Bourque G. *et al.*, 2004).

Rappelons que le but de ce mémoire est de présenter un algorithme prenant, comme entrée, un ensemble de génomes complets modélisés par des suites de gènes signés. L'algorithme produit, en sortie, une liste de groupes de segments de ces génomes vérifiant que toute paire de segments d'un même groupe a un *contenu en gènes relativement similaire.*

Cette notion de similarité se situe au niveau des fonctions biologiques des gènes de ces segments. C'est-à-dire qu'un groupe de segments sera produit si l'ensemble des fonctions biologiques des gènes est semblable.

Nous nous intéressons à une question centrale dans l'analyse comparative de génomes : comment déterminer que deux signaux génomiques, au sens introduit précédemment,

sont similaires du point de vue de leur fonction biologique ? Cette question est importante, car le but des programmes de génomique à grande échelle est de comprendre la fonction biologique des signaux présents dans un génome. Mais l'afflux de données (séquences de génomes, gènes et protéines entre autres) est tel qu'elles ne peuvent être analysées par les méthodes expérimentales traditionnelles de biologie. D'où l'intérêt d'avoir des outils bioinformatiques capables, étant donné un signal génomique nouvellement obtenu, de pouvoir en mesurer la *vraisemblance* avec les signaux génomiques déjà connus et analysés, c'est-à-dire de fonction connue.

Nous comparons des suites de gènes en nous basant sur le concept d'homologie. C'est à partir de ce concept que les suites sont construites tel que décrit dans le chapitre II. L'homologie est très importante, étant donné qu'elle est à la base de notre problématique.

Sans entrer immédiatement dans les détails de la méthode que nous avons utilisée, nous avons obtenu quelques groupes de gènes intéressants en comparant *Bacillus subtilis* et *Escherichia coli* illustrés aux figures 1.12, 1.13 et 1.14. Les deux premières lignes de chacun des exemples sont les deux segments appartenant au groupe obtenu, dont les gènes ont été étiquetés par des lettres. Lorsque deux gènes sont déclarés homologues par notre algorithme, ils sont étiquetés par la même lettre. Chaque groupe est composé d'un segment appartenant à *Bacillus subtilis* (BS.) et d'un autre appartenant à *Escherichia coli* (EC.). Les autres lignes, une pour chaque gène des segments, contiennent l'information relative à ces gènes. Par exemple, sur la première ligne de la figure 1.12, nous avons le gène *def* appartenant à *Bacillus subtilis* (BS.). Ce gène a été étiqueté temporairement par le numéro 497 par notre algorithme et a été ré-étiqueté $A$ pour des raisons de lisibilité. La position de ce gène, en terme d'acides aminés sur son génome est 1645812, son orientation est «  positive » et les biologistes lui ont donné l'annotation «  polypeptide deformylase ».

BS. : A B C
EC. : A B C

A : 497 BS.def 1645812 + polypeptide deformylase
B : 498 BS.fmt 1646299 + methionyl-tRNA formyltransferase
C : 499 BS.yloM 1647239 + unknown ; similar to RNA-binding Sun protein
A : 497 EC.def 7645609 + Peptide deformylase, N-formylmethionylaminoacyl-tRNA deformylase
B : 498 EC.fmt 7646133 + Methionyl-tRNA formyltransferase
C : 499 EC.rsmB 7647126 + 16S rRNA m5C967 SAM-dependent methyltransferase

**Figure 1.12** Exemple de résultats a)

BS. : A B C D E F G
EC. : A B C D G E F

A : 535 BS.rpsB 1717226 + ribosomal protein S2
B : 536 BS.tsf 1718068 + elongation factor Ts
C : 537 BS.pyrH 1719095 + uridylate kinase
D : 538 BS.frr 1719819 + ribosome recycling factor
E : 539 BS.uppS 1720507 + probable undecaprenyl pyrophosphate synthetase
F : 540 BS.cdsA 1721293 + phosphatidate cytidylyltransferase
G : 541 BS.dxr 1722164 + probable 1-deoxy-D-xylulose-5-phosphate reductoisomerase
A : 535 EC.rpsB 4404156 + 30S ribosomal subunit protein S2
B : 536 EC.tsf 4405139 + EF-Ts, elongation factor for transcription, stable
C : 537 EC.pyrH 4406137 + UMP kinase
D : 538 EC.frr 4407154 + Ribosome recycling factor ; essential gene ; dissociates ribosomes from mRNA after termination of translation
G : 541 EC.dxr 4407803 + 1-deoxy-D-xylulose 5-phosphate reductoisomerase forms 2-C-methyl-D-erythritol 4-phosphate ; alternative nonmevalonate pathway for terpenoid biosynthesis
E : 539 EC.yaeS 4409185 + Function unknown
F : 540 EC.cdsA 4409959 + CDP-diglyceride synthase

**Figure 1.13** Exemple de résultats b)

BS. : A B C
EC. : A C B

A : 678 BS.yqiQ 2506493 - unknown ; similar to phosphoenolpyruvate mutase
B : 679 BS.mmgE 2507416 - function unknown
C : 327 BS.mmgD 2508849 - citrate synthase III
A : 678 EC.prpB 4562188 + Propionate catabolism operon, Salmonella homology and some expression information
C : 327 EC.prpC 4563518 + Propionate catabolism operon, Salmonella homology and some expression information
B : 679 EC.prpD 4564721 + Propionate catabolism operon, Salmonella homology and some expression information

**Figure 1.14** Exemple de résultats c)

Dans chacun de ces groupes, nous pouvons remarquer la similarité dans la composition des segments de gènes. Le groupe de la figure 1.12 contient deux segments identiques, pour le groupe de la figure 1.13 nous pouvons supposer qu'il y a eu une transposition du gène $G$, tandis que pour le groupe de la figure 1.14, il y a peut-être eu plusieurs réarrangements.

Ce qui est plus intéressant que la similarité dans la composition des segments, c'est que chaque groupe contient un gène dont l'annotation est «  function unknown », ce qui veut dire que la fonction de ce gène est inconnue par les biologistes. Dans le groupe de la figure 1.12, il s'agit du gène *yloM* de *Bacillus subtilis*, dans le groupe de la figure 1.13 c'est le gène *yaeS* de *Escherichia coli* et dans le groupe de la figure 1.14, c'est le gène *mmgE* de *Bacillus subtilis*. En examinant ces gènes de fonction inconnue à l'intérieur de leur groupe de gènes, c'est-à-dire en reliant chaque paire de gènes ayant la même étiquette, il est possible d'émettre une hypothèse sur leur fonction. Par exemple, le gène *yloM* de *Bacillus subtilis* du groupe de la figure 1.12 est homologue au gène *rsmB* de *Escherichia coli*, on peut supposer que la fonction biologique du gène *yloM* est semblable à celle du gène *rsmB*, ce qui reste à vérifier en laboratoire. Il en est de même pour les deux autres exemples : le gène *yaeS* de *Escherichia coli* peut avoir une fonction similaire au gène *uppS* de *Bacillus subtilis* et le gène *mmgE* de *Bacillus subtilis* peut avoir une

fonction similaire au gène *prpD* de *Escherichia coli*.

Comme nous venons de le mentionner, un des intérêts du calcul des groupes de gènes vise l'annotation. Nous sommes en mesure d'émettre des hypothèse quant à l'annotation d'un gène en nous basant sur nos groupes de gènes calculés. Deux autres intérêts du calcul des groupes de gènes sont au niveau de l'évolution et au niveau de l'orthologie.

Dans la section suivante, nous présentons les modèles de Bergeron *et al.* 2.3.2, He et Goldwasser 2.3.2 et Pasek *et al.* 2.3.2, dont certains imposent une restriction sévère sur les définitions ou donnent lieu a des algorithmes exponentiels. Dans le chapitre IV, nous présentons notre modèle sans restriction sévère sur nos définitions et un algorithme non exponentiel, c'est-à-dire polynomial.

# Chapitre II

# LES REGROUPEMENTS DE GÈNES

## 2.1 Introduction

Nous avons vu, au chapitre précédent, qu'un génome pouvait être modélisé par une suite de gènes signés, dans laquelle les gènes homologues ont tous la même étiquette. La comparaison de deux ou de plusieurs génomes se fait en deux étapes fondamentales :

- L'identification des familles de gènes homologues. Chaque gène d'une même famille aura la même étiquette, quelque soit le génome considéré.
- L'identification de groupes ou d'équipes de gènes, appartenant à des familles différentes, dont la proximité ou la fonction est conservée d'un génome à l'autre.

Notons que, dans la littérature, ces deux étapes sont désignées par le même terme *regroupement de gènes* (« gene clusters »), ce qui porte souvent à confusion. Nous conserverons donc le terme « famille de gènes » pour l'étape d'identification des gènes homologues, et le terme « groupe de gènes » pour désigner un ensemble de gènes de familles différentes.

## 2.2 La construction de familles d'homologues

Une famille de gènes est un ensemble de gènes tous homologues entre eux. La relation d'homologie est une relation d'équivalence et une famille de gènes est une classe de cette relation.

Idéalement, si nous connaissions les relations d'orthologie et de paralogie, telles que présentées à la section 1.3, nous pourrions se servir d'elles pour l'étiquetage des gènes homologues. Mais nous ne pouvons que les approximer, car nous ne connaissons pas les génomes ancestraux ni les évènements de spéciation et de duplication.

La difficulté est de trouver une bonne approximation de cette relation d'équivalence, qui se fait généralement à l'aide d'algorithmes de comparaison de séquences. Ces algorithmes calculent une mesure de similarité ou de distance entre des séquences de nucléotides ou d'acides aminés. Nous nous servons ensuite des résultats de ces comparaisons pour classer les gènes en familles. Cependant, il arrive que les biologistes classent des gènes dans une même famille pour d'autres raisons que la similarité de leurs séquences, tout comme il arrive que des séquences similaires ne fassent pas partie de la même famille. C'est le cas notamment des bases de données COG et PFAM discutées dans les sections 2.2.3 et 2.2.4.

Lorsque les données le permettent, une séquence donnée appartient à une et une seule famille, et l'étiquetage est facile. Malheureusement, la nature ne suit pas toujours un modèle mathématique et nous devons prendre certaines décisions afin d'étiqueter ces séquences. Nous présenterons quelques exemples comportant certains problèmes.

### 2.2.1 L'utilisation de mesures de similarité pour la classification

Dans le problème qui nous intéresse, nous étudions des familles de gènes homologues, issues d'un ensemble de génomes. Nous évaluons cette relation d'homologie par des mesures de similarité. L'outil le plus utilisé pour déterminer la similarité de séquences est BLAST, présenté à la section 2.2.2.

Nous mesurons la similarité entre deux séquences $s_1$ et $s_2$ à l'aide d'une fonction $f(s_1, s_2)$ à valeurs numériques. Cette fonction peut être plus ou moins complexe :

- dans un cas, caricaturalement simple, $f(s_1, s_2) = 1$ si $s_1$ et $s_2$ ont exactement la même séquence et 0 sinon ;
- l'exemple le plus classique est sans doute le cas où $f(s_1, s_2)$ est la *distance d'édition* (Gusfield D., 1997), ou une de ses nombreuses variantes, entre $s_1$ et $s_2$ ;
- l'outil le plus utilisé en génomique est BLAST, présenté à la section 2.2.2, qui calcule plusieurs paramètres pour déterminer la similarité entre $s_1$ et $s_2$.

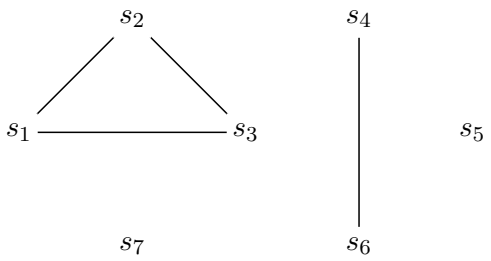
La vraie relation d'homologie, celle décrite par les génomes ancestraux et les évènements de spéciation et de duplication est symétrique, réflexive et transitive, mais cela n'est pas le cas de toutes les mesures de similarité. Par exemple, la distance d'édition est symétrique, tandis que la mesure de similarité de BLAST, selon les options d'exécution choisies, peut ne pas être symétrique.

Soit une mesure de similarité $f$ et un ensemble $E$ de signaux génomiques. Une famille de gènes est, de manière générale, un sous-ensemble de $E$ défini en termes de propriétés de la fonction $f$. On représente habituellement les éléments de $E$ comme les sommets d'un graphe orienté dont les arcs sont valués par $f$, ce qui revient à définir la notion de famille de gènes en terme de sous-graphe de ce graphe.

**Exemple 2.2.1** Considérons la mesure de similarité $f$ définie par le tableau suivant :

| $f$ | $s_1$ | $s_2$ | $s_3$ | $s_4$ | $s_5$ | $s_6$ | $s_7$ |
|---|---|---|---|---|---|---|---|
| $s_1$ | | 1 | 1 | 0 | 0 | 0 | 0 |
| $s_2$ | 1 | | 1 | 0 | 0 | 0 | 0 |
| $s_3$ | 1 | 1 | | 0 | 0 | 0 | 0 |
| $s_4$ | 0 | 0 | 0 | | 0 | 1 | 0 |
| $s_5$ | 0 | 0 | 0 | 0 | | 0 | 0 |
| $s_6$ | 0 | 0 | 0 | 1 | 0 | | 0 |
| $s_7$ | 0 | 0 | 0 | 0 | 0 | 0 | |

et le graphe $G$ dont les sommets et les arêtes sont :




Nous obtenons, en considérant les composantes connexes du graphe $G$, une relation d'équivalence où les familles de gènes sont les quatre classes d'équivalence suivantes :

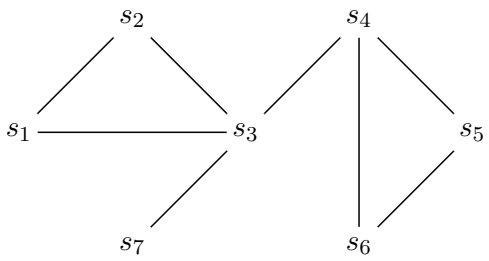
- $\{s_1,s_2,s_3\}$
- $\{s_4,s_6\}$
- $\{s_5\}$
- $\{s_7\}$

En général, les mesures de similarité ne produisent pas des résultats aussi simples. Considérons l'exemple suivant :

**Exemple 2.2.2** Soit la mesure de similarité $f$ définie par le tableau suivant :

| $f$ | $s_1$ | $s_2$ | $s_3$ | $s_4$ | $s_5$ | $s_6$ | $s_7$ |
|---|---|---|---|---|---|---|---|
| $s_1$ | | 1 | 1 | 0 | 0 | 0 | 0 |
| $s_2$ | 1 | | 1 | 0 | 0 | 0 | 0 |
| $s_3$ | 1 | 1 | | 1 | 0 | 0 | 1 |
| $s_4$ | 0 | 0 | 1 | | 1 | 1 | 0 |
| $s_5$ | 0 | 0 | 0 | 1 | | 1 | 0 |
| $s_6$ | 0 | 0 | 0 | 1 | 1 | | 0 |
| $s_7$ | 0 | 0 | 1 | 0 | 0 | 0 | |

et le graphe $G$ suivant :




Le relation d'équivalence obtenue n'est pas intéressante puisque tout les gènes sont dans la même classe. Nous devons imposer une restriction afin de scinder cette classe. Si nous décidons que les classes d'équivalence sont les cliques maximales de $G$, nous obtenons les familles suivantes :

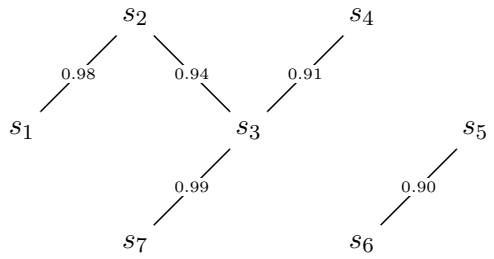
- $\{s_1,s_2,s_3\}$
- $\{s_4,s_5,s_6\}$
- $\{s_3,s_4\}$
- $\{s_3,s_7\}$

On peut noter dans cet exemple, que les familles de gènes obtenues ne forment pas une partition de l'ensemble des gènes. En effet, le gène $s_3$ est présent dans trois familles et le gène $s_4$, dans deux familles. Dans le prochain exemple, au lieu d'imposer une restriction après la construction du graphe, nous choisissons une mesure de similarité plus complexe et nous imposons une restriction avant la construction de ce graphe.

**Exemple 2.2.3** Considérons la mesure de similarité $f$ définie par le tableau suivant :

| $f$ | $s_1$ | $s_2$ | $s_3$ | $s_4$ | $s_5$ | $s_6$ | $s_7$ |
|---|---|---|---|---|---|---|---|
| $s_1$ | | 98% | 64% | 0 | 0 | 0 | 0 |
| $s_2$ | 98% | | 94% | 0 | 0 | 0 | 0 |
| $s_3$ | 64% | 94% | | 91% | 0 | 0 | 99% |
| $s_4$ | 0 | 0 | 91% | | 78% | 88% | 0 |
| $s_5$ | 0 | 0 | 0 | 78% | | 90% | 0 |
| $s_6$ | 0 | 0 | 0 | 88% | 90% | | 0 |
| $s_7$ | 0 | 0 | 99% | 0 | 0 | 0 | |

et le graphe $G$ suivant où nous n'avons conservé que les arêtes de poids $\geq 0.9$ :




Nous obtenons une relation d'équivalence où les familles de gènes sont les deux classes d'équivalence suivantes :

- $\{s_1,s_2,s_3,s_4,s_7\}$
- $\{s_5,s_6\}$

Dans cet exemple, contrairement à l'exemple 2.2.2, les familles de gènes obtenues forment une partition des sommets du graphe et ce, peu importe les valeurs du tableau de départ.

### 2.2.2 Mesures de similarité

Lorsque deux signaux sont des séquences de nucléotides ou d'acides aminés, par exemple des gènes ou des protéines, on mesure habituellement la similarité entre eux en comparant leur séquence.

#### Distance d'édition et alignement optimal

La façon la plus classique de comparer deux séquences est sans doute la *distance d'édition*. Cette distance, entre deux séquences ou deux chaînes de caractères, est le nombre minimum de transformations (insertions, suppressions, mutations) pour passer d'une séquence à l'autre. Par exemple, on peut passer de la séquence « $AABCB$ » à la séquence « $ABABD$ » en seulement trois transformations :

1) une insertion : « $A\underline{B}ABCB$ » ;

2) une suppression : « $ABABC\_$ » ;

3) ainsi qu'une mutation : « $ABAB\underline{D}$ ».

Cette comparaison s'illustre par un *alignement*, c'est-à-dire par la superposition des deux séquences où chaque transformation peut être visualisée, telle qu'illustrée ci-dessous. Un tiret dans la séquence du bas représente une insertion, un tiret dans la séquence du haut représente une suppression. Lorsque deux caractères non identiques sont superposés, c'est une mutation qui est représentée, et on trace un trait vertical vis-à-vis deux caractères identiques lorsqu'il n'y a pas de transformation :

$$\begin{array}{l} ABABD\text{-} \\ |\ \ |\ | \\ A\text{-}ABCB \end{array}$$

La distance d'édition entre deux séquences $s_1$ et $s_2$, de longueur respectivement $n$ et $m$, se calcule avec un algorithme de programmation dynamique (Gusfield D., 1997). Si $D(i,j)$ est la distance entre les préfixes $s_1[1..i]$ et $s_2[1..j]$, on a alors les équations de programmation dynamique suivantes :

$$\text{Initialisation} \begin{cases} D(1,j) = j-1 \\ D(i,1) = i-1 \end{cases} ;$$

$$D(i,j) = \min \begin{cases} D(i-1,j) & + & 1, \text{ si } i > 1 \\ D(i,j-1) & + & 1, \text{ si } j > 1 \\ D(i-1,j-1) & + & \delta(i,j), \text{ si } i > 1 \text{ et } j > 1 \end{cases} ,$$

où $\delta(i,j) = 0$ si $s_1[i] = s_2[j]$ et 1, sinon.

**Exemple 2.2.4** La distance d'édition est donnée par la valeur de $D(n,m)$. Dans le cas de notre exemple, la distance d'édition est de 3, donnée par la case $D(5,5)$, de la table 2.1.

| | | $A$ | $A$ | $B$ | $C$ | $B$ |
|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 |
| $A$ | 1 | 0 | 1 | 2 | 3 | 4 |
| $B$ | 2 | 1 | 1 | 1 | 2 | 3 |
| $A$ | 3 | 2 | 1 | 2 | 2 | 3 |
| $B$ | 4 | 3 | 2 | 1 | 2 | 2 |
| $D$ | 5 | 4 | 3 | 2 | 2 | 3 |

**Table 2.1** Calcul de la table des distances entre les préfixes de deux mots

Toutefois, la complexité temporelle de cet algorithme et de ses variantes est quadratique en la longueur des séquences, ce qui est beaucoup trop lent pour les utiliser sur les bases de données qui peuvent contenir des milliers de séquences, et des séquences de plusieurs millions de caractères. Il est donc essentiel de faire appel à des méthodes plus rapides.

### Méthodes heuristiques

Une méthode heuristique est un algorithme qui produit une solution approximative. La solution n'est pas nécessairement la solution optimale, mais en est proche en général. Étant donné que les heuristiques calculent une approximation, ces algorithmes s'exécutent généralement plus rapidement que ceux qui calculent une solution exacte.

Il existe plusieurs méthodes heuristiques pour la comparaison de séquences, mais BLAST (Basic Local Alignment Search Tool) est probablement une des plus utilisée (Altschul S. *et al.*, 1990). Le but de BLAST est surtout de bien retrouver des séquences proches, c'est-à-dire à courte distance l'une de l'autre. Le principe de BLAST repose sur le fait qu'un alignement entre deux séquences proches comporte des sous-alignements exacts (sans insertion et suppression). L'heuristique de BLAST consiste à calculer ces sous-alignements exacts, appelés *ancres*, et à les combiner en un alignement final. La figure 2.1 illustre un exemple du principe des ancres, où chacun des traits représente une ancre de longueur 2 ou plus. Une fois que ces ancres sont calculées, BLAST les assemble pour

obtenir un ensemble d'alignements plus longs, mais qui ne sont plus exacts : lorsque plusieurs ancres relativement proches forment plus ou moins une diagonale, elles définissent un tel alignement. Dans notre exemple, illustré par les figures 2.2 et 2.3, les ancres formant l'alignement final des deux séquences sont tracées en gras sur la figure 2.2. Le meilleur alignement ainsi obtenu est celui montré par la figure 2.3. Toutefois, il existe d'autres assemblages d'ancres intéressants bien qu'ils soient moins grands, comme ceux montrés par la figure 2.4. Ces assemblages d'ancres détectent des régions hautement similaires entre les deux séquences, par exemple les domaines de protéines qui sont des sous-séquences de gène très conservées. Dans ce cas-ci, il est intéressant de détecter ces alignements locaux de sous-séquences plutôt que l'alignement global.

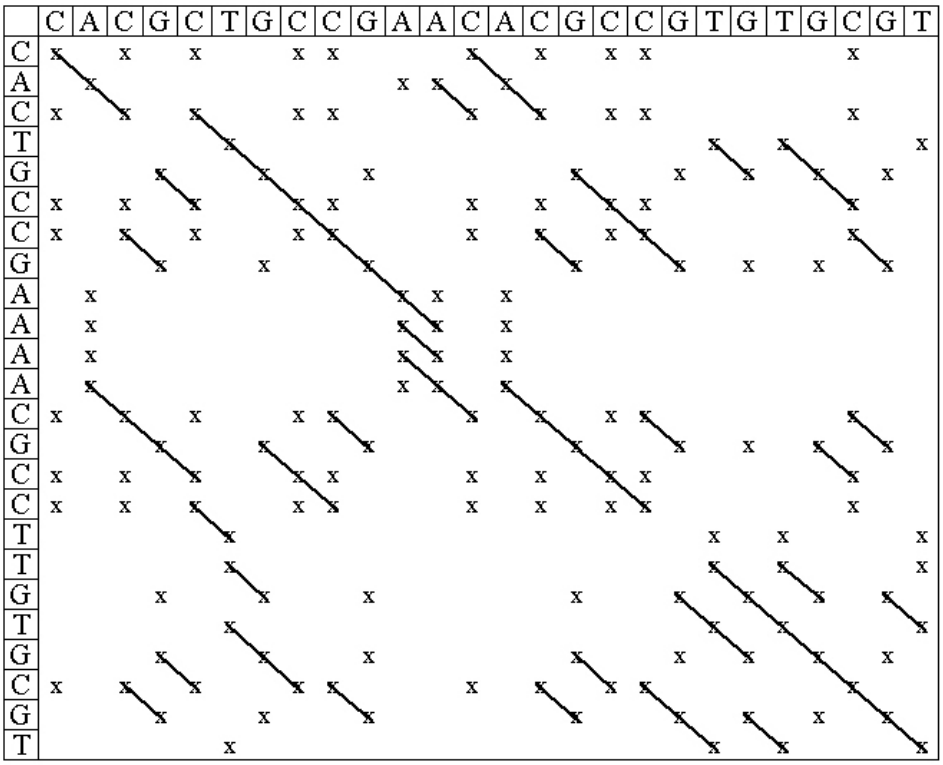



**Figure 2.1** Les ancres

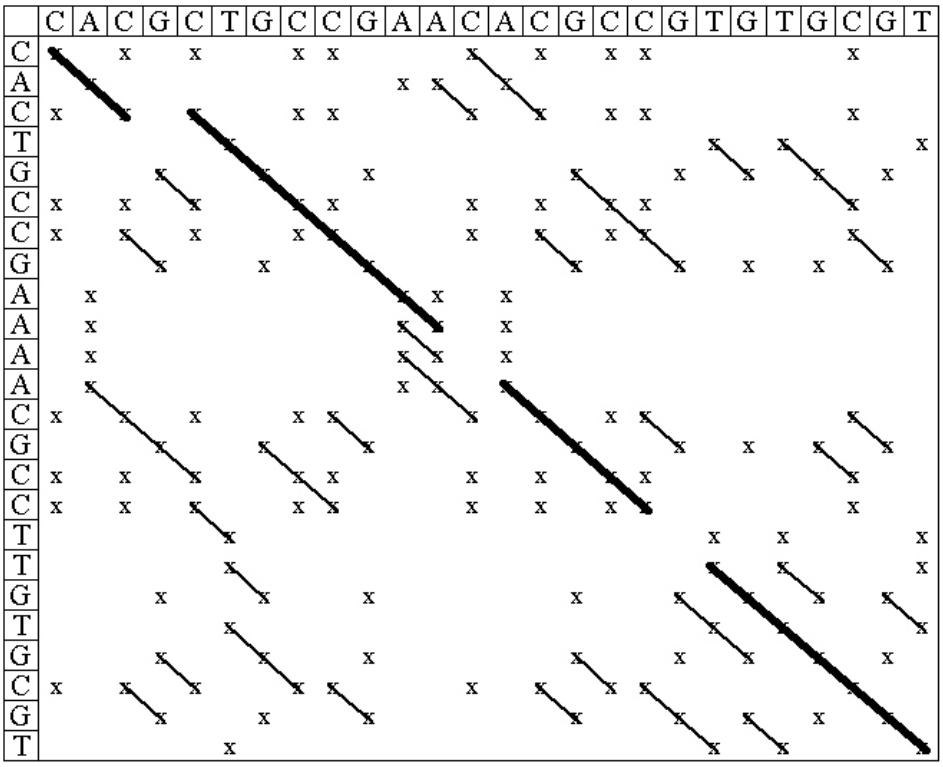



**Figure 2.2** Les ancres de l'alignement

*CACGCTGCCGAACACGCCGTGTGCGT*
*CAC--TGCCGAAAACGCCTTGTGCGT*

**Figure 2.3** L'alignement

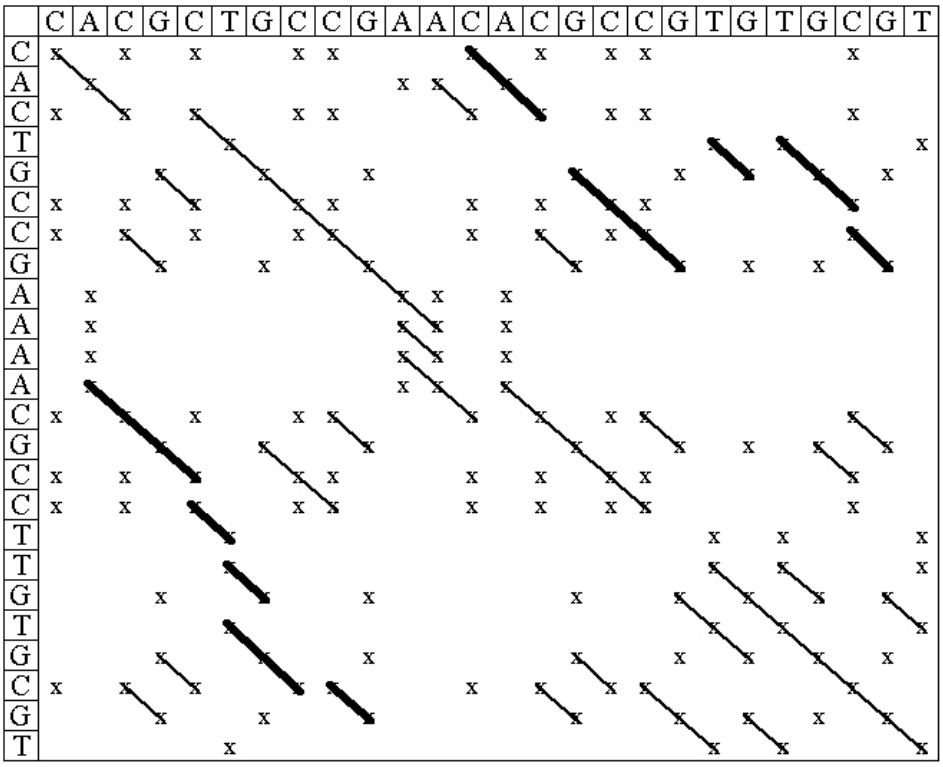



**Figure 2.4** Les ancres de petits alignements

Lorsque BLAST compare deux séquences, le résultat de cette comparaison est un ensemble d'alignements, chacun étant accompagné d'un ensemble d'informations numériques, statistiques et combinatoires. Chacune des lignes de la figure 2.5 est un alignement contenant : un lien Internet, une annotation de la séquence et deux paramètres statistiques. Le lien internet nous dirige vers plus d'informations, tel qu'illustré par la figure 2.6. Ces deux paramètres statistiques sont le « score » et la valeur $e$. Le « score » est basé sur la valeur de l'alignement, en terme du nombre d'insertions, de suppressions et d'alignements (avec ou sans mutation(s)), et est normalisé par rapport à la longueur de l'alignement. Il peut donc être utilisé pour comparer des alignements différents. En cliquant sur le « score », un lien Internet nous dirige vers une visualisation de l'alignement, tel qu'illustré par la figure 2.6. La valeur $e$ indique le niveau d'importance de l'alignement obtenu. Plus la valeur $e$ est petite, meilleur est l'alignement : c'est-à-dire

que l'alignement est statistiquement plus significatif qu'un alignement aléatoire. Les paramètres combinatoires fournis par BLAST sont illustrés sur la figure 2.6. Ces paramètres combinatoires sont les longueurs des séquences (« Length » et le dénominateur de « Identities ») et celle de l'alignement (le numérateur de « Identities ») et le pourcentage d'identité en terme de caractères (le pourcentage de « Identities »), ainsi que l'alignement lui-même sont également des informations fournies par BLAST.

| Sequences producing significant alignments: | | Score (bits) | E Value | |
|---|---|---|---|---|
| gi\|16077069\|ref\|NP_387882.1\| | dnaA [Bacillus subtilis subsp.... | 882 | 0.0 | G |
| gi\|52783856\|ref\|YP_089685.1\| | DnaA [Bacillus licheniformis D... | 845 | 0.0 | |
| gi\|52145208\|ref\|YP_081620.1\| | chromosomal replication initia... | 754 | 0.0 | G |
| gi\|42779082\|ref\|NP_976329.1\| | chromosomal replication initia... | 754 | 0.0 | G |
| gi\|30018279\|ref\|NP_829910.1\| | Chromosomal replication initia... | 752 | 0.0 | G |
| gi\|23193191\|gb\|AAN14420.1\| | DnaA [Bacillus weihenstephanensis] | 748 | 0.0 | |
| gi\|15612564\|ref\|NP_240867.1\| | initiation of chromosome repli... | 716 | 0.0 | G |
| gi\|23097456\|ref\|NP_690922.1\| | chromosomal replication initia... | 697 | 0.0 | G |
| gi\|46906225\|ref\|YP_012614.1\| | chromosomal replication initia... | 662 | 0.0 | G |
| gi\|16799080\|ref\|NP_469348.1\| | Chromosomal replication initia... | 662 | 0.0 | G |
| gi\|15893299\|ref\|NP_346648.1\| | DNA replication initiator prot... | 565 | e-159 | G |
| gi\|18308983\|ref\|NP_560917.1\| | chromosomal replication initia... | 560 | e-158 | G |
| gi\|23121455\|ref\|ZP_00103745.1\| | COG0593: ATPase involved in ... | 554 | e-156 | |
| gi\|15922991\|ref\|NP_370525.1\| | chromosomal replication initia... | 538 | e-151 | G |
| gi\|49482254\|ref\|YP_039478.1\| | chromosomal replication initia... | 538 | e-151 | G |
| gi\|48859624\|ref\|ZP_00313556.1\| | COG0593: ATPase involved in ... | 535 | e-151 | |
| gi\|27466919\|ref\|NP_763556.1\| | chromosomal replication initia... | 531 | e-149 | G |
| gi\|49235562\|ref\|ZP_00329629.1\| | COG0593: ATPase involved in ... | 531 | e-149 | |
| gi\|51891139\|ref\|YP_073830.1\| | chromosomal replication initia... | 519 | e-146 | G |
| gi\|47093227\|ref\|ZP_00231000.1\| | chromosomal replication init... | 518 | e-146 | |
| gi\|20806543\|ref\|NP_621714.1\| | ATPase involved in DNA replica... | 513 | e-144 | G |
| gi\|28209866\|ref\|NP_780810.1\| | chromosomal replication initia... | 509 | e-143 | G |
| gi\|29374662\|ref\|NP_813814.1\| | chromosomal replication initia... | 478 | e-133 | G |
| gi\|48824924\|ref\|ZP_00286238.1\| | COG0593: ATPase involved in ... | 473 | e-132 | |

**Figure 2.5** Le résultat de BLAST lorsque la séquence du gène *DnaA* de *Bacillus subtilis* lui est soumise

```
>gi|16077069|ref|NP_387882.1|  dnaA [Bacillus subtilis subsp. subtilis str. 168]
gi|2632268|emb|CAB11777.1|  dnaA [Bacillus subtilis subsp. subtilis str. 168]
gi|279705|pir||IQBSOC   replication initiation protein dnaA - Bacillus subtilis
gi|467391|dbj|BAA05237.1|   initiation protein of replicaton [Bacillus subtilis]
gi|118704|sp|P05648|DNAA_BACSU   Chromosomal replication initiator protein dnaA
gi|40014|emb|CAA26217.1|   pot. ORF 446 (aa 1-446) [Bacillus subtilis]
          Length = 446

 Score =  882 bits (2279), Expect = 0.0
 Identities = 446/446 (100%), Positives = 446/446 (100%)

Query: 1   MENILDLWNQALAQIEKKLSKPSFETWMKSTKAHSLQGDTLTITAPNEFARDWLESRYLH 60
           MENILDLWNQALAQIEKKLSKPSFETWMKSTKAHSLQGDTLTITAPNEFARDWLESRYLH
Sbjct: 1   MENILDLWNQALAQIEKKLSKPSFETWMKSTKAHSLQGDTLTITAPNEFARDWLESRYLH 60

Query: 61  LIADTIYELTGEELSIKFVIPQNQDVEDFMPKPQVKKAVKEDTSDFPQNMLNPKYTFDTF 120
           LIADTIYELTGEELSIKFVIPQNQDVEDFMPKPQVKKAVKEDTSDFPQNMLNPKYTFDTF
Sbjct: 61  LIADTIYELTGEELSIKFVIPQNQDVEDFMPKPQVKKAVKEDTSDFPQNMLNPKYTFDTF 120

Query: 121 VIGSGNRFAHAASLAVAEAPAKAYNPLFIYGGVGLGKTHLMHAIGHYVIDHNPSAKVVYL 180
           VIGSGNRFAHAASLAVAEAPAKAYNPLFIYGGVGLGKTHLMHAIGHYVIDHNPSAKVVYL
Sbjct: 121 VIGSGNRFAHAASLAVAEAPAKAYNPLFIYGGVGLGKTHLMHAIGHYVIDHNPSAKVVYL 180

Query: 181 SSEKFTNEFINSIRDNKAVDFRNRYRNVDVLLIDDIQFLAGKEQTQEEFFHTFNTLHEES 240
           SSEKFTNEFINSIRDNKAVDFRNRYRNVDVLLIDDIQFLAGKEQTQEEFFHTFNTLHEES
Sbjct: 181 SSEKFTNEFINSIRDNKAVDFRNRYRNVDVLLIDDIQFLAGKEQTQEEFFHTFNTLHEES 240

Query: 241 KQIVISSDRPPKEIPTLEDRLRSRFEWGLITDITPPDLETRIAILRKKAKAEGLDIPNEV 300
           KQIVISSDRPPKEIPTLEDRLRSRFEWGLITDITPPDLETRIAILRKKAKAEGLDIPNEV
Sbjct: 241 KQIVISSDRPPKEIPTLEDRLRSRFEWGLITDITPPDLETRIAILRKKAKAEGLDIPNEV 300

Query: 301 MLYIANQIDSNIRELEGALIRVVAYSSLINKDINADLAAEALKDIIPSSKPKVITIKEIQ 360
           MLYIANQIDSNIRELEGALIRVVAYSSLINKDINADLAAEALKDIIPSSKPKVITIKEIQ
Sbjct: 301 MLYIANQIDSNIRELEGALIRVVAYSSLINKDINADLAAEALKDIIPSSKPKVITIKEIQ 360
```

**Figure 2.6** Le résultat détaillé de BLAST correspondant au premier alignement de la figure 2.5

### 2.2.3 La base de données COG

Lorsque les signaux comparés sont des protéines, il est possible d'utiliser les outils décrits à la section précédente. Toutefois, c'est l'outil COG qui est le plus souvent utilisé puisqu'il est spécifique pour les protéines.

Les COG (« Clusters of Orthologous Groups of proteins »), sont des regroupements en familles de séquences de protéines prédites orthologues. Les COG ont été conçus dans le but de classifier les protéines des génomes complets, selon le concept de l'orthologie (Section 1.3). Tous ces regroupements et cette classification sont entreposés dans une

base de données sur Internet (http ://www.ncbi.nlm.nih.gov/COG) (Tatusov R. *et al.*, 2000).

Les COG sont des regroupements de séquences de protéines basé sur leur similarité. L'idée générale des COG repose sur le fait que les séquences d'acides aminés des protéines d'une même famille fonctionnelle, chez les procaryotes notamment, sont généralement très conservées.

L'application la plus directe des COG est la prédiction de fonctions des protéines ou des groupes de protéines, incluant celles qui proviennent de nouveaux génomes récemment séquencés. En effet, lorsqu'une ou plusieurs protéines de fonction inconnue ont des séquences ayant une forte similarité (mesurées avec BLAST par exemple) à celles d'un groupe de protéines de fonctions sont connues, il est raisonnable de supposer que la fonction de ces nouvelles protéines soit proche de celles du groupe.

La construction initiale des COG, à partir d'une base de séquences de protéines, a suivi le processus ci-dessous (Tatusov R. *et al.*, 2000), que nous détaillons car il illustre bien les concepts introduits dans ce chapitre :

1. On considère un ensemble $P$ de séquences de protéines.
2. Toutes les séquences d'acides aminés de l'ensemble $P$ sont comparées entre elles. On obtient alors un graphe aux arêtes valuées selon les résultats de BLAST.
3. Les protéines *paralogues* sont détectées et éliminées : un groupe de protéines est considéré comme paralogue s'il forme une clique telle que tous les sommets proviennent du même génome et, pour chaque sommet, toute arête incidente à ce sommet et n'appartenant pas à cette clique a un poids inférieur à toutes les arêtes incidentes à ce sommet appartenant à cette clique (Figure 2.7).
4. Les protéines *orthologues* sont détectées : un groupe de protéines est considéré comme orthologue s'il forme une clique telle que tous les sommets proviennent de génomes différents et pour chaque sommet, toute arête incidente à ce sommet et n'appartenant pas à cette clique a un poids inférieur à toutes les arêtes incidentes à ce sommet appartenant à cette clique (Figure 2.7).

5. Un ensemble de protéines orthologues forme une famille COG et reçoit un numéro.
6. Cette première classification est raffinée à l'aide d'outils basés sur l'analyse d'arbres phylogénétiques, l'analyse de «  clusters » et l'inspection visuelle des alignements BLAST.

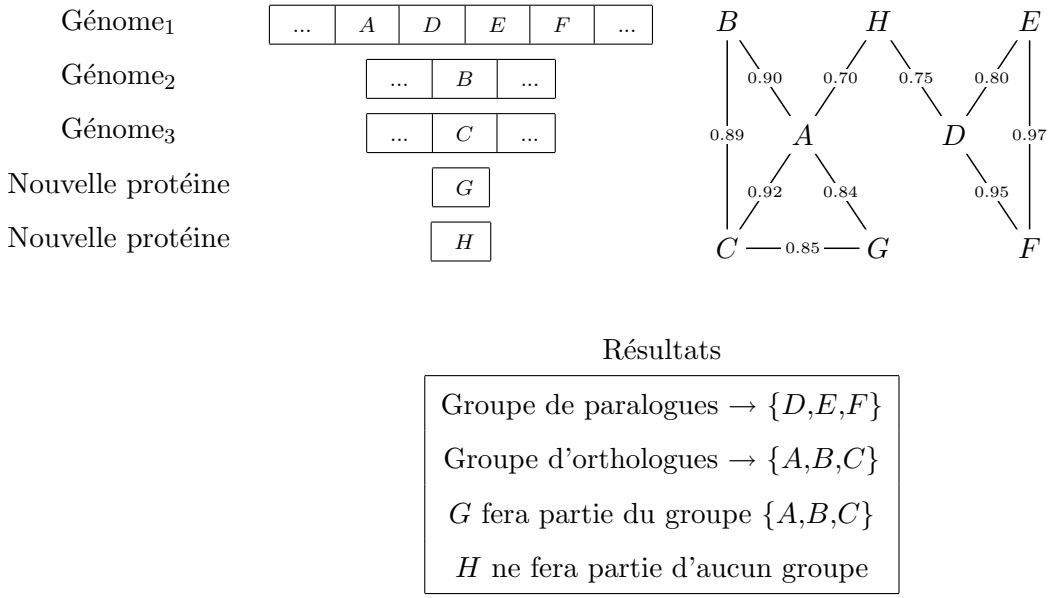



**Figure 2.7** Exemple fictif de la construction des COG

Il est également possible d'ajouter de nouvelles séquences de protéines aux COG déjà formés, en les comparant, avec BLAST, à toute la base de données COG. Si pour une protéine, les deux meilleurs résultats proviennent d'un même COG, la protéine en question sera ajoutée à ce COG (Figure 2.7).

Selon ces méthodes, un COG regroupe donc minimalement trois protéines de trois génomes distincts, étant donné que les plus petites cliques considérées sont celles à trois sommets.

Il faut cependant noter que plus la base de données de COG grandit, moins elle devient fiable quant à la classification des nouvelles protéines. En effet, plus il y a de protéines

et de génomes dans la base de données, plus il y a de chances que les deux meilleurs résultats proviennent du même COG, car les séquences correspondantes sont identiques ou presque. Il est donc possible de varier certains paramètres, pour obtenir une efficacité satisfaisante.

Selon (Tatusov R. *et al.*, 2000), 60% des COG ont une seule protéine, ce qui veut dire que ces protéines ne font pas partie de groupes créés par COG et n'ont aucune protéine paralogue et orthologue connue.

**Remarque 2.2.1** Il est important de noter que les COG ne forment pas une partition de l'ensemble des séquences de protéines : une séquence peut se retrouver dans plusieurs COG. Cet aspect est problématique car il devient impossible d'assigner sans ambiguïté une étiquette (en terme de COG) à chaque gène d'un génome, car certains gènes peuvent recevoir deux étiquettes ou plus.

### 2.2.4 La base de données de domaines PFAM

Encore une fois, lorsque les signaux sont des protéines, l'outil spécialisé PFAM (Protein FAMilies database of alignments and HMMs) est souvent utilisé. C'est un autre type de base de données de séquences basée sur la similarité.

PFAM est une base de données de familles de domaines de protéines, qui a été développée pour repérer l'homologie distante (http ://www.sanger.ac.uk/Software/Pfam) (Bateman A. *et al.*, 2000). Un des buts principaux de PFAM est la classification automatique et rapide des protéines. Cette classification peut donc être utilisée pour l'annotation de protéines.

Les recherches de similarités de séquences, comme celles produites par BLAST, ne donnent pas toujours d'indication précise concernant les structures de domaines des protéines, PFAM doit donc utiliser un outil spécialisé : les profils HMM. En effet, BLAST vise les recherches de similarité entre deux séquences, tandis que les HMM servent à faire des recherches de similarités entre une séquence et une famille de séquences.

Un *Profil HMM* est un modèle de Markov caché (Baldi, P - Brunak S., 2001) représentant un ensemble de séquences (une famille). Ce modèle probabiliste est construit par apprentissage automatique à partir de la famille de séquences. Étant donné un HMM pour une famille $F$ et une séquence $s$, on peut calculer alors un « score » probabiliste exprimant la vraisemblance que $s$ soit un membre de la famille $F$.

La base de données PFAM contient un ensemble de HMM, chacun modélisant une famille de séquences de domaines de protéines. Au contraire de COG, PFAM forme une partition des séquences de domaines.

### 2.2.5 Conclusion

Il existe donc plusieurs façons d'étiqueter les signaux. Les outils et bases de données BLAST, COG et PFAM par exemple, se basent sur la similarité des séquences, mais l'étiquetage pourrait très bien se baser sur un autre aspect, par exemple l'annotation : deux gènes ont la même étiquette si leur annotation est identique.

## 2.3 Équipes de gènes

Nous introduisons maintenant la notion d'équipe de gènes, à travers les travaux sur celle-ci, entre autres de Bergeron *et al.* (Bergeron A. *et al.*, 2002), qui ont motivé notre travail.

Intuitivement, étant donné un ensemble de génomes, représentés par des séquences de gènes, chaque gène étant étiqueté par la famille à laquelle il appartient, on cherche à exhiber des groupes de segments de gènes tels que les segments d'un même groupe soient similaires en terme de leur contenu en gènes.

### 2.3.1 Définitions et problématique

Soit $\Sigma$ l'ensemble des étiquettes associées aux gènes présents dans les génomes que nous considérons. Par abus de langage, nous appelons $\Sigma$ « l'alphabet des gènes ».

Soit $E$ un ensemble non vide de gènes, tel que $E \subseteq \Sigma$.

Une $\delta$-*occurrence* de $E$ est un segment $S$ d'un génome tel que :

1) le premier et le dernier gène de $S$ sont dans $E$ ;
2) chaque sous-segment de $S$ de longueur $\delta$ contient au moins un gène de $E$ ;
3) $S$ n'est pas inclus dans un segment plus long qui possède les propriétés 1) et 2).

Dans une $\delta$-*occurrence* de $E$, tous les sous-segments formés uniquement de gènes n'appartenant pas à $E$ sont donc de longueur inférieure ou égale à $\delta$. Par exemple, si nous avons $\delta = 2$, $E = \{\text{A, B}\}$ et la séquence de gènes :

A B C D B C A B A D C,

nous obtenons les deux $\delta$-occurrences de $E$ suivantes :

A B

et

B C A B A.

On dit que $F$, où $F \subseteq E$, est une *extension* de $E$ si toute $\delta$-occurrence de $E$ est contenue dans une $\delta$-occurrence de $F$. Par exemple, si nous avons $\delta = 2$, $E = \{\text{B, C}\}$,

$G_1 =$ A B C D E H I B E C D A,
$G_2 =$ F G A B C E F B D C G A,

alors nous obtenons les $\delta$-occurrences suivantes :

$G_1 =$ A <u>B C</u> D E H I <u>B E C</u> D A,
$G_2 =$ F G A <u>B C</u> E F <u>B D C</u> G A.

Si nous avons $F = \{\text{A, B, C}\}$, alors $F$ est une extension de $E$ car toutes les $\delta$-occurrences de $E$ sont incluses dans celles de $F$ :

$$G_1 = \underline{\text{A B C}}\ \text{D E H I}\ \underline{\underline{\text{B E C}}\ \text{D A}},$$
$$G_2 = \text{F G}\ \underline{\text{A B C}}\ \text{E F}\ \underline{\underline{\text{B D C}}\ \text{G A}}.$$

Tandis que si nous avons $G = \{\text{B, C, D}\}$, nous n'aurons pas d'extension de $E$, car il existe au moins une $\delta$-occurrence de $E$ qui n'est pas incluse dans l'une de celles de $G$ :

$$G_1 = \text{A}\ \underline{\underline{\text{B C D}}}\ \text{E H I}\ \underline{\underline{\text{B E C D}}}\ \text{A},$$
$$G_2 = \text{F G A}\ \underline{\text{B C}}\ \text{E F}\ \underline{\underline{\text{B D C}}}\ \text{G A}.$$

Une *équipe* de gènes est un ensemble de gènes ayant au moins une $\delta$-occurrence et qui n'a pas d'extension.

### 2.3.2 Calculs des équipes de gènes

Le modèle d'équipes de gènes a été introduit par Bergeron *et al.* (Bergeron A. *et al.*, 2002) et a donné lieu à plusieurs travaux que nous survolons maintenant.

**Équipes de gènes sans duplication.**

La notion d'équipe de gènes a été initialement conçue pour comparer des génomes sans gènes dupliqués : un gène peut appartenir à plusieurs génomes, mais il ne peut pas apparaître plus d'une fois dans un même génome. Par exemple,

C F G E D A H B

est un génome valide, tandis que

A B F G B C G D E

ne l'est pas car B et G apparaissent deux fois dans le génome.

**Exemple.** Les équipes de gènes présentes dans

$$G_1 = \text{C F G E D A H B},$$
$$G_2 = \text{A B I J K C L D E}$$

avec $\delta = 3$, sont :

- {A, B} dont les $\delta$-occurrences sont A H B et A B ;
- {C, D, E} dont les $\delta$-occurrences sont C F G E D et C L D E.

Bergeron *et al.* (Bergeron A. *et al.*, 2002) proposent deux algorithmes calculant les équipes de gènes et leurs occurrences pour $k$ génomes : le premier, basé sur une approche simple, a une complexité temporelle de $\mathcal{O}(k^2n^2)$, où $n$ est le nombre de gènes total et une complexité spatiale linéaire ; le second algorithme, plus élaboré, a une complexité temporelle de $\mathcal{O}(kn \log^2 n)$ et une complexité spatiale linéaire également.

La figure 2.8 illustre un résultat obtenu par ces algorithmes lors de la comparaison de génomes bactériens (Bergeron A. *et al.*, 2003). Il s'agit de l'opéron *Tryptophane* chez trois bactéries : *A. fulgidus* (arcfu), *M. thermoautotrophicum* (metth) et *P. abyssi* (pyrab). Les symboles [R] (« reverse ») et [F] (« forward ») indiquent l'orientation des gènes. Les trois lignes du haut sont l'équipe produite par le programme où les nombres sont les étiquettes attribuées aux gènes orthologues. Les trois lignes du bas sont la même équipe, où les étiquettes numériques ont été changées par des lettres afin d'identifier l'opéron *Tryptophane*. La ligne de *M. thermoautotrophicum* (metth) a été inversée pour mieux visualiser l'alignement des trois segments.

$$325 \rightarrow A\ (trpA)$$
$$299 \rightarrow B\ (trpB)$$
$$362,\ 364 \rightarrow G\ (trpG)$$
$$226 \rightarrow E\ (trpE)$$
$$198 \rightarrow D\ (trpD)$$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **arcfu** | **325** | **299** | * | **364** | **226** | **198** | |
| metth | **226** | **364** | * | * | **299** | **325** | **198** |
| pyrab | **325** | **299** | * | **362** | **226** | **198** | |

Original output for the Trp operon, where the genes are labeled by the identifier of the orthologous triplet they belong to.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| arcfu | - | A | B | *F* | - | G | E | C-D | | [R] |
| metth | D | A | B | *F* | *C* | G | E | - | | [F] inverted |
| pyrab | - | A | B | *F* | - | G | E | D | (C) | [R] |

**Figure 2.8** L'opéron tryptophane (Bergeron A. *et al.*, 2003)

Mentionnons que l'assignation des étiquettes s'est fait avec les meilleurs résultats bi-directionnels de BLAST. C'est-à-dire que le meilleur résultat fourni par BLAST, de la comparaison du gène $s_1$ avec tous les gènes de toutes les espèces, est $s_2$ et vice-versa.

### Équipes de gènes entre deux génomes.

Récemment, He et Goldwasser (He X. - Goldwasser M., 2004) ont étendu les algorithmes de Bergeron *et al.* pour prendre en compte de la présence de plusieurs copies d'un même gène dans un génome. Cependant, leur méthode est limitée à la comparaison de deux génomes, ce qui leur permet de proposer un algorithme en temps $\mathcal{O}(mn)$ et en espace $\mathcal{O}(m+n)$, où $m$ et $n$ sont le nombre de gènes dans les deux génomes.

Ils ont appliqué leur méthode à *E. coli* et *B. subtilis*. La table 2.2 illustre leurs résultats. Il y a soixante-cinq équipes de gènes dont chacune contient deux gènes, seize équipes sont composées de trois gènes chacune, trois équipes sont composées de quatre gènes

chacune et une équipe de gènes contient vingt-et-un gènes.

| Nb. de gènes dans une équipe | Nb. d'équipes de gènes |
|---|---|
| 2 | 65 |
| 3 | 16 |
| 4 | 3 |
| 21 | 1 |
| Total | 85 |

**Table 2.2** Les résultats de He et Goldwasser (He X. - Goldwasser M., 2004)

**Remarque 2.3.1** Si les duplications sont considérées et qu'il y a plus de deux génomes traités, alors la taille de la sortie peut être exponentielle (Pasek S. *et al.*, 2005).

Voici un exemple qui donne une idée de l'ordre de grandeur que peut avoir le nombre d'équipes générées. Soit $\delta = 3$, $\Sigma = \{A, B, C, D, E\}$ et les cinq segments suivants :

$$A\ B\ C\ D$$
$$A\ B\ C\ E$$
$$A\ B\ D\ E$$
$$A\ C\ D\ E$$
$$B\ C\ D\ E$$

les dix équipes sont tous les sous-ensembles de deux et trois éléments de $\Sigma$. Pour plus de détails, voir (Pasek S. *et al.*, 2005).

**Équipes de gènes générales.**

Finalement, Pasek *et al.* (Pasek S. *et al.*, 2005) ont proposé un algorithme, exponentiel dans le pire des cas qui, étant donné $k$ génomes calcule toutes les équipes avec leurs occurrences. Le caractère exponentiel de cet algorithme vient du fait que le nombre d'équipes peut être exponentiel en le nombre d'étiquettes total.

Ils ont appliqué leur méthode, entre autre, sur un jeu de données de quatre bactéries : *Y. pestis*, *S. typhi*, *V. cholerae* et *E. coli*. La figure 2.9 illustre un exemple d'équipe de gènes obtenue. Chaque gène est composé d'un ou de plusieurs domaines PFAM, par exemple le gène *fruB* chez *T. pestis* contient les domaines étiquetés 359 et 381. Les équipes de gènes sont donc construites à partir des étiquettes des domaines PFAM.

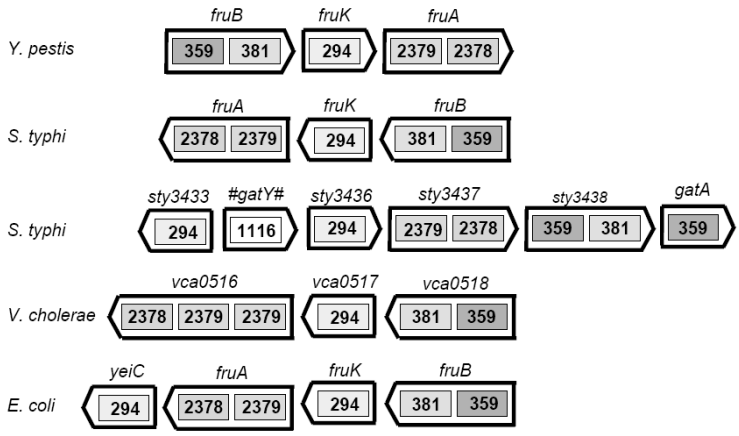



**Figure 2.9** Exemple de résultats de Pasek *et al.* (Pasek S. *et al.*, 2005)

**Conclusion.**

À la lumière de ces modèles et de leurs restrictions, nous avons voulu créer un modèle de même nature que les équipes de gènes, mais un peu plus grossier, de manière à réduire la complexité des calculs sans toutefois ajouter de restrictions quant aux duplications et au nombre de génomes pouvant être considérés.

# Chapitre III

# UNE APPROCHE PRAGMATIQUE

## 3.1 Introduction

De manière similaire aux modèles présentés à la section 2.3, notre approche consiste à calculer des groupes de segments similaires en termes de contenu en gènes. Nous prenons en entrée des génomes, qui sont des suites de gènes étiquetés, et nous produisons des groupes de segments des génomes. Pour ces raisons de lisibilité, nous avons utilisé des lettres pour étiqueter les gènes de nos exemples précédents. Dans notre algorithme, nous utiliserons des chiffres pour étiqueter les gènes.

## 3.2 Description de haut niveau

Notre algorithme prend en entrée $k$ tableaux d'entiers $G_1$, ..., $G_k$ de tailles respectives $n_1$, ..., $n_k$, représentant $k$ génomes. Pour des raisons de simplicité, notre algorithme traite la concaténation de ces k tableaux. Les entiers dans ces tableaux sont appelés des gènes et sont éléments de l'ensemble $\{1, 2, ..., M\}$. La taille totale des tableaux est $N = \sum_{i=1}^{k} n_i$. L'entrée comprend aussi deux entiers ($\delta$ et $\omega$) et un paramètre réel ($\gamma \in [0, 1]$), dont les rôles seront décrits plus loin.

La sortie produite par notre algorithme est une liste de groupes de segments des génomes fournis en entrée.

L'idée générale derrière notre façon de calculer des segments de génomes est de s'ap-

puyer sur une conservation très locale du contenu en gènes, qui rend les calculs peu coûteux. La motivation biologique de cette idée repose sur l'observation que la majorité des réarrangements, notamment chez les procaryotes, sont des réarrangements courts (Sankoff D., 2002; Sankoff D., 2003; Sankoff D. *et al.*, 2005).

L'algorithme se décompose en deux grandes étapes indépendantes :

- la segmentation des génomes $G_1$, ..., $G_k$ en listes de segments non chevauchants, basé sur la conservation du contenu en gènes de ces segments ;
- le calcul de groupes de segments, où chaque groupe sera composé de segments dont le contenu en gènes est conservé.

## 3.3 Segmentation

### 3.3.1 Principe de la segmentation

Un segment $g_i$ ... $g_j$ d'un génome, c'est-à-dire les entiers compris entre les positions $i$ et $j$ inclusivement, du génome $G$, est un *$\delta$-segment* si $j$ - $i \leq \delta$, où $i \leq j$. Rappelons que $\delta$ est un paramètre d'entrée.

**Définition 3.3.1** Un segment $g_i$ ... $g_j$ est un *$\delta$-segment conservé* s'il vérifie une des deux propriétés suivantes :

- $g_i = g_j$ : on dit que $g_i$ ... $g_j$ est un $\delta$-segment conservé *tandem*, ou
- il existe au moins un autre $\delta$-segment, dans le même génome ou dans un autre génome, ne chevauchant pas $g_i$ ... $g_j$, et dont les extrémités sont étiquetées par $g_i$ et $g_j$, peu importe l'ordre : on dit alors que $g_i$ ... $g_j$ est un $\delta$-segment conservé *dupliqué*.

**Exemple 3.3.1** Supposons que $\delta = 2$ et que deux segments de génome(s) sont :

| $A$ | $B$ | $A$ | $C$ | $D$ | $E$ | ... |
|---|---|---|---|---|---|---|

| ... | $A$ | $E$ | $F$ | $D$ | ... |
|---|---|---|---|---|---|

nous y retrouvons un $\delta$-segment conservé tandem,

- | $A$ | $B$ | $A$ |

car les gènes aux extrémités ont la même étiquette, et deux $\delta$-segments conservés dupliqués,

- | $D$ | $E$ |
- | $E$ | $F$ | $D$ |

car les gènes aux extrémités de chaque $\delta$-segment ont la même paire d'étiquettes.

À la fin de cette étape de segmentation, nous obtenons, de manière imagée, un ensemble de $\delta$-segments conservés des génomes (représentés par des traits, possiblement chevauchants, tel qu'illustré à la figure 3.1.

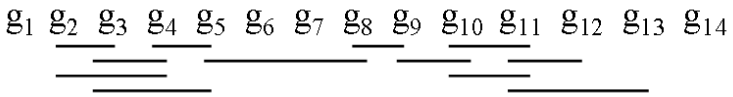



**Figure 3.1** La segmentation

Dans une seconde étape, nous assignons un poids à chaque région intergénique, c'est-à-dire aux espaces situés entre deux gènes consécutifs $g_i$ et $g_{i+1}$. Ce poids est le nombre de $\delta$-segments conservés calculés lors de l'étape de segmentation et contenant à la fois $g_i$ et $g_{i+1}$, tel qu'illustré à la figure 3.2.

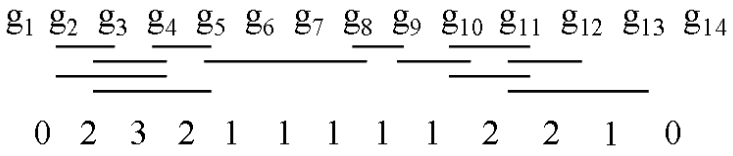



**Figure 3.2** La segmentation et les poids

Selon le paramètre d'entrée $\omega$, nous ne conservons que les régions intergéniques ayant un poids $\geq \omega$. Nous obtenons un ensemble de segments non chevauchants tels que chaque région intergénique d'un segment a un poids $\geq \omega$.

Par exemple, pour la segmentation de la figure 3.2, si $\omega = 2$, nous obtenons les deux segments suivants :

- $g_2\ g_3\ g_4\ g_5$
- $g_{10}\ g_{11}\ g_{12}$

La liste de segments ainsi obtenue servira d'entrée à la seconde phase de notre algorithme, qui calculera des groupes de segments similaires.

### 3.3.2 Détails algorithmiques

**Structure de données pour le poids des adjacences d'un génome $G_i$**

Étant donné qu'un génome est une suite linéaire de $n$ étiquettes, les régions intergéniques peuvent être représentées par un tableau linéaire de $n$ - 1 éléments. Nous utilisons donc un tableau $R$ de $n$ - 1 entiers, noté $R_i$ lorsqu'il est associé au génome $G_i$. La case $R_i[j]$ est le poids de la région intergénique entre $G_i[j]$ et $G_i[j+1]$. Les cases du tableau initialisé contiennent un 0.

**Exemple 3.3.2** Prenons le génome suivant et $\delta = 2$ :

$$A\ B\ C\ D\ F\ G\ B\ E\ A\ C.$$

Il comporte les quatre $\delta$-segments conservés dupliqués,

$$A\ B,\ B\ E\ A$$
et
$$A\ B\ C,\ A\ C$$

et le tableau pour les poids serait celui de la figure 3.3.

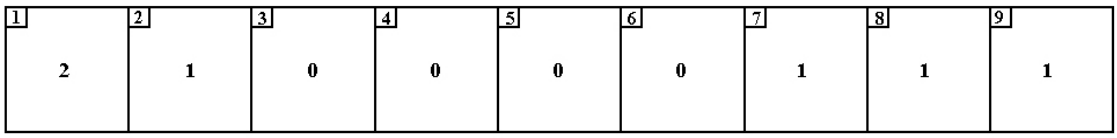



**Figure 3.3** Le tableau utilisé pour les poids

**Structure de données utilisée pour la détection des $\delta$-segments conservés**

Lors de la lecture des génomes, nous utilisons un tableau $T$ de $M$ enregistrements $T[1]$, ..., $T[M]$, où $M$ est la cardinalité de l'ensemble des étiquettes attribuées aux gènes des génomes. Chaque enregistrement $T[g]$ est un arbre binaire équilibré où les sommets contiennent les informations relatives au premier segment rencontré d'extrémités étiquetées $g$ et $h$, nous supposons $g < h$. Les coordonnées de ce premier segment sont données par $l$ le numéro de génome selon l'ordre de la liste d'entrée, $d$ et $f$ les positions de début et de fin dans le génome numéro $l$. La dernière information relative à ce premier segment rencontré est un booléen *occ* de valeur 1 si ce segment n'a pas été rencontré ailleurs et 0 s'il a été rencontré ailleurs au moins une fois.

**Exemple 3.3.3** Prenons le génome suivant :

$$A\ B\ C\ D\ F\ G\ B\ E\ A\ C.$$

Le tableau pour la détection des $\delta$-segments conservés après la lecture des sept premiers gènes serait celui de la figure 3.4 si $\delta = 2$.

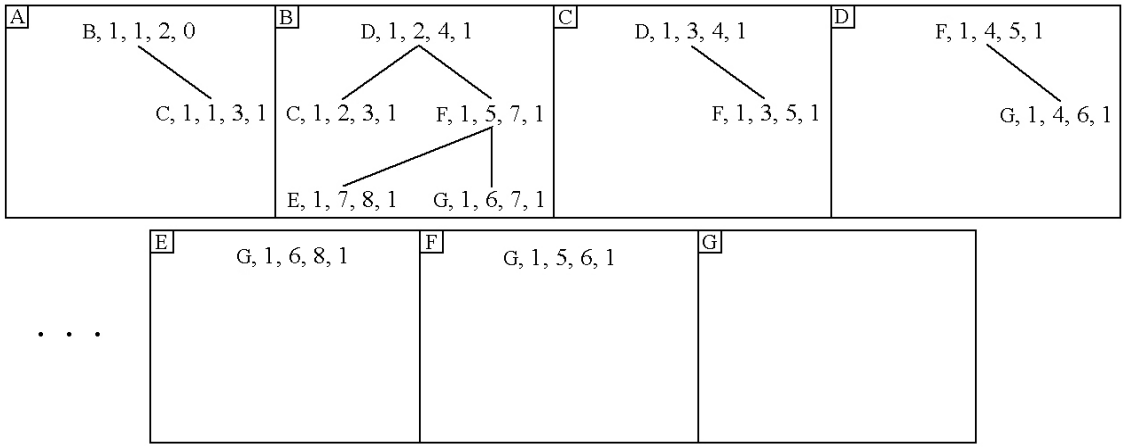



**Figure 3.4** Le tableau utilisé pour la détection des $\delta$-segments conservés des sept premiers gènes du génome $A\ B\ C\ D\ F\ G\ B\ E\ A\ C$

Nous sommes rendus à la lecture du huitième gène, $E$. Nous considérons chaque segment de gènes formés de ce huitième gène $E$ et des $\delta$ gènes suivants : $A$ et $C$. Pour le segment d'extrémités étiquetés $E$ et $A$, nous regardons dans la case $A$, car $A < E$, s'il y a déjà un noeud $E$. Dans ce cas-ci il n'y a pas de noeud $E$, nous ajoutons donc un noeud dans la case $A$ contenant les informations suivantes, tel qu'illustré par la figure 3.5 :

- l'étiquette de l'extrémité : $E$
- $l$ le numéro du génome : 1
- $d$ la position de début du segment : 8
- $f$ la position de fin du segment : 9
- *occ* le segment est rencontré seulement une fois : 1

Ensuite, pour le segment d'extrémités étiquetés $E$ et $C$, nous regardons dans la case $C$ s'il y a déjà un noeud $E$. Il n'y a pas de noeud $E$, alors nous ajoutons le noeud ($E$, 1, 8, 10, 1) dans la case $C$, tel qu'illustré par la figure 3.6.

Finalement, lors de la lecture du neuvième gène $A$, nous considérons l'unique segment d'extrémités étiquetés $A$ et $C$. Dans la case $A$, il y a un noeud $C$, nous affectons la

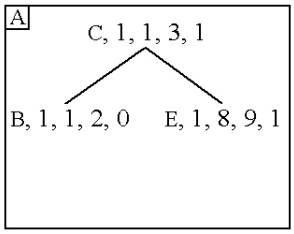



**Figure 3.5** La case $A$ après l'ajout du noeud $E$

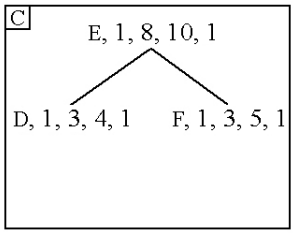



**Figure 3.6** La case $C$ après l'ajout du noeud $E$

valeur 0 au booléen *occ* du noeud $C$, tel qu'illustré par la figure 3.7 montrant le tableau $T$ après la lecture de tout le génome.

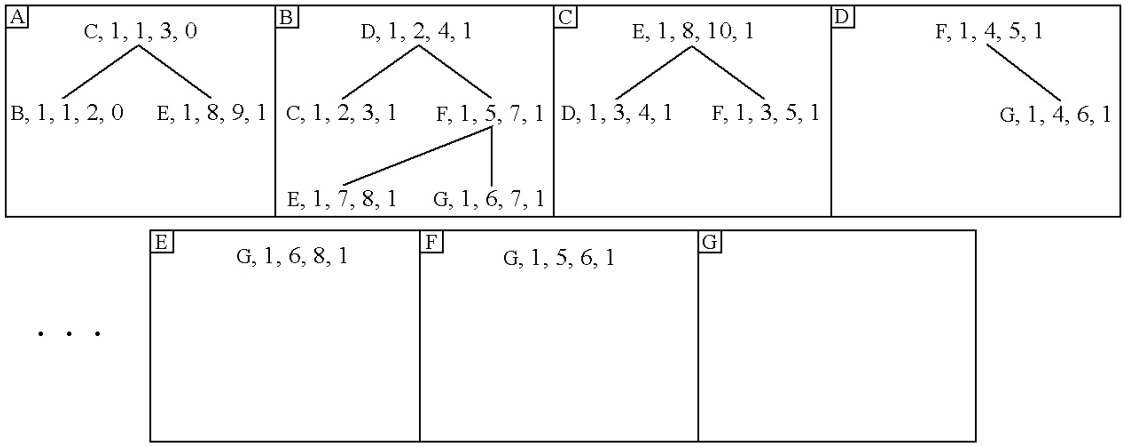



**Figure 3.7** Le tableau utilisé pour la détection des $\delta$-segments conservés de tout le génome $A\ B\ C\ D\ F\ G\ B\ E\ A\ C$

**Algorithme de calcul des poids**

Pour $x$ bouclant sur les $k$ génomes faire

Pour $i$ bouclant sur les $n_x$ - 1 premiers gènes du génome $x$ faire

Pour $j$ bouclant sur les gènes des positions $i + 1$ à $min(n_x, i + \delta)$ du génome $x$ faire

$g \leftarrow min(G_x[i], G_x[j])$

$h \leftarrow max(G_x[i], G_x[j])$

Si $T[g]$ contient un segment dont les extrémités sont étiquetées $g$ et $h$ alors

Mettre à jour $R_x[i + 1]$ à $R_x[j - 1]$

Si *occ* de $T[g] = 1$ alors

*occ* de $T[g] \leftarrow 0$

Mettre à jour $R_l[d + 1]$ à $R_l[f - 1]$

Fin Si

Sinon

$l \leftarrow x$

$d \leftarrow i$

$f \leftarrow j$

$occ \leftarrow 1$

Insérer ce segment dans $T[g]$

Fin Si

Fin Pour

Fin Pour

Fin Pour

Le tableau $T$ pour la détection des $\delta$-segments conservés occupe un espace en $\mathcal{O}(M^2)$, où $M$ est le nombre total d'étiquettes possibles.

Le calcul de la complexité en temps de l'algorithme de calcul des poids, pour chacune des étapes est :

- Les deux boucles externes parcourent une et une seule fois tous les gènes de tous les génomes, sauf le dernier de chaque génome. Donc leurs instructions internes

seront exécutées $N - k\delta$ fois.

- La boucle interne parcourt une fenêtre de $\delta$ pour chaque gène rencontré lors des deux boucles précédentes. Les instructions à l'intérieur des trois boucles imbriquées seront exécutées $\delta(N - k\delta)$ fois.
- Le test qui vérifie si une case du tableau $T$ contient un segment aux extrémités étiquetés $g$ et $h$ effectue une recherche dans un arbre binaire équilibré. La complexité de cette recherche est $\mathcal{O}(log(N))$. Si la case du tableau $T$ contient le segment aux extrémités étiquetés $g$ et $h$, alors une mise à jour des poids d'au plus $\delta$ - 1 cases du tableau $R_x$ s'effectue en $\delta - 1$ instructions.

Nous obtenons donc une complexité en $\mathcal{O}(\delta N max(\delta, log(N)))$. Comme $\delta$ est en général petit, entre 2 et 4 dans nos applications, nous obtenons une complexité en $\mathcal{O}(Nlog(N))$.

## 3.4 Calcul des groupes de segments

Le tableau $R_x$ et l'entier $\omega$ définissent implicitement la liste des segments dont nous avons besoin. Rappelons qu'un segment utilisé pour le calcul des groupes est un segment dont chacune des régions intergéniques a un poids $\geq \omega$.

### 3.4.1 Principe du calcul des groupes de segments

Le principe du calcul des groupes de segments est de créer un graphe $G$ dont les sommets sont les segments non chevauchants définis dans la phase de segmentation. Nous définissons un groupe de segments comme étant une composante connexe de ce graphe.

Le problème consiste à définir les arêtes du graphe $G$. Pour ce faire, chaque segment est alors considéré comme étant le multi-ensemble des étiquettes des gènes qui le compose. Nous calculons ensuite, pour chaque paire de segments $(S_i, S_j)$, le rapport suivant :

$$p_{ij} = \frac{nombre\ d'étiquettes\ de\ S_i\ appartenant\ à\ S_j}{nombre\ de\ gènes\ total\ de\ S_i}.$$

Nous obtenons une matrice de rapports $P$ qui servira de guide à la construction du graphe $G$ dont les sommets sont des segments. Une arête entre deux segments $S_i$ et

$S_j$ est créé si les rapports correspondants $p_{ij}$ et $p_{ji}$ sont $\geq \gamma$. Rappelons que $\gamma$ est un paramètre d'entrée.

**Exemple 3.4.1** Soient les six segments suivants :

| | | | | | | |
|---|---|---|---|---|---|---|
| $S_1$ | ... | $A$ | $B$ | $A$ | ... | |
| $S_2$ | ... | $E$ | $G$ | $B$ | ... | |
| $S_3$ | ... | $B$ | $E$ | $D$ | $C$ | ... |
| $S_4$ | ... | $D$ | $F$ | $E$ | ... | |
| $S_5$ | ... | $A$ | $E$ | $F$ | $D$ | ... |
| $S_6$ | ... | $B$ | $G$ | $A$ | $F$ | $B$ | ... |

alors en nous aidant du tableau $P$ suivant :

| $\lvert\cap\rvert$ | $S_1$ | $S_2$ | $S_3$ | $S_4$ | $S_5$ | $S_6$ |
|---|---|---|---|---|---|---|
| $S_1$ | | 33% | 33% | 0% | 33% | 100% |
| $S_2$ | 33% | | 66% | 33% | 33% | 66% |
| $S_3$ | 25% | 50% | | 50% | 50% | 25% |
| $S_4$ | 0% | 33% | 66% | | 100% | 50% |
| $S_5$ | 25% | 0% | 50% | 75% | | 50% |
| $S_6$ | 60% | 60% | 40% | 20% | 40% | |

nous obtenons le graphe de gauche $G$ de la figure 3.8, si $\gamma = 50\%$ et celui de droite si $\gamma = 60\%$.

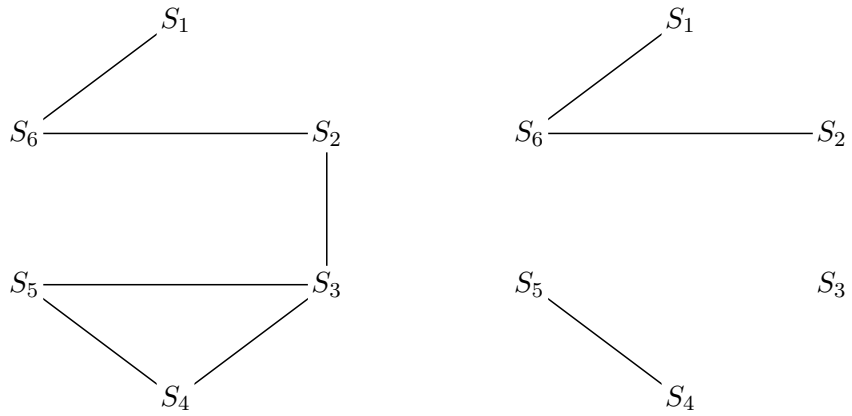



**Figure 3.8** Deux graphes obtenus pour les segments de l'exemple 3.4.1

Les listes des groupes (c'est-à-dire les composantes connexes du graphe $G$) sont les suivantes, pour $\gamma = 50\%$ :

- $\{S_1, S_2, S_3, S_4, S_5, S_6\}$

et, pour $\gamma = 60\%$ :

- $\{S_1, S_2, S_6\}$
- $\{S_4, S_5\}$
- $\{S_3\}$

### 3.4.2 Détails algorithmiques

À la fin de l'étape de segmentation, nous obtenons une liste de segments qui peut être longue. Il est important de mentionner que nous ne codons pas le graphe $G$ par une matrice d'adjacences, mais pour chaque sommet, nous calculons la liste de ses voisins (sommets incidents).

Pour construire ce graphe $G$, nous comparons chaque segment à tous les autres, afin d'obtenir le tableau $P$. La comparaison de deux segments se fait aisément en temps linéaire en la taille de ces deux segments et cette étape demande donc un temps et un espace en $\mathcal{O}(N^2)$, dans le pire des cas. Nous avons noté qu'en pratique nous utilisons un espace en $\mathcal{O}(N)$.

La dernière étape du calcul des groupes de segments consiste à récupérer les composantes connexes du graphe $G$ créé. Il s'agit d'effectuer un parcours en profondeur classique (Neapolitan R. - Naimipour K., 2005) en temps $\mathcal{O}(N^2)$.

# Chapitre IV

# LES RÉSULTATS

Dans ce chapitre, nous présentons les résultats obtenus par notre algorithme avec plusieurs jeux de données. Premièrement, nous avons comparé les génomes des bactéries de *Bacillus subtilis* et *Escherichia coli*, ce qui nous a permis de comparer les résultats de notre méthode avec ceux de He et Goldwasser (He X. - Goldwasser M., 2004). Nous avons ensuite utilisé notre algorithme pour analyser trois génomes, *Escherichia coli*, *Salmonella typhimurium* et *Haemophilus influenzae* à la recherche des opérons de *Escherichia coli*, et nous avons comparé nos résultats avec ceux de Chen *et al.* (Chen Y. *et al.*, 2004). Finalement, nous avons analysé un jeu de douze génomes de $\gamma$-Protéobactéries.

Avant de présenter ces trois expériences, nous détaillons à la section 4.1 la méthode que nous avons utilisée pour définir des familles de gènes homologues pour nos trois jeux de données.

## 4.1 Calcul de familles de gènes homologues

Rappelons que notre algorithme de calcul des groupes de gènes prend en entrée un ensemble de séquences signées, chacune représentant un génome. Cependant, les données que nous utilisons proviennent des banques de données génomiques, telles que GENBANK, SubtiList, Colibri et NCBI. Dans ces banques de données, un génome est représenté par un fichier contenant, entre autres informations, la séquence d'acides aminés de chacun des gènes de ce génome. Il nous faut donc, dans un premier temps,

transformer ces fichiers en suites signées.

Pour cela, nous utilisons la méthode décrite à la section 2.2. Dans un premier temps, nous extrayons des fichiers les séquences d'acides aminés de chacun des gènes présents. Ensuite, nous comparons deux à deux toutes ces séquences d'acides aminés avec le programme BLAST. À partir de ces comparaisons, nous définissons un graphe, dont les sommets sont les gènes et les arêtes sont définies comme suit :

**Définition 4.1.1** Soient deux gènes $g_1$ et $g_2$, alignées à l'aide du programme BLAST, tel qu'illustré par la figure 4.1.

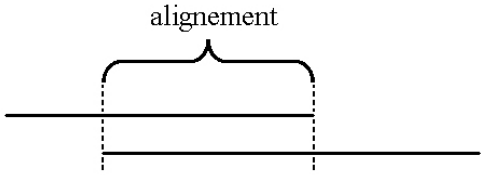



**Figure 4.1** Le schéma d'un alignement entre deux gènes

Une arête lie ces deux gènes si :

- le ratio d'identité de l'alignement défini par

$$\frac{nombre\ d'acides\ aminés\ identiques\ de\ l'alignement}{nombre\ d'acides\ aminés\ total\ de\ l'alignement}$$

  est $\geq 0.30$ (30%), fixé arbitrairement

- pour chacun des deux gènes, le ratio de chevauchement, défini par

$$\frac{longueur\ de\ l'alignement}{longueur\ de\ la\ séquence\ du\ gène}$$

  est $\geq 0.70$ (70%), fixé arbitrairement.

On définit alors une famille de gènes homologues comme un ensemble de gènes formant une composante connexe de ce graphe, et on assigne à chaque famille de gènes homologues un numéro unique.

Pour obtenir la suite signée représentant un génome, nous représentons, pour chaque chromosome, chaque gène par le numéro de la famille à laquelle il appartient, le signe étant donné par l'orientation de ce gène.

Si les fichiers de génomes proviennent de la banque de données GENBANK, cette phase de génération de suites de séquences signées est faite automatiquement à partir d'un ensemble de scripts fournis en annexe.

**Exemple 4.1.1** Soient les deux génomes suivants, $S_1$ composé d'un chromosome de trois gènes et $S_2$ composé d'un chromosome de deux gènes :

$$S_1 : \quad g_1\ g_2\ g_3$$
$$S_2 : \quad g_4\ g_5.$$

Supposons que les séquences d'acides aminés des gènes de ces deux génomes sont les suivantes :

$$g_1 : \quad \text{AGLPVIENLQ}$$
$$g_2 : \quad \text{YLAGNPVIANLQ}$$
$$g_3 : \quad \text{NWQCLGW}$$
$$g_4 : \quad \text{DLPHIENWQVL}$$
$$g_5 : \quad \text{LRMCFDTKN}$$

Nous construisons un graphe $G$ dont les sommets sont les cinq gènes et les arêtes définissent des familles de gènes homologues.

Considérons l'alignement suivant des gènes $g_2$ et $g_5$

$$\begin{array}{lcccccccccccc} g_2 : & \text{Y} & \text{L} & \text{A} & \text{G} & \text{N} & \text{P} & \text{V} & \text{I} & \text{A} & \text{N} & \text{L} & \text{Q} \\ & & | & & & & & & & & | & & \\ g_5 : & & \text{L} & \text{R} & \text{M} & \text{C} & \text{F} & \text{D} & \text{T} & \text{K} & \text{N} & & \end{array}$$

La condition de chevauchement est respectée car nous obtenons un ratio de chevauchement de $\frac{9}{12}$ pour le gène $g_2$ et $\frac{9}{9}$ pour le gène $g_5$, qui sont $\geq 0.70$ (70%). Toutefois, la condition d'identité n'est pas respectée car nous obtenons un ratio d'identité de $\frac{2}{9}$ qui est $< 0.30$ (30%). Ces deux gènes, $g_2$ et $g_5$, ne sont pas reliés par une arête dans le graphe $G$.

Considérons maintenant l'alignement suivant des gènes $g_3$ et $g_4$

$$
\begin{array}{lcccccccccccc}
g_3: & & & & & & & \mathrm{N} & \mathrm{W} & \mathrm{Q} & \mathrm{C} & \mathrm{L} & \mathrm{G} & \mathrm{W} \\
 & & & & & & & | & | & | & & | & & \\
g_4: & \mathrm{D} & \mathrm{L} & \mathrm{P} & \mathrm{H} & \mathrm{I} & \mathrm{E} & \mathrm{N} & \mathrm{W} & \mathrm{Q} & \mathrm{V} & \mathrm{L} & &
\end{array}
$$

La condition de chevauchement n'est pas respectée étant donné que le ratio de chevauchement est de $\frac{5}{11}$ pour le gène $g_4$, ce qui est $< 0.70$. Ces deux gènes, $g_3$ et $g_4$, ne sont pas reliés par une arête dans le graphe $G$.

Soit l'alignement suivant des gènes $g_1$ et $g_2$

$$
\begin{array}{lcccccccccccc}
g_1: & & & \mathrm{A} & \mathrm{G} & \mathrm{L} & \mathrm{P} & \mathrm{V} & \mathrm{I} & \mathrm{E} & \mathrm{N} & \mathrm{L} & \mathrm{Q} \\
 & & & | & | & & | & | & | & & | & | & | \\
g_2: & \mathrm{Y} & \mathrm{L} & \mathrm{A} & \mathrm{G} & \mathrm{N} & \mathrm{P} & \mathrm{V} & \mathrm{I} & \mathrm{A} & \mathrm{N} & \mathrm{L} & \mathrm{Q}
\end{array}
$$

La condition de chevauchement est respectée car les ratios de chevauchement sont de $\frac{10}{10}$ et $\frac{10}{12}$ pour les gènes $g_1$ et $g_2$ respectivement, et sont $\geq 0.70$. La condition d'identité est également respectée car le ratio d'identité est de $\frac{8}{10}$, et est $\geq 0.30$. Ces deux gènes, $g_1$ et $g_2$, sont reliés par une arête dans le graphe $G$ et font donc partie de la même famille de gènes homologues.

Après avoir considéré tous les alignements des couples de gènes, nous obtenons le graphe $G$ de la figure 4.2 :

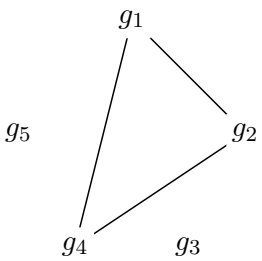



**Figure 4.2** Le graphe $G$ des familles d'homologues

Les familles d'homologues, qui sont représentées par les composantes connexes du graphe $G$, sont :

- $g_1$, $g_2$ et $g_4$ (étiquetés par 1)
- $g_3$ (étiqueté par 2)
- $g_5$ (étiqueté par 3)

Nous obtenons les deux suites signées suivantes :

$$S_1 : \quad 1\ 1\ 2$$
$$S_2 : \quad 1\ 3$$

## 4.2 Comparaison des génomes de *Bacillus subtilis* et *Escherichia coli*

Pour notre première expérience, nous avons utilisé un jeu de données composé des génomes *Bacillus subtilis* (*B. subtilis*) et *Escherichia coli* (*E. coli*). Ce choix était motivé par le fait que He et Goldwasser avaient comparé ces deux génomes à l'aide du modèle des équipes de gènes (He X. - Goldwasser M., 2004).

### 4.2.1 Données et paramètres utilisés

**Données.** Les fichiers contenant les séquences d'acides aminés pour les gènes proviennent des sites SubtiList (Lien Internet, Subtilist) pour l'organisme *B. subtilis* et de

Colibri (Lien Internet, Colibri) pour *E. coli K-12*. Ces fichiers contiennent les séquences de 4106 gènes codants pour *B. subtilis* et 4169 gènes codants pour *E. coli*.

Nous avons calculé les familles de gènes homologues et produit deux suites signées tel que décrit à la section 4.1. Nous avons obtenu 5816 familles dont les tailles sont données par le tableau de la table 4.1. Par exemple, il y a 4551 familles de 1 gène et 895 familles de 2 gènes.

| nombre de gènes | nombre de familles |
|---|---|
| 1 | 4551 |
| 2 | 895 |
| 3 | 173 |
| 4 | 76 |
| 5 | 48 |
| 6 | 18 |
| 7 | 10 |
| 8 | 14 |
| 9 | 9 |
| 10 | 4 |
| 11 | 1 |
| 12 | 2 |
| 13 | 2 |
| 14 | 3 |
| 15 | 3 |
| 16 | 2 |
| 19 | 1 |
| 20 | 3 |
| 22 | 1 |
| 43 | 1 |
| 44 | 1 |
| 107 | 1 |

**Table 4.1** La répartition des tailles des familles

La répartition de ces familles entre les deux organismes est décrite dans le tableau de la table 4.2. Par exemple, il y a 2506 familles qui ne contiennent que des gènes de *B. subtilis*.

| | nombre de familles |
|---|---|
| *B. subtilis* | 2506 |
| *E. coli* | 2505 |
| *B. subtilis* et *E. coli* | 807 |

**Table 4.2** La répartition des familles entre les organismes

**Paramètres de calcul des groupes des segments de gènes.** Nous avons utilisé $\delta = 2$ et $\omega = 1$ pour l'étape de la segmentation, décrite en section 3.3 et $\gamma = 0.5$ pour l'étape de la création des groupes, décrite en section 3.4.

### 4.2.2 Analyse des résultats

Nous présentons deux analyses des groupes calculés avec ce jeu de données. Dans un premier temps, nous comparons quantitivement nos résultats avec ceux de He et Goldwasser (He X. - Goldwasser M., 2004). Nous présentons ensuite quelques exemples qui illustrent l'utilité de la détection de groupes de gènes pour l'annotation de gènes.

**Comparaison avec He et Goldwasser.** Rappelons que He et Goldwasser ont proposé une extension du modèle des équipes de gènes permettant de prendre en compte la présence de plusieurs occurrences d'une même famille dans un génome, mais limitée à l'analyse de deux génomes (He X. - Goldwasser M., 2004). Les résultats de leur analyse des génomes de *B. subtilis* et *E. coli*, basée sur cette notion d'équipe de gènes et l'utilisation des COGs pour la construction des familles de gènes homologues sont disponibles en `http ://euler.slu.edu/~goldwasser/homologyteams/`

La table 4.3 présente une comparaison quantitative de nos résultats et ceux de He et Goldwasser.

| Nb. de familles dans un groupe | *B. subtilis* | *E. coli* | Deux bactéries | Total | He et Goldwasser |
|---|---|---|---|---|---|
| 2 | 9 | 10 | 51 | 70 | 65 |
| 3 | 6 | 11 | 20 | 37 | 16 |
| 4 | 3 | 5 | 19 | 27 | 3 |
| 5 | 2 | 3 | 6 | 11 | |
| 6 | | | 5 | 5 | |
| 7 | | 2 | 3 | 5 | |
| 8 | | 1 | 5 | 6 | |
| 9 | | | 2 | 2 | |
| 10 | 1 | | 2 | 3 | |
| 11 | 1 | 1 | | 2 | |
| 12 | | 1 | | 1 | |
| 16 | | | 1 | 1 | |
| 21 | | | | | 1 |
| 23 | | | 1 | 1 | |
| Total | 22 | 34 | 115 | 171 | 85 |

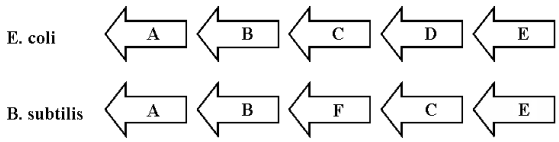
**Table 4.3** La comparaison avec He et Goldwasser

Par exemple, nous avons obtenu 70 groupes de segments de gènes contenant 2 familles de gènes. Parmi ces 70 groupes, il y en a 9 qui ne contiennent que des gènes de *B. subtilis*, 10 qui ne contiennent que des gènes de *E. coli* et 51 qui contiennent des gènes des deux bactéries. He et Goldwasser ont obtenu 65 groupes contenant 2 familles de gènes. Nos résultats sont comparables, étant donné que nos distributions de groupes sont semblables.

Toutefois nous avons obtenu le double de groupes au total, donc He et Goldwasser passent à côté de plusieurs groupes intéressants. Les figures 4.3 et 4.4 illustrent deux groupes que nous avons obtenus et que He et Goldwasser n'ont pas. Chaque flèche représente un gène et, lorsqu'elle pointe vers la droite, cela signifie que l'orientation du gène est positive sinon, l'orientation du gène est négative. Par exemple, dans la figure 4.3, il y a deux segments composés de cinq gènes chacun, dont le premier appartient à *E. coli* et le second à *B. subtilis*. Chacune des figures est composée d'une illustration d'un groupe des segments. Sous le groupe de segments, chacune des lignes contient l'information à propos d'un gène. Par exemple, la première ligne d'information de la

figure 4.3 indique que le gène *glgP* de l'organisme *E. coli* (EC.) appartient à la famille d'homologues étiquetée 848, est à la position 3253 dans le génome, son orientation est négative et son annotation fonctionnelle est «  Alpha-Glucan phosphorylase ». Pour des raisons de lisibilité, dans les flèches représentant les gènes nous avons ré-étiqueté les familles avec des lettres, dans ce cas-ci la famille 848 devient la famille *A*.

Dans le groupe de la figure 4.3, le gène *C* dans le segment de *E. coli* et le gène *F* dans le segment de *B. subtilis* ont été insérés. Dans le groupe de la figure 4.4, ce sont les gènes *F* et *G* dans le segment de *B. subtilis* qui ont été insérés.




A : 848 EC.glgP 3253 - Alpha-Glucan phosphorylase

B : 849 EC.glgA 3254 - Glycogen synthase

C : 850 EC.glgC 3255 - Glucose-1-phosphate adenylyltransferase

D : 5336 EC.glgX 3256 - Glycogen debranching enzyme

E : 851 EC.glgB 3257 - 1,4-alpha-Glucan branching enzyme

A : 848 BS.glgP 3094 - glycogen phosphorylase

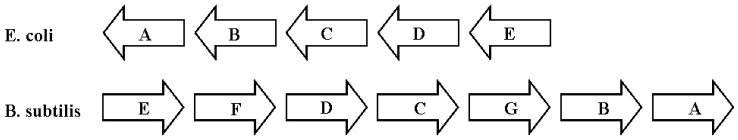
B : 849 BS.glgA 3095 - starch (bacterial glycogen) synthase

F : 3007 BS.glgD 3096 - required for glycogen biosynthesis

C : 850 BS.glgC 3097 - glucose-1-phosphate adenylyltransferase

E : 851 BS.glgB 3098 - 1,4-alpha-glucan branching enzyme

**Figure 4.3** Un groupe de segments non détecté par He et Goldwasser a)




A : 549 EC.pnp 3001 - Polynucleotide phosphorylase

B : 548 EC.rpsO 3002 - 30S ribosomal subunit protein S15

C : 546 EC.truB 3003 - Pseudouridine synthase, psi 55 specific pseudouridine

D : 545 EC.rbfA 3004 - Overexpression suppresses cold-sensitive 16S rRNA ; ribosome binding factor

E : 544 EC.infB 3005 - Protein chain initiation factor 2, IF2

E : 544 BS.infB 1663 + initiation factor IF-2

F : 2164 BS.ylxP 1664 + unknown ; similar to unknown proteins

D : 545 BS.rbfA 1665 + ribosome-binding factor A

C : 546 BS.truB 1666 + tRNA pseudouridine 5S synthase

G : 547 BS.ribC 1667 + riboflavin kinase / FAD synthase

B : 548 BS.rpsO 1668 + ribosomal protein S15 (BS18)

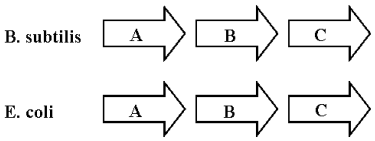
A : 549 BS.pnpA 1669 + polynucleotide phosphorylase (PNPase)

**Figure 4.4** Un groupe de segments non détecté par He et Goldwasser b)

**Annotation de gènes.** Nous présentons trois exemples (Figures 4.5, 4.6, 4.7) de groupes de segments pour lesquels la conservation de l'ordre des gènes entre *B. subtilis* et *E. coli* permet d'émettre une hypothèse sur la fonction d'un gène annoté « unknown ».

L'exemple de la figure 4.5 suggère que le gène *yloM* de *B. subtilis*, étiqué $C$ pourrait avoir la même annotation ou avoir la même fonction que le gène *rsmB* de *E. coli* étiqueté $C$. Dans l'exemple de la figure 4.6, il est intéressant de remarquer le réarrangement entre les gènes étiquetés $E$, $F$ et $G$. Les annotations des gènes ayant l'étiquette $G$ par exemple se ressemblent, même si dans le segment de *B. subtilis* le gène d'étiquette $G$ est à la septième position et dans le segments de *E. coli*, il est à la cinquième position. Dans cet exemple, le groupe suggère que le gène *yaeS* de *E. coli* pourrait avoir la même annotation

ou avoir la même fonction que le gène *uppS* de *B. subtilis*. L'exemple de la figure 4.7 suggère que le gène *yrbF* de *B. subtilis* pourrait avoir la même annotation ou avoir la même fonction que le gène *yajC* de *E. coli* même si, encore une fois, les positions ne correspondent pas en raison d'un réarrangement entre les deux segments.




A : 497 BS.def 1645812 + polypeptide deformylase

B : 498 BS.fmt 1646299 + methionyl-tRNA formyltransferase

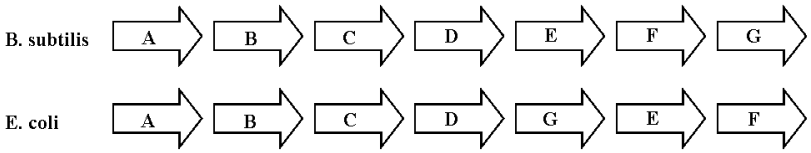
C : 499 BS.yloM 1647239 + unknown ; similar to RNA-binding Sun protein

A : 497 EC.def 7645609 + Peptide deformylase, N-formylmethionylaminoacyl-tRNA deformylase

B : 498 EC.fmt 7646133 + Methionyl-tRNA formyltransferase

C : 499 EC.rsmB 7647126 + 16S rRNA m5C967 SAM-dependent methyltransferase

**Figure 4.5** Annotation du gène *yloM* de *B. subtilis*




A : 535 BS.rpsB 1717226 + ribosomal protein S2

B : 536 BS.tsf 1718068 + elongation factor Ts

C : 537 BS.pyrH 1719095 + uridylate kinase

D : 538 BS.frr 1719819 + ribosome recycling factor

E : 539 BS.uppS 1720507 + probable undecaprenyl pyrophosphate synthetase

F : 540 BS.cdsA 1721293 + phosphatidate cytidylyltransferase

G : 541 BS.dxr 1722164 + probable 1-deoxy-D-xylulose-5-phosphate reductoisomerase

A : 535 EC.rpsB 4404156 + 30S ribosomal subunit protein S2

B : 536 EC.tsf 4405139 + EF-Ts, elongation factor for transcription, stable

C : 537 EC.pyrH 4406137 + UMP kinase

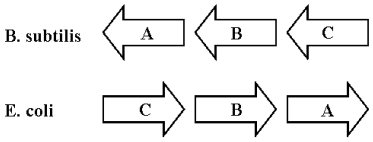
D : 538 EC.frr 4407154 + Ribosome recycling factor ; essential gene ; dissociates ribosomes from mRNA after termination of translation

G : 541 EC.dxr 4407803 + 1-deoxy-D-xylulose 5-phosphate reductoisomerase forms 2-C-methyl-D-erythritol 4-phosphate ; alternative nonmevalonate pathway for terpenoid biosynthesis

E : 539 EC.yaeS 4409185 + Function unknown

F : 540 EC.cdsA 4409959 + CDP-diglyceride synthase

**Figure 4.6** Annotation du gène *yaeS* de *E. coli*




A : 755 BS.yrbF 2831653 - unknown ; similar to unknown proteins

B : 756 BS.tgt 2831956 - tRNA-guanine transglycosylase

C : 757 BS.queA 2833128 - S-adenosylmethionine tRNA ribosyltransferase

C : 757 EC.queA 4638517 + S-Adenosylmethionine :tRNA ribosyltransferase- isomerase

B : 756 EC.tgt 4639643 + tRNA-guanine transglycosylase

A : 755 EC.yajC 4640793 + Part of Sec translocon

**Figure 4.7** Annotation du gène *yrbF* de *B. subtilis*

## 4.3 Comparaison de génomes de *Escherichia coli*, *Haemophilus influenzae* et *Salmonella thyphimurium*

Nous présentons maintenant une analyse de trois génomes : *Escherichia coli* (*E. coli*), *Salmonella thyphimurium* (*S. thyphimurium*) et *Haemophilus influenzae* (*H. influenzae*). Le but de cette analyse est double. Premièrement, comme *E. coli* et *S. thyphimurium* sont des organismes proches du point de vue évolutionnaire, nous pouvons avoir un premier aperçu de l'influence de la présence de génomes semblables dans le calcul de groupes de gènes conservés. Deuxièmement, nous avons choisi ces trois génomes car ils ont été utilisés par Chen *et al.* (Chen Y. *et al.*, 2004) pour la détection d'opérons dans le génome d'*E. coli* et nous pouvons donc comparer notre approche et celle de Chen *et al.* sur ce problème particulier.

### 4.3.1 Données et paramètres utilisés

Les données proviennent du site NCBI (Lien Internet, NCBI) pour les trois organismes. Plus précisément, nous avons utilisé les fichiers suivants : fichier RefSeq[1] NC_000913.2 pour *E. coli*, fichier RefSeq NC_003197.1 pour *S. thyphimurium* et NC_000907.1 pour *H. influenzae*.

Nous avons calculé les familles de gènes et produit trois suites signées selon la même méthode que pour la comparaison de *B. subtilis* et *E. coli*, décrite à la section 4.2.

Nous avons calculé nos groupes de gènes avec une valeur de 3 pour le paramètre $\delta$ et de 0.5 pour le paramètre $\gamma$. Pour le paramètre $\omega$, nous avons utilisé les cinq valeurs possibles, à savoir $\omega = 1, 2, 3, 4, 5$ ; nous avons ensuite concaténé les résultats pour les différentes valeurs de $\omega$ en ne gardant qu'une copie des groupes obtenus avec plusieurs valeurs de $\omega$. Nous avons effectué deux comparaisons, sur deux jeux de données :

- *E. coli*, *S. thyphimurium* et *H. influenzae*,
- *E. coli* et *H. influenzae*,

et nous avons analysé nos résultats en nous intéressant à *E. coli* qui est l'organisme dont le génome est le mieux connu.

### 4.3.2 Analyse des résultats

#### Analyse des groupes de gènes.

Nous avons obtenu 907 groupes lors de notre expérience avec les trois bactéries (*E. coli*, *H. influenzae* et *S. thyphimurium*). Il y a 855 segments de *E. coli* qui apparaissent parmi les 907 groupes, qui sont distribués ainsi :

- 18 segments appartiennent à des groupes qui sont composés uniquement de segments de *E. coli* ,

[1] `http ://www.ncbi.nlm.nih.gov/RefSeq/`

- 7 segments appartiennent à des groupes qui sont composés de segments de *E. coli* et *H. influenzae*,
- 783 segments appartiennent à des groupes qui sont composés de segments de *E. coli* et *S. thyphimurium* et
- 47 segments appartiennent à des groupes qui sont composés de segments de *E. coli*, *H. influenzae* et *S. thyphimurium*

Lors de la comparaison de *E. coli* et *H. influenzae*, nous avons obtenu exactement 378 groupes de gènes conservés, dont 321 contiennent des segments de *E. coli* et *H. influenzae* et 56 ne comportent que des segments de *E. coli*.

Parmi les 907 groupes obtenus en comparant les trois génomes, les segments de *E. coli* contiennent 3483 gènes différents. Rappelons que le fichier décrivant le génome de *E. coli* que nous avons utilisé comporte 4530 gènes. Les segments de *E. coli* présents dans les groupes calculés couvrent donc approximativement 75% des gènes, ce qui est un pourcentage assez élevé. Lors de l'expérience ne comportant que *E. coli* et *H. influenzae*, les 321 segments de *E. coli* ne couvrent que 1225 gènes, ce qui représente à peu près 25%.

Ces deux statistiques illustrent bien l'influence de la présence, dans le jeu de données, de deux génomes très semblables. Par exemple, le fait que près de 75% des gènes de *E. coli* se retrouvent dans au moins un groupe conservé lors de la comparaison des trois génomes, ne permet pas d'accorder une grande significativité à ce fait. Naturellement, la majorité de ces groupes ne contiennent que des segments des deux génomes très semblables : *E. coli* et *S. thyphimurium*. Il est plus probable que les segments de gènes intéressants sont les segments qui appartiennent à des groupes contenant aussi des segments de *H. influenzae*, ce qui se réduit à près de 25% des gènes de *E. coli*.

La table 4.4 présente les statistiques que nous venons d'évoquer.

| $\delta = 3$, $\gamma = 0.50$ | *E. coli*, *H. influenzae*, *S. thyphimurium* | *E. coli*, *H. influenzae* |
|---|---|---|
| Groupes | 907 | 378 |
| Segments de *E. coli* | 855 | 377 |
| *E. coli* | 18 | 56 |
| *E. coli*, *H. influenzae* | 7 | 321 |
| *E. coli*, *S. thyphimurium* | 783 | - |
| *E. coli*, *H. influenzae*, *S. thyphimurium* | 47 | - |
| Gènes de *E. coli* (4530) | 3483 | 1225 |

**Table 4.4** La répartition des segments

### 4.3.3 Analyse des opérons

Un opéron est un groupe de gènes consécutifs qui est transcrit en un seul ARNm. Les gènes qui forment un opéron sont en général assez proches physiquement : le nombre de bases séparant deux gènes consécutifs d'un opéron est plus faible que le nombre de bases séparant deux gènes transcrits en deux ARNm différents. De plus, les gènes qui forment un opéron ont tous la même orientation. Finalement il faut remarquer que dans la plupart des cas, les gènes qui constituent un opéron sont impliqués dans un même processus cellulaire ou biochimique. On peut voir les opérons comme des segments de gènes sur lesquels il existe une pression fonctionnelle pour conserver ces gènes proches, et donc un banc d'essai naturel pour notre méthode. Dans cette section, nous analysons les résultats décrits précédemment sur la comparaison de *E. coli*, *H. influenzae* et *S. thyphimurium* sous l'angle de la détection des opérons de *E. coli*.

Nous comparons aussi nos résultats avec ceux de Chen *et al.* (Chen Y. *et al.*, 2004), qui ont considéré les mêmes génomes, mais ont utilisé une approche spécifique à la détection d'opérons, basée sur l'utilisation de plusieurs outils et banques de données génomiques : COGNITOR (Lien Internet, COGNITOR), SIGSCAN (Lien Internet, SIGSCAN), PSSM (Lien Internet, PSSM), TRANSTERM (Lien Internet, TRANSTERM). Il nous a semblé intéressant de comparer notre approche générale et basée uniquement sur la conservation de groupes de gènes, avec une approche très spécifique.

Pour détecter des opérons potentiels dans *E. coli*, nous avons filtré les segments présents dans un groupe de gènes calculé par notre programme. Nous avons extrait les segments de *E. coli* constitués de gènes ayant la même orientation et tels que deux gènes consécutifs sont séparés par au plus 300 bases. Nous avons utilisé cette valeur de 300 bases car elle était utilisée par Chen *et al.* (Chen Y. *et al.*, 2004). Chacun des segments de *E. coli* ainsi obtenu est considéré comme un opéron potentiel.

Dans un premier temps, nous avons comparé les opérons prédits par notre méthode avec une liste des opérons connus chez *E. coli* (Salgado H. *et al.*, 2004). Dans sa version la plus récente, cette liste comporte 327 opérons, qui contiennent 1161 gènes. La table 4.5 présente nos résultats.

| | *E. coli*, *S. thyphimurium* et *H. influenzae* | *E. coli* et *H. influenzae* |
|---|---|---|
| Opérons prédits | 748 | 333 |
| Vrais positifs | 278 | 181 |
| Opérons exacts | 98 | 72 |
| Faux positifs | 470 | 176 |
| Faux négatifs | 28 | 176 |
| Gènes manquants (sur 1161) | 119 | 565 |

**Table 4.5** Les résultats des comparaisons de trois et deux bactéries

Dans ce tableau,

- les vrais positifs sont les opérons prédits qui contiennent au moins un gène appartenant à un opéron connu,
- les opérons exacts sont les opérons prédits qui correspondent exactement à un opéron connu,
- les faux positifs sont les opérons prédits qui ne contiennent aucun gène appartenant à un opéron connu,
- les faux négatifs sont les opérons de *E. coli* dont aucun gène n'apparaît dans un opéron prédit,
- les gènes manquants sont les gènes appartenant à un opéron connu et n'apparaissant dans aucun opéron prédit.

Nous avons aussi analysé les segments de gènes de *E. coli* produits par notre algorithme et contenant des gènes de plusieurs opérons, et inversement les opérons de *E. coli* « éclatés » par nos prédictions en plusieurs segments. Ces résultats sont présentés dans la table 4.6.

| | *E. coli*, *S. thyphimurium* et *H. influenzae* | | | | | *E. coli* et *H. influenzae* | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Candidats | | | Opérons | | | | | | |
| Distribution | 470 | 1 | → | 0 | | 152 | 1 | → | 0 | |
| | 256 | 1 | → | 1 | | 173 | 1 | → | 1 | |
| | 18 | 1 | → | 2 | | 7 | 1 | → | 2 | |
| | 3 | 1 | → | 3 | | 1 | 1 | → | 3 | |
| | 1 | 1 | → | 4 | | | | | | |
| | | | | | | | 0 | ← | 1 | 149 |
| | | | | | | | 1 | ← | 1 | 167 |
| | | 0 | ← | 1 | 28 | | 2 | ← | 1 | 10 |
| | | 1 | ← | 1 | 293 | | 3 | ← | 1 | 1 |
| | | 2 | ← | 1 | 6 | | | | | |

**Table 4.6** Les distributions des comparaisons de trois et deux bactéries

Par exemple, dans le cas de la comparaison des trois génomes,

- pour 470 opérons prédits, aucun gène du segment n'appartient à un opéron,
- 256 opérons prédits contiennent des gènes présents dans 1 opéron connu,
- 18 opérons prédits contiennent des gènes présents dans 2 opérons connus,
- 3 opérons prédits contiennent des gènes présents dans 3 opérons connus,
- 1 opéron prédit contien des gènes présents dans 4 opérons connus,
- 28 opérons connus n'ont aucun gène dans un opéron prédit,
- 293 opérons connus ont certains de leurs gènes dans un seul opéron prédit,
- 6 opérons connus ont certains de leurs gènes répartis dans deux opérons prédits.

On peut remarquer que la comparaison des trois génomes permet d'exhiber presque tous les opérons connus de *E. coli*, ce qui n'est pas surprenant car, comme nous l'avons décrit précédemment, presque 75% des gènes de *E. coli* sont couverts par des segments appartenant à un groupe conservé. La contrepartie de cette grande sensibilité dans la prédiction des opérons est une spécificité assez faible, illustrée par 470 faux positifs. Au contraire,

la comparaison de *E. coli* et *H. influenzae* fournit des résultats très intéressants. Par exemple, le ratio entre vrais positifs et faux positifs devient meilleur en ne comparant que ces deux génomes. Ceci permet de considérer qu'un opéron prédit par la comparaison des deux génomes a de meilleures chances d'être un vrai opéron qu'un opéron prédit par la comparaison des trois génomes. On peut aussi noter la croissance du taux d'opérons exacts qui sont prédits, qui illustre encore une prédiction de meilleure qualité.

Dans un deuxième temps, nous avons comparé nos opérons prédits avec ceux prédits par l'approche de Chen *et al.*. Rappelons que leur approche spécifique à la détection des opérons utilise plusieurs outils et banques de données génomiques. Les résultats de nos deux approches, lors des expérimentations sur les deux bactéries *E. coli* et *H. influenzae*, sont illustrés par les tables 4.7 et 4.8.

Nous devons mentionner que la version de la liste des opérons connus que nous utilisons (Salgado H. *et al.*, 2004) est plus récente que celle utilisée par Chen *et al.*. Dans notre version, la liste comporte 327 opérons, qui contiennent 1161 gènes, tandis que la liste de la version de Chen *et al.* comporte 237 opérons.

| $\delta = 3$, $\gamma = 0.50$, post-traitement | Notre approche (327) | Chen *et al.* (237) |
|---|---|---|
| Opérons candidats | 333 | 237 |
| Opérons exacts | 72 | 61 |
| Faux positifs | 152 | 176 |
| Faux négatifs | 149 | 176 |
| Gènes manquants (sur 1161) | 565 | - |

**Table 4.7** Les résultats de la comparaison avec Chen *et al.* des deux bactéries

| | *E. coli*, *S. thyphimurium* et *H. influenzae* | | | | | *E. coli* et *H. influenzae* |
|---|---|---|---|---|---|---|
| | Candidats | | | Opérons | | |
| Distribution | 152 | 1 | → | 0 | | - |
| | 173 | 1 | → | 1 | | |
| | 7 | 1 | → | 2 | | |
| | 1 | 1 | → | 3 | | |
| | | 0 | ← | 1 | 149 | |
| | | 1 | ← | 1 | 167 | |
| | | 2 | ← | 1 | 10 | |
| | | 3 | ← | 1 | 1 | |

**Table 4.8** Les distributions de la comparaison avec Chen *et al.* des deux bactéries

Nous pouvons remarquer que nous obtenons un peu plus de d'opérons candidats et d'opérons exacts, tout en obtenant un peu moins de faux positifs et de faux négatifs.

## 4.4 Comparaison de génomes de douze bactéries

Nous avons comparé douze génomes de bactéries de $\gamma$-Proteobacteria et nous avons concentré l'analyse de nos résultats sur un opéron bien connu, l'opéron tryptophane. Les fichiers des résultats de cette expérimentation sont disponibles sur Internet (`http ://adn.bioinfo.uqam.ca/~genoc/CLUSTERS`).

### 4.4.1 Données

Nos données proviennent de la base de données des génomes complets de bactéries du NCBI[2]. Les douze bactéries sont les suivantes : *Escherichia coli K12* (*E. coli*, numéro d'accession du fichier RefSeq NC_000913), *Salmonella typhimurium* (*S. typhimurium*, NC_003197), *Yersinia pestis CO92* (*Y. pestis CO92*, NC_003143), *Yersinia pestis KIM* (*Y. pestis KIM*, NC_004088), *Buchnera aphidicola* (*B. aphidicola*, NC_002528), *Wigglesworthia glossinidia* (*W. glossinidia*, NC_004344), *Haemophilus influenzae* (*H. in-*

[2] `http ://www.ncbi.nlm.nih.gov/genomes/lproks.cgi`

*fluenzae*, NC_000907), *Pasteurella multocida* (*P. multocida*, NC_002663), *Pseudomonas aeruginosa* (*P. aeruginosa*, NC_002516), *Xylella fastidiosa*, (*X. fastidiosa*, NC_002488) *Xanthomonas axonopodis* (*X. axonopodis*, NC_003919) et *Xanthomonas campestris* (*X. campestris*, NC_003902).

Ce jeu de données est intéressant car il contient des bactéries très différentes et éloignées les unes des autres, et en même temps trois paires de bactéries semblables et proches l'une de l'autre (*E. coli* et *S. typhimurium*, les deux *Yersina pestis*, et les deux *Xanthomonas*).

### 4.4.2 Similarité

Nous avons considéré tous les gènes codants, les ARNt et les ARNr. Pour les ARNt et les ARNr, nous nous sommes basé sur leur annotation pour la construction des familles d'homologues. Si deux annotations sont identiques, alors les deux ARNt ou ARNr feront partie de la même famille d'homologues. En ce qui concerne les gènes codants, nous avons construit les familles d'homologues tel que décrit à la section 4.1.

### 4.4.3 Statistiques

Le nombre total de gènes de tous les génomes est de 39 050 gènes et nous avons obtenu 11 771 familles d'homologues. Sur ces 11 771 familles, il y en a 6 221 qui ne contiennent qu'un seul gène et par conséquent, ne se retrouvent que dans un seul génome. Une de ces familles contient 434 gènes et est composée exclusivement de gènes codants pour des protéines liant l'ATP. En tout, 6 538 familles ne sont présentes que dans un seul génome, 679 familles sont présentes dans au moins dix génomes et 284 de ces 679 familles sont présentes dans les douze génomes.

Notre expérimentation se divise en deux parties. La première partie consiste à comparer les douze génomes entre eux. Pour la deuxième partie, nous avons enlevé trois génomes (*S. typhimurium*, *Y. pestis KIM* et *X. campestris*) de sorte que les paires de génomes semblables soient brisées. Dans les deux cas, nous avons utilisé $\delta = 2$ et 3, pour chaque

valeur de $\delta$, nous avons utilisé $\omega = 1$ , ..., $2\delta$ - 1 et $\gamma = 0.50$, 0.60, 0.70, 0.80, 0.90 et 1. Nous obtenons donc 48 combinaisons différentes de paramètres.

Le calcul des groupes de segments pour les douze génomes s'est effectué approximativement en dix minutes et pour les neuf génomes, le calcul s'est effectué en quatre minutes. La différence notable de temps s'explique par l'absence des paires de génomes semblables, qui réduit le nombre de $\delta$-segments conservés et donc le temps de calcul pour la création des groupes.

Rappelons que notre algorithme construit un graphe dont les sommets sont des segments et les composantes connexes sont des groupes de segments. En terme d'espace, le plus grand graphe obtenu pour la comparaison des douze génomes a 7 688 sommets ($\delta = 3$, $\omega = 1$ et $\gamma = 0.50$) et pour la comparaison des neuf génomes, le plus grand graphe a 10 176 sommets. Cette augmentation du nombre de segments s'explique par l'éclatement de longs segments entre les paires de génomes semblables. Pour cette expérimentation, nous pouvons penser que la complexité spatiale de l'algorithme, en pratique, est linéaire par rapport au nombre total de gènes.

Le nombre de groupes de segments pour les douze génomes varie de 199 groupes ($\delta = 3$, $\omega = 2$ et $\gamma = 1$) à 930 groupes ($\delta = 2$, $\omega = 2$ et $\gamma = 0.60$). En ce qui concerne l'expérimentation sur les neuf génomes, le nombre de groupes de segments varie de 249 groupes ($\delta = 3$, $\omega = 5$ et $\gamma = 1$) à 913 groupes ($\delta = 2$, $\omega = 1$ et $\gamma = 0.60$).

Nous avons calculé la couverture en gènes pour les génomes de *E. coli* et *H. influenzae*, c'est-à-dire le nombre de gènes présents dans au moins un groupe. Selon les données de notre expérimentation, le génome de *E. coli* est composé de 4 350 gènes. Le nombre de gènes de *E. coli* présents dans au moins un groupe varie de 183 gènes ($\delta = 3$, $\omega = 1$ et $\gamma = 1$) à 3 111 gènes ($\delta = 2$, $\omega = 1$ et $\gamma = 0.50$) pour la comparaison des douze génomes. Ce nombre varie de 359 gènes ($\delta = 3$, $\omega = 1$ et $\gamma = 1$) à 2 635 gènes ($\delta = 2$, $\omega = 1$ et $\gamma = 0.50$) pour la comparaison des neufs génomes. Finalement, le génome de *H. influenzae* est composé de 1 732 gènes, selon les données de notre expérimentation. Le nombre de gènes de *H. influenzae* présents dans au moins un groupe varie de 160 gènes

($\delta = 3$, $\omega = 4$ et $\gamma = 0.90$) à 1182 gènes ($\delta = 2$, $\omega = 1$ et $\gamma = 0.50$) pour la comparaison des douze génomes. Ce nombre varie de 177 gènes ($\delta = 3$, $\omega = 5$ et $\gamma = 1$) à 1208 gènes ($\delta = 2$, $\omega = 1$ et $\gamma = 0.50$) pour la comparaison des neufs génomes.

### 4.4.4 L'opéron tryptophane

De manière à avoir un premier aperçu des résultats de notre algorithme, nous avons vérifié si l'opéron tryptophane est présent dans nos résultats. L'opéron tryptophane est composé des gènes *trpA*, *trpB*, *trpC*, *trpD* et *trpE* (Xie G. *et al.*, 2003). Il est à noter qu'aucun gène de cet opéron n'est présent chez la bactérie *W. glossinidia*. La construction des familles d'homologues nous donne les résultats suivants :

- le gène *trpA* forme la famille étiquetée[3] 1 et le gène *trpB* forme la famille 2,
- le gène *trpC* forme deux familles : la famille 3 (*E. coli*, *S. typhimurium*, les deux *Y. pestis*, *B. aphidicola*, *H. influenzae* et *P. multocida*) et la famille 7 (*P. aeruginosa*, *X. fastidiosa*, *X. axonopodis* et *X. campestris*),
- le gène *trpD* forme aussi deux familles : la famille 4 (toutes les bactéries exceptées *E. coli*, *S. typhimurium* et *Y. pestis KIM*) et la famille 5 (*E. coli* et *S. typhimurium*; aucun gène nommé *trpD* n'apparaît dans la bactérie *Y. pestis KIM* mais l'annotation du gène correspondant dans le fichier NC_004088 indique « disrupted by frameshift » (décalage du cadre de lecture)),
- le gène *trpE* forme la famille 6 ; la bactérie *X. fastidiosa* contient trois gènes appartenant à cette famille et les bactéries *X. axonopodis*, *X. campestris* et *P. aeruginosa* contiennent deux gènes chacune.

---

[3]Pour des raisons de lisibilité, nous avons ré-étiqueté les familles correspondant aux gènes de l'opéron tryptophane. Nous avons étiqueté par le nombre 1 la famille 260 qui avait été produite par notre algorithme, la famille 261 devient la famille 2, 262 devient 3, 263 devient 4, 1428 devient 5, 1429 devient 6 et 3944 devient 7. Nous n'avons pas ré-étiqueté les autres familles qui apparaissent dans les flèches pointillées dans nos figures.

La figure 4.8 illustre un groupe de segments, où les gènes représentés par des flèches pleines sont liés aux gènes de l'opéron tryptophane et les gènes représentés par des flèches pointillées n'ont pas de lien avec les gènes de cet opéron. Nous avons ajouté dans les flèches le nom des gènes correspondant aux familles appartenant à l'opéron tryptophane. Les nombres au dessus des premières et dernières flèches de chaque segment indique les positions de début et de fin dans le génome. Nous pouvons remarquer la grande variation dans l'ordre des gènes parmi les segments de l'opéron, en particulier l'insertion des gènes étiquetés 98 ainsi que plusieurs réarrangements qui ont parfois scindé cet opéron en plusieurs segments.

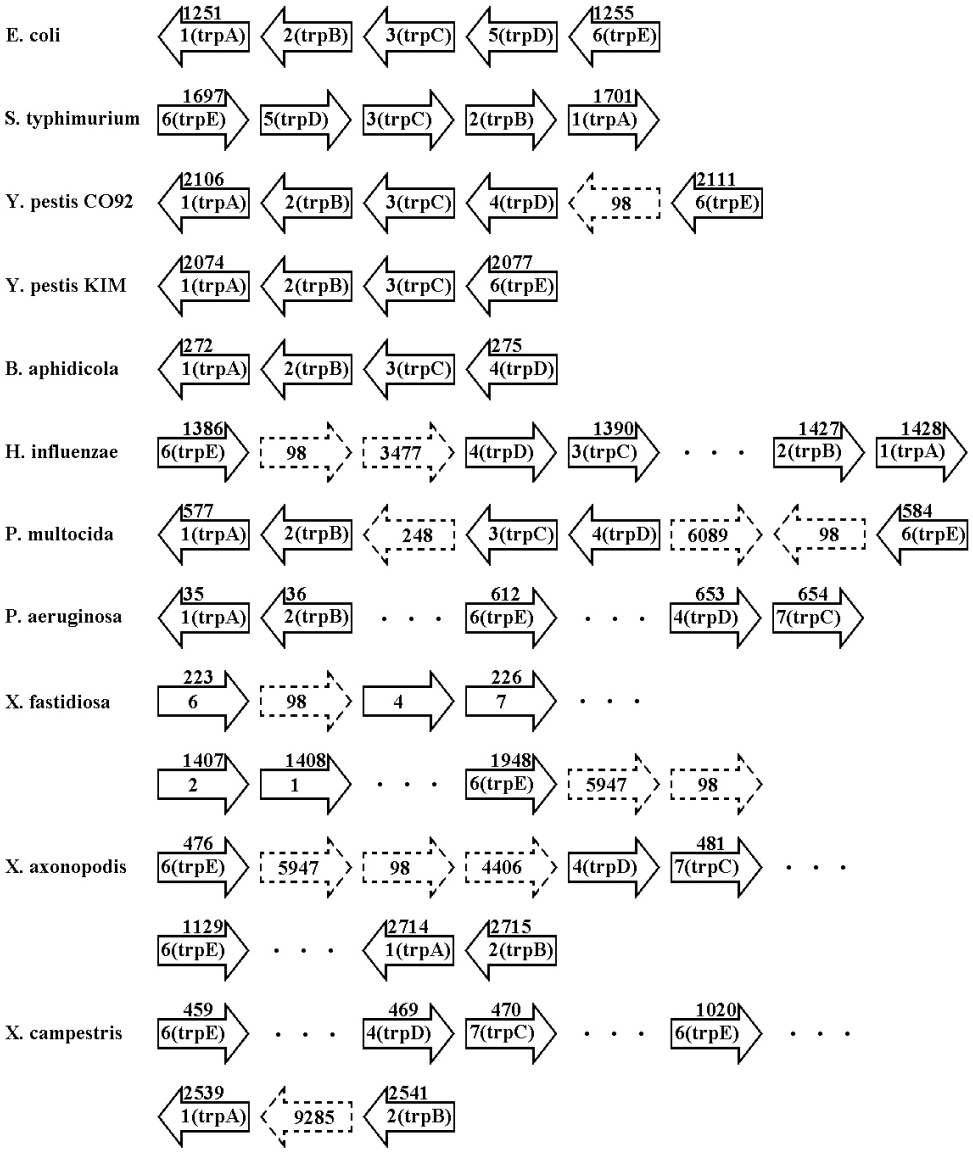



**Figure 4.8** L'opéron tryptophane

Nous avons calculé les groupes de segments pour les deux expérimentations (douze génomes et neuf génomes) avec les paramètres $\delta = 2$, $\gamma = 0.50$ et $\omega = 1, 2, 3$. Nous avons considéré toutes les valeurs possibles pour le paramètre $\omega$ dans le but d'étudier l'utilité de ce paramètre dans la détection des groupes de gènes intéressants. Nous avons ensuite filtré chaque liste de groupes afin de conserver uniquement les groupes de

segments contenant au moins un gène appartenant à une des familles étiquetées de 1 à 7 (voir note 3, p. 88).

Pour l'expérimentation avec douze génomes, nous avons obtenu onze groupes avec le paramètre $\omega = 1$ et huit groupes chacun avec les paramètres $\omega = 2$ et $\omega = 3$. Le groupe le plus intéressant est obtenu avec le paramètre $\omega = 2$, qui a regroupé la majorité des segments de l'opéron tryptophane. Ce groupe est illustré par la figure 4.9.

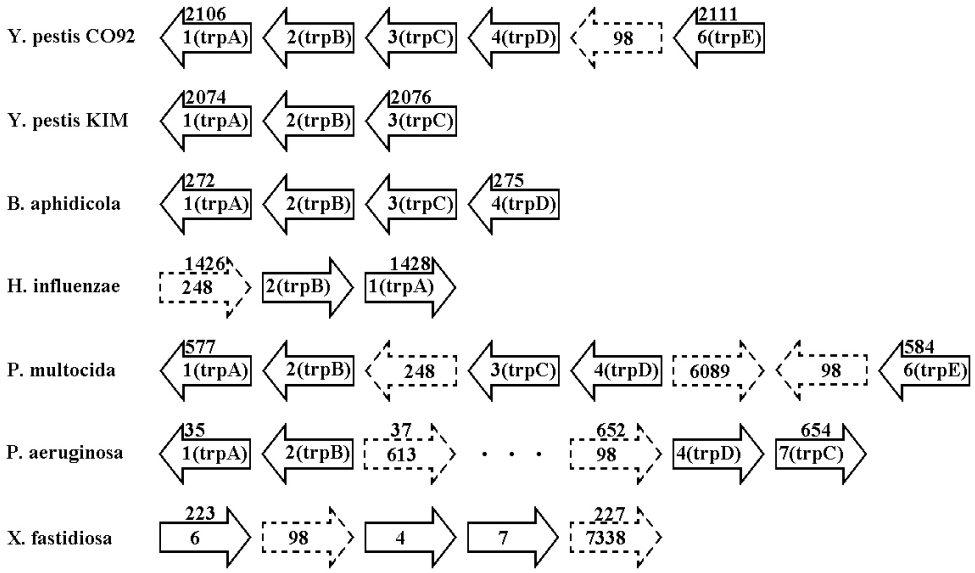
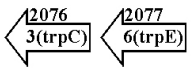


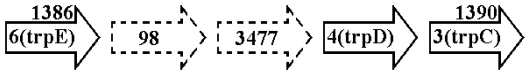

**Figure 4.9** Un groupe obtenu lors de l'expérimentation avec douze génomes avec les paramètres $\delta = 2$, $\omega = 2$ et $\gamma = 0.50$

Nous avons noté que ce groupe de la figure 4.9 ne capture pas le gène *trpE* à la position 2077 du génome de la bactérie *Y. pestis KIM*. Ceci vient du fait que le $\delta$-segment suivant :

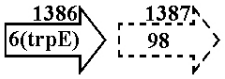



appartenant à cette bactérie n'est pas conservé dans les autres avec le paramètre $\delta = 2$. Dans ce groupe, il manque trois segments contenant un seul gène, le gène *trpE*, ce qui

est normal étant donné qu'un $\delta$-segment conservé contient au moins deux gènes (voir section 3.3). Le segment suivant :




de la bactérie *H. influenzae* n'apparaît pas dans le groupe car l'insertion du gène 3 477 réduit le poids du $\delta$-segment suivant :




à la valeur 1. En effet, avec le paramètre $\omega = 1$, ce segment apparaît dans la figure 4.10. Ceci illustre l'utilité de considérer toutes les combinaisons possibles du paramètre $\omega$.

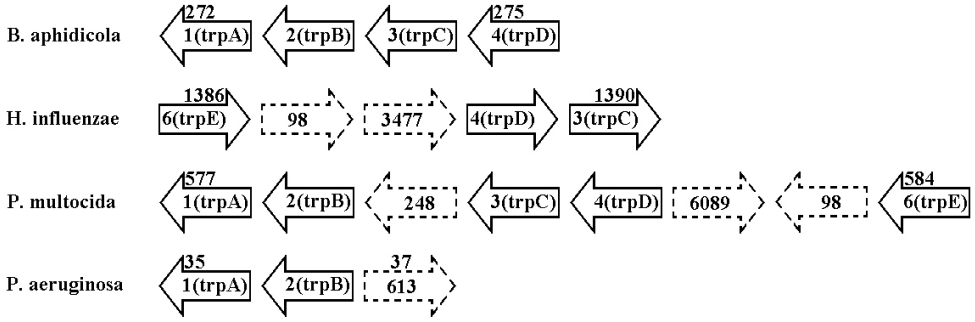



**Figure 4.10** Un groupe obtenu lors de l'expérimentation avec douze génomes avec les paramètres $\delta = 2$, $\omega = 1$ et $\gamma = 0.50$

Pour l'expérimentation avec neuf génomes, nous avons obtenu six groupes avec le paramètre $\omega = 1$, trois groupes avec $\omega = 2$ et deux groupes avec $\omega = 3$. Nous décrivons par les figures 4.11 et 4.12 deux groupes de segments intéressants illustrant l'influence de l'absence des trois paires de génomes semblables, car ces groupes capturent les segments de *E. coli* de l'opéron tryptophane.

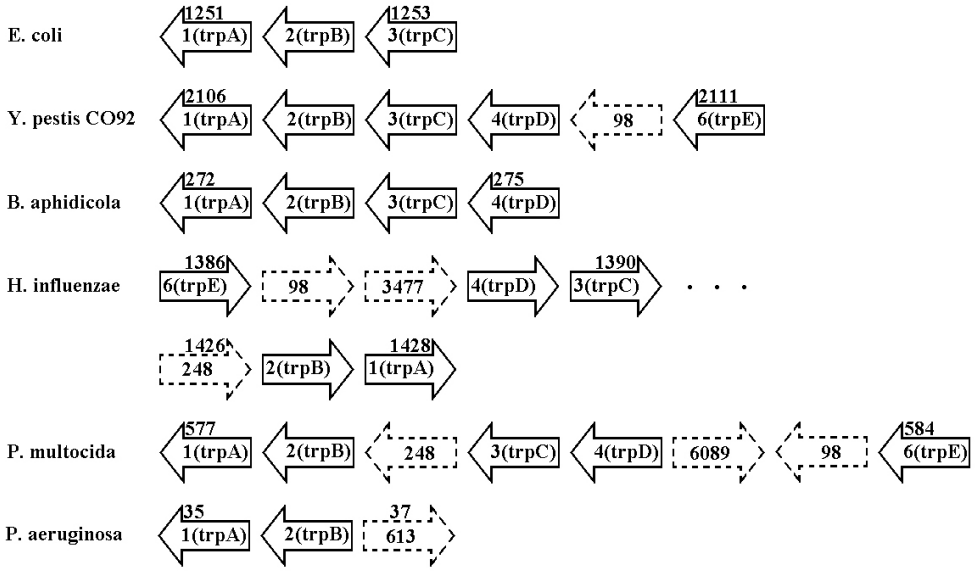



**Figure 4.11** Un groupe obtenu lors de l'expérimentation avec neuf génomes avec les paramètres $\delta = 2$, $\omega = 1$ et $\gamma = 0.50$

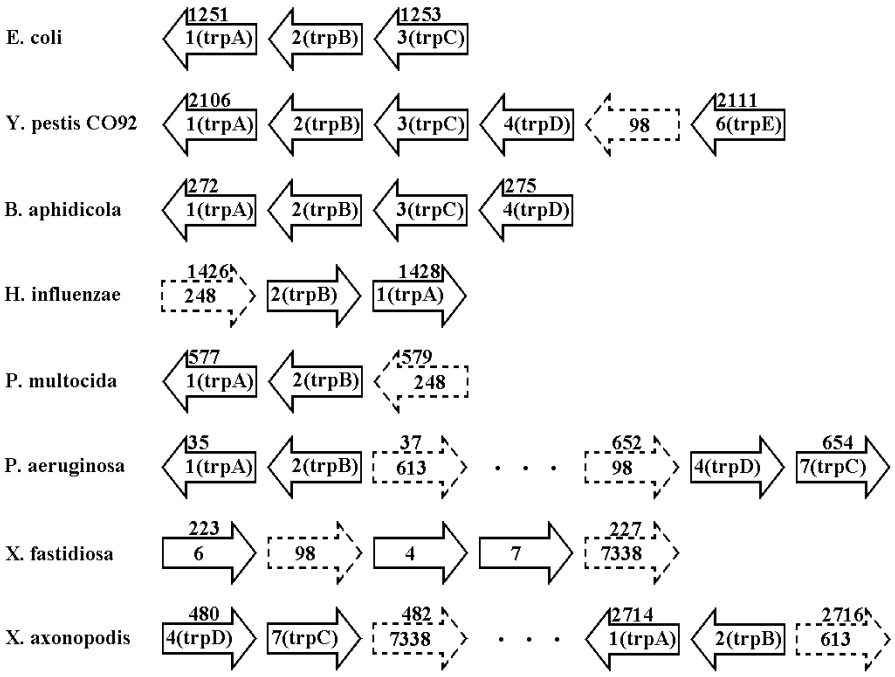



**Figure 4.12** Un groupe obtenu lors de l'expérimentation avec neuf génomes avec les paramètres $\delta = 2$, $\omega = 2$ et $\gamma = 0.50$

# CONCLUSION

Le calcul du modèle général de regroupements de gènes nécessite un temps d'exécution exponentiel (Pasek S. *et al.*, 2005). Nous avons présenté un algorithme efficace permettant de considérer plusieurs copies d'un gène dans un génome et pouvant comparer plus de deux génomes à la fois.

Notre modèle, simple et efficace, a donné des résultats très intéressants lors de nos trois expérimentations. Plus particulièrement pour la détection des opérons, notre algorithme a obtenu des résultats de même ou de meilleure qualité qu'un algorithme conçu spécialement à cet effet (Chen Y. *et al.*, 2004).

À partir de nos travaux, nous avons publié un article :

- St-Onge, K., Chauve, C., et Bergeron, A.. (2004) Duplications and Whole Genome Comparison : A Pragmatic Approach. Actes de «  Journées Ouvertes Biologie, Informatique, Mathématiques (JOBIM'04, Montréal, Juin 2004) », édités par M. F. Sagot et F. Major, article JO-75.

Nous prévoyons créer une page web afin de rendre disponible notre algorithme sur internet. Nous travaillons également sur un projet de visualisation des groupes de segments produits par notre algorithme.

# Annexe A

# LES SCRIPTS ET LE PROGRAMME

Cette section contient le code source de nos scripts et de notre programme, qui ont été écrits par Cedric Chauve et Karine St-Onge.

## A.1 Les scripts

### A.1.1 blast-2-data.c

```
// Computing the gene families from a Blast-graph file
// Cedric Chauve, february 2005

#include <stdio.h>
#include <assert.h>

static int FAM, SIZEFAM;

static int   NBG; // Number of genomes
static int   NBL; // Size of the graph file

static int  *SIZE, *STATFAM, **SIGNS, **START, **END;
static FILE **FILES, *GRAPH;
static char *** ANNOT;

// --------------------------------------------------

typedef struct neighbor {
  int genome;
  int pos;
  int family;
  struct neighbor *next;
```

```
} *NEIGHBOR;

static NEIGHBOR **GENES;

static void ADD_EDGE(int k1, int i1, int k2, int i2) {
  if ( GENES[k1-1][i1-1]->next == NULL ) {
    GENES[k1-1][i1-1]->next = (NEIGHBOR) malloc(sizeof(struct neighbor));
    GENES[k1-1][i1-1]->next->genome = k2;
    GENES[k1-1][i1-1]->next->pos    = i2;
    GENES[k1-1][i1-1]->next->family = 0;
    GENES[k1-1][i1-1]->next->next   = NULL;
  }
  else {
    NEIGHBOR ngb = (NEIGHBOR) malloc(sizeof(struct neighbor));
    ngb->genome = k2;
    ngb->pos    = i2;
    ngb->family = 0;
    ngb->next   = GENES[k1-1][i1-1]->next;
    GENES[k1-1][i1-1]->next = ngb;
  }
}

// -------------------------------------------------

static void DFS(int k1, int i1) {
  int k2, i2;
  NEIGHBOR ngb;

  if ( GENES[k1-1][i1-1]->family == 0 ) {
    printf("%5d %5d %s", k1, i1, ANNOT[k1-1][i1-1]);
    GENES[k1-1][i1-1]->family = FAM;
    SIZEFAM++;
    ngb = GENES[k1-1][i1-1]->next;
    while ( ngb != NULL ) {
      DFS(ngb->genome, ngb->pos);
      ngb = ngb->next;
    }
  }
}

#define LG 200

main (int argc, char *argv[]) {
  int   i, j, c, d;
```

```
int   k1, k2, i1, i2;
char *line  = (char *) malloc(LG * sizeof(char));
char *field = (char *) malloc(LG * sizeof(char));

NBG   = atoi(argv[1]);
GRAPH = fopen(argv[2], "r");
NBL   = atoi(argv[3]);
FAM   = atoi(argv[4]);

GENES  = (NEIGHBOR **)  malloc(NBG * sizeof(NEIGHBOR *));
SIGNS  = (int **)  malloc(NBG * sizeof(int *));
START  = (int **)  malloc(NBG * sizeof(int *));
END    = (int **)  malloc(NBG * sizeof(int *));
SIZE   = (int *)   malloc(NBG * sizeof(int));
FILES  = (FILE **) malloc(3 * NBG * sizeof(FILE *));
ANNOT  = (char ***) malloc(NBG * sizeof(char **));
STATFAM= (int *)   calloc(NBL, sizeof(int));

for ( i = 0; i < NBG; i++ ) {
  //printf("----LECTURE GENOME %d\n", i);
  FILES[i]       = fopen(argv[5+4*i], "w");
  FILES[NBG+i]   = fopen(argv[6+4*i], "r");
  FILES[2*NBG+i] = fopen(argv[7+4*i], "r");
  SIZE[i]        = atoi(argv[8+4*i]);
  GENES[i]       = (NEIGHBOR *) calloc(SIZE[i], sizeof(NEIGHBOR));
  for ( j = 0; j < SIZE[i]; j++ ) {
    GENES[i][j] = (NEIGHBOR) malloc(sizeof(struct neighbor));
  }
  ANNOT[i] = (char **) malloc(SIZE[i] * sizeof(char *));
  for ( j = 0; j < SIZE[i]; j++ ) {
    line = fgets(line, LG, FILES[2*NBG+i]);
    ANNOT[i][j] = (char *) malloc(LG * sizeof(char));
    ANNOT[i][j] = strcpy(ANNOT[i][j], line);
    //ANNOT[i][j] = (char *) realloc(ANNOT[i][j], strlen(line) * sizeof(char));
    //printf("------- %s", ANNOT[i][j]);
    GENES[i][j]->genome = 0;
    GENES[i][j]->pos    = 0;
    GENES[i][j]->family = 0;
    GENES[i][j]->next   = NULL;
  }
  //printf("----LECTURE GENOME %d\n", i);
}

// Reading the edges of the graph
```

```
  for ( j = 1; j <= NBL; j++ ) {
    line = fgets(line, LG, GRAPH);

    c = 0; while ( line[c] != ' ' ) { field[c] = line[c++]; }
    field[c++] = '\0'; k1 = atoi(field);
    d = c; while ( line[c] != ' ' ) { field[c-d] = line[c++]; }
    field[c++-d] = '\0'; i1 = atoi(field);
    d = c; while ( line[c] != ' ' ) { field[c-d] = line[c++]; }
    field[c++-d] = '\0'; k2 = atoi(field);
    d = c; while ( line[c] != '\0' ) { field[c-d] = line[c++]; }
    field[c++-d] = '\0'; i2 = atoi(field);

    ADD_EDGE(k1, i1, k2, i2);
    ADD_EDGE(k2, i2, k1, i1);
  }
  fclose(GRAPH);

  for ( k1 = 1; k1 <= NBG; k1++ ) {
    for ( i1 = 1; i1 <= SIZE[k1-1]; i1++ ) {
      if ( GENES[k1-1][i1-1]->family == 0 ) {
SIZEFAM = 0;
FAM++;
printf(" ******** FAMILY %d ***************\n", FAM);
DFS(k1, i1);
STATFAM[SIZEFAM]++;
printf(" -------- FAMILY %d SIZE %d---------------\n", FAM, SIZEFAM);
      }
    }
  }

  for ( j = 0; j < NBL; j++ ) {
    if ( STATFAM[j] > 0 ) printf("[%d,%d]", j, STATFAM[j]);
  }
  printf("\n%d\n", FAM);

  for ( k1 = 1; k1 <= NBG; k1++ ) {
    for ( i1 = 1; i1 <= SIZE[k1-1]; i1++ ) {
      line = fgets(line, LG, FILES[NBG+k1-1]);
      fprintf(FILES[k1-1], "%d %s", GENES[k1-1][i1-1]->family, line);
    }
    fclose(FILES[k1-1]);
    fclose(FILES[NBG+k1-1]);
  }
}
```

### A.1.2 blast-2-families-graph.awk

```
BEGIN {
  FIRST=0;
  NUMBER=0;
}

{
  if ( $1 == "#" ) {
    FIRST = 0;
  }
  else {
    if ( FIRST == 0 ) {
      FIRST = 1;
      NUMBER++;
    }

    NUM1=$1;
    j1 = 1;      while ( substr(NUM1, j1, 1) != ":" ) { j1++; }
    j1++;        while ( substr(NUM1, j1, 1) != ":" ) { j1++; }
    j1++; k = 1; while ( substr(NUM1, j1+k, 1) != ":" ) { k++; }
    S1 = substr(NUM1, j1, k);
    j1 += k+1; k = 1; while ( substr(NUM1, j1+k, 1) != ":" ) { k++; }
    E1 = substr(NUM1, j1, k);
    j1 += k+1;
    L1 = E1-S1+1;

    NUM2=$2;
    j2 = 1;      while ( substr(NUM2, j2, 1) != ":" ) { j2++; }
    j2++;        while ( substr(NUM2, j2, 1) != ":" ) { j2++; }
    j2++; k = 1; while ( substr(NUM2, j2+k, 1) != ":" ) { k++; }
    S2 = substr(NUM2, j2, k);
    j2 += k+1; k = 1; while ( substr(NUM2, j2+k, 1) != ":" ) { k++; }
    E2 = substr(NUM2, j2, k);
    j2 += k+1;
    L2 = E2-S2+1;

    if ( ($3 >= ID) && (($8 - $7 + 1) >= (OV/100 * L1/3)) && (($10 - $9 + 1) >= (OV/100 * L2/3)) ) {
      k = 1; while ( substr(NUM1, j1+k, 1) != ":" ) { k++; }
      printf("%s ", substr(NUM1, j1, k));
      j1 += k+1; k = 1; while ( j1+k <= length(NUM1) ) {  k++; }
      printf("%s ", substr(NUM1, j1, k));

      k = 1; while ( substr(NUM2, j2+k, 1) != ":" ) { k++; }
```

```
      printf("%s ", substr(NUM2, j2, k));
      j2 += k+1; k = 1; while ( j2+k <= length(NUM2) ) {  k++; }
      printf("%s\n", substr(NUM2, j2, k));
    }
  }
}


END {
}
```

### A.1.3 convert-genbank-2-genes-file.perl

```
#!/usr/bin/perl

use Bio::Seq;
use Bio::SeqIO;

$input_seqs = Bio::SeqIO->new ( '-format' => 'GenBank',
                                '-file'   => readdir(IN)
                              );

while ( $s = $input_seqs->next_seq() ) {

    foreach $f ( $s->all_SeqFeatures() ) {

if ( (! $f->has_tag("pseudo")) &&
     ($f->primary_tag eq "CDS" || (($f->primary_tag eq "tRNA" || $f->primary_tag eq "rRNA") && $f->has_tag("

    print "> ", $f->primary_tag, " ";
    if ( $f->strand eq "1" ) { print "+ "; }
    else { print "- "; }
    print $f->start, " ", $f->end, " ", $f->each_tag_value("locus_tag"), " ";

    if ( $f->has_tag("gene") )   { print $f->each_tag_value("gene"), " "; }
    else {
if ( $f->has_tag("locus_tag") ) {  print $f->each_tag_value("locus_tag"), " "; }
else { print "not_annotated "; }
    }

    if ( $f->has_tag("product") ) { print $f->each_tag_value("product"); }
    else { print "no_product"; }

    if ( $f->primary_tag eq "CDS" ) {
```

```
print "\n", $f->each_tag_value("translation"), "\n";
    }
    else {
print "\n", $f->each_tag_value("product"), "\n";
    }
}
    }
}
```

### A.1.4 format-genes-file-for-comparison.awk

```
BEGIN {
  NUM=0;
}

{
  if ( $1 == ">" ) {
    NUM++;
    if ( $2 == "CDS" ) {
      printf("%s%s:%s:%s:%s:%s:%d\n", $1, $2, $3, $4, $5, NUMORG, NUM) >> DIR1"/BLAST-DB/"ORG"-CDS.txt";
      TYPE="CDS";
    }
    else {
      printf("%s %s %s %s %s %s %d\n", $1, $2, $3, $4, $5, NUMORG, NUM) >> DIR1"/RNA/"ORG"-"NUM"-RNA.txt";
      TYPE="RNA";
    }

    printf("%s %s %s\n", $3, $4, $5) >> DIR1"/AUX-FILES/"ORG"-LOC.txt";

    printf("%s ", ORG) >> DIR1"/"ORG"-ANNOT.txt";
    for ( i = 6; i <= NF; i++ )
      printf("%s ", $i) >> DIR1"/"ORG"-ANNOT.txt";
      printf("\n") >> DIR1"/"ORG"-ANNOT.txt";
  }
  else {
    if ( TYPE == "CDS" )
      printf("%s\n", $0) >> DIR1"/BLAST-DB/"ORG"-CDS.txt";
    else
      printf("%s\n", $0) >> DIR1"/RNA/"ORG"-"NUM"-RNA.txt";
  }
}

END {
}
```

### A.1.5 generate-input-data.sh

```
############################################################
# Generating data for gene order analysis                  #
# Version 1,                                               #
############################################################
# Cedric Chauve  (LaCIM, UQAM, April 2005)                 #
# Karine St-onge (LaCIM, UQAM, April 2005)                 #
############################################################

#!/bin/sh

############################################################
# Parameters                                               #
############################################################
ID=$1 # Identity ratio
OV=$2 # Overlap ratio

############################################################
# A. Reading arguments in a ORGS array (NBORGS genomes)    #
############################################################
declare -a ORGS

IND=1
NBORGS=0
for F in "$@"
  do
  if [ "$IND" -gt "2" ]
      then
      NBORGS=‘expr $NBORGS + 1‘
      ORGS[$NBORGS]=$F
  fi
  IND=‘expr $IND + 1‘
done
#-----------------------------------------------------------
#-----------------------------------------------------------

############################################################
# B. Creating the directories associated to the genomes    #
# and their BLAST comparison                               #
############################################################

############################################################
# 1. Creating the directory
```

```
#############################################################
IND=1
DIR1="DATA"
while [ "$IND" -le "$NBORGS" ]
  do
  DIR1=$DIR1"-"${ORGS[$IND]}
  IND=‘expr $IND + 1‘
done
#-------------------------------------------------------

#############################################################
# 2. If the directory exists, all that step is skipped
#############################################################
if [ ! -d $DIR1 ]
    then
    ########################################################
    # 1. Creating the subdirectories
    ########################################################
    mkdir $DIR1
    mkdir $DIR1"/AUX-FILES"
    mkdir $DIR1"/BLAST-DB"
    mkdir $DIR1"/RNA"

    ########################################################
    # 2. Extracting the information from the genbank files
    #    The genbank files are in DIRGENOMES
    #    The extracted information are annotations and
    #    positions for every gene, as well as AA sequences
    #    for coding genes.
    ########################################################
    DIRGENOMES=GENBANK-GENOMES
    IND=1
    while [ "$IND" -le "$NBORGS" ]
      do
      echo "--- Conversion from NCBI"
      convert-genbank-2-genes-file.perl   \
  $DIRGENOMES"/"${ORGS[$IND]}"-NCBI.txt" \
  > $DIR1"/AUX-FILES/"${ORGS[$IND]}"-genes-file.txt";
      IND=‘expr $IND + 1‘
    done

    ########################################################
    # 3. Creating the different files: annotation files
    #    in the main directory, fasta file for coding
```

```
    #    genes and individual files for every RNA
    ########################################################
    IND=1
    while [ "$IND" -le "$NBORGS" ]
      do
      echo "--- Generating CDS, RNA and annotation files"
      gawk -f format-genes-file-for-comparison.awk \
  -v DIR1=$DIR1 -v ORG=${ORGS[$IND]} -v NUMORG=$IND \
  $DIR1"/AUX-FILES/"${ORGS[$IND]}"-genes-file.txt";
      IND=‘expr $IND + 1‘
    done

    ########################################################
    # 4. Formating the Fasta files for blast queries
    ########################################################
    IND=1
    while [ "$IND" -le "$NBORGS" ]
      do
      echo "--- Formating for BLAST"
      formatdb -i $DIR1"/BLAST-DB/"${ORGS[$IND]}"-CDS.txt";
      IND=‘expr $IND + 1‘
    done

    ########################################################
    # 5. Blast queries of every genome vs all other genomes
    ########################################################
    IND1=1
    while [ "$IND1" -le "$NBORGS" ]
      do
      IND2=1
      while [ "$IND2" -le "$NBORGS" ]
do
echo "BLAST " ${ORGS[$IND1]} " vs " ${ORGS[$IND2]}
blastall -p blastp -F F -m 9                        \
    -i $DIR1"/BLAST-DB/"${ORGS[$IND1]}"-CDS.txt"   \
    -d $DIR1"/BLAST-DB/"${ORGS[$IND2]}"-CDS.txt"   \
    -o $DIR1"/AUX-FILES/"${ORGS[$IND1]}"-"${ORGS[$IND2]}"-blast.txt" \
    2> $DIR1"/AUX-FILES/errors.txt";

IND2=‘expr $IND2 + 1‘
      done
      IND1=‘expr $IND1 + 1‘
    done
```

```
    ########################################################
    # 6. Computing RNA genes families from their annotations
    ########################################################
    echo "RNA processing"
    rna-2-families-graph.sh $DIR1
fi
#--------------------------------------------------------
#--------------------------------------------------------

############################################################
# C. Creating the directories associated to values of    #
#    ID and OV of the corresponding genes families       #
############################################################

############################################################
# 1. If the directory exists, all that step is skipped   #
############################################################
DIR2=$DIR1/"FAMILIES-"$ID"-"$OV

if [ ! -d $DIR2 ]
    then
    ########################################################
    # 1. Transferring RNA families into the families file
    ########################################################
    mkdir $DIR2
    cat $DIR1"/AUX-FILES/families-graph-rna.txt" \
> $DIR2"/families-graph.txt";

    ########################################################
    # 2. Transforming the BLAST results into families
    ########################################################
    IND1=1
    while [ "$IND1" -le "$NBORGS" ]
      do
      IND2=$IND1
      while [ "$IND2" -le "$NBORGS" ]
do
echo "Transforming Blast results into the family graph " \
    ${ORGS[$IND1]} " " ${ORGS[$IND2]}
gawk -f blast-2-families-graph.awk -v \
    ID=$ID -v OV=$OV                              \
    $DIR1"/AUX-FILES/"${ORGS[$IND1]}"-"${ORGS[$IND2]}"-blast.txt"  \
    >> $DIR2"/families-graph.txt";
IND2=‘expr $IND2 + 1‘
```

```
      done
      IND1=`expr $IND1 + 1`
    done

    ######################################################
    # 3. Computing the final data
    ######################################################
    echo "Generating data files"
    ARGS="$NBORGS $DIR2"/families-graph.txt" \
    `wc -l $DIR2"/families-graph.txt" | gawk '{print $1}'` 0"

    IND1=1
    while [ "$IND1" -le "$NBORGS" ]
      do
      ARGS="$ARGS $DIR2"/"${ORGS[$IND1]}"-GENES.txt" \
      $DIR1"/AUX-FILES/"${ORGS[$IND1]}"-LOC.txt" \
      $DIR1"/"${ORGS[$IND1]}"-ANNOT.txt" \
      `wc -l $DIR1"/AUX-FILES/"${ORGS[$IND1]}"-LOC.txt" | \
      gawk '{print $1}'`"
      IND1=`expr $IND1 + 1`
    done

    blast-2-data $ARGS > $DIR2"/FAMILIES.txt"
fi
#-----------------------------------------------------------
#-----------------------------------------------------------
```

### A.1.6 rna-2-families-graph.sh

```
#!/bin/sh

cd $1"/RNA";

L1=`ls *.txt`
L2=`ls *.txt`

for F1 in $L1; do
  for F2 in $L2; do
    if [[ "$F1" > "$F2" ]]
    then
      if [ "`gawk -f ../../PROGRAMS/rna-product.awk $F1`" == "`gawk -f ../../PROGRAMS/rna-product.awk $F2`"
      then
echo `gawk -f ../../PROGRAMS/rna-position.awk $F1` `gawk -f ../../PROGRAMS/rna-position.awk $F2` >> ../AUX-F
      fi
```

```
    fi
  done
done

cd ../..
```

### A.1.7 rna-position.awk

```
BEGIN {
}

{
  if ( $1 == ">" ) { printf("%s %s ", $6, $7); }
}

END {
}
```

### A.1.8 rna-product.awk

```
BEGIN {
}

{
  if ( $1 != ">" ) { printf("%s\n", $0); }
}

END {
}
```

## A.2 Le programme

### A.2.1 compute-clusters.c

```
// ------------------------------------------------------------------------------
// Computation of gene clusters, v1.1
// Cedric Chauve and Karine St-Onge
// LaCIM, Universite du Quebec a Montreal
// May 2005.
// ------------------------------------------------------------------------------

// ******************************************************************************
```

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
// ******************************************************************************

// ******************************************************************************
// CONSTANTS
// ------------------------------------------------------------------------------

#define SEGMENT_MAX    10000 // Max number of genes in a segment
#define WEIGHT_MAX     100   // Max weight of a point
#define FAM_UNDEF      0     // Number indicating a gene in the input does not
                             // belong to any family
// ******************************************************************************

// ******************************************************************************
// INPUT
// ------------------------------------------------------------------------------

// Number of different gene families
static long NB_FAMILIES;

// Number of genomes
static int  NB_GENOMES;

// Number of genes in each genome
static long *SIZE_GENOMES;
// Size of genome _K_ (_K_ from 1 to NB_GENOMES)
#define SIZE(_K_)  SIZE_GENOMES[(_K_)-1]

// Arrays of sequences of genomes
static long **GENOMES;
// Gene in position _I_ (from 1 to SIZE_GENOMES[_K_])
// of genome _K_ (from 1 to NB_GENOMES)
#define GENE(_K_, _I_) GENOMES[(_K_)-1][(_I_)-1]

// Arrays of signs of genes in genomes
static long **SIGNS;
// Sign of gene in position (from 1 to SIZE_GENOMES[_K_])
// of genome _K_ (from 1 to NB_GENOMES)
#define SIGN(_K_, _I_) SIGNS[(_K_)-1][(_I_)-1]
// The three values for the sign of a gene
#define SIGN_UNDEF 0 // Undefined
#define SIGN_PLUS  1 // Plus
```

```
#define SIGN_MINUS 2 // Minus

// Starting positions of genes in genomes
static long **POS_START;
// Starting base of gene in position (from 1 to SIZE_GENOMES[_K_])
// of genome _K_ (from 1 to NB_GENOMES)
#define START(_K_, _I_) POS_START[(_K_)-1][(_I_)-1]

// Ending positions of genes in genomes
static long  **POS_END;
// Ending base of gene in position (from 1 to SIZE_GENOMES[_K_])
// of genome _K_ (from 1 to NB_GENOMES)
#define END(_K_, _I_) POS_END[(_K_)-1][(_I_)-1]

// Importing the data from the two files containing a genome
// Maximum length of a line containing the gene information other than annotations
#define LG_STR 50
// genes = name of the file containing the genes informations (number, sign,
//         starting base, ending base)
// annotations = name og the file containing the annotations
// k = number of the genome (from 1 to NB_GENOMES)
// nb_genes = number of genes in the current genome
static void INPUT_Read_Genome(char *genes, int k, int nb_genes) {
  FILE *F;    // File containing the sequence of genes
  int   i;    // Cursor on the lines of the file genes
  int   j, l; // Cursors on a line of the file genes
  long  x;    // The current gene
  // Temporary strings used to read the file genes
  char *gene = (char *) malloc(LG_STR * sizeof(char));
  char *info = (char *) malloc(LG_STR * sizeof(char));

  F = fopen(genes, "r");

  SIZE(k)        = nb_genes;
  GENOMES[k-1]   = (long *)  malloc (nb_genes * sizeof(long));
  SIGNS[k-1]     = (long *)  malloc (nb_genes * sizeof(long));
  POS_START[k-1] = (long *)  malloc (nb_genes * sizeof(long));
  POS_END[k-1]   = (long *)  malloc (nb_genes * sizeof(long));

  for ( i = 1; i <= nb_genes; i++ ) {
    // Reading the gene number
    gene = fgets(gene, LG_STR, F);
    j = 0; while ( gene[j] != ' ' ) { j++; }
    info = strncpy(info, gene, j); info[j] = '\0';
```

```
    GENE(k,i) = atol(info);
    // Reading the gene sign
    j++; if ( gene[j] == '-' ) { SIGN(k,i) = SIGN_MINUS; }
    else                       { SIGN(k,i) = SIGN_PLUS; }
    // Reading the starting and ending positions
    j+= 2; l = j; while ( gene[j] != ' ' ) { j++; }
    info = strncpy(info, gene, j); info[j] = '\0';
    START(k,i) = atol(info+l);
    info = strcpy(info, gene);
    END(k,i) = atol(info+j+1);
    if ( START(k,i) > END(k,i) ) {
      l = START(k,i); START(k,i) = END(k,i); END(k,i) = l;
    }
    // Reading the annotation of the current gene
    if ( GENE(k,i) > NB_FAMILIES )
      NB_FAMILIES = GENE(k,i);
  }
  free(gene); free(info);
}

// If some genes are numbered by 0, one gives to each of them a new and
// unique numbers: each of them defines a family with one member.
static void INPUT_Add_Unlabelled() {
  int k, i; // Cursors on the genes
  for ( k = 1; k <= NB_GENOMES; k++ ) {
    for ( i = 1; i <= SIZE(k); i++ ) {
      if ( GENE(k, i) == FAM_UNDEF ) { GENE(k,i) = ++NB_FAMILIES; }
    }
  }
}
// ****************************************************************************

// ****************************************************************************
// SEGMENTS OF GENES: DATA STRUCTURE AND BASIC FUNCTIONS
// ----------------------------------------------------------------------------

// Data structure for a segment, defined on a genome k, between genes in positions
// i and j. Sets of segments are recorded into linked lists.
typedef struct segment {
  int  k; // genome
  long i; // i = start, j = end
  long j; // j > i
  struct segment *next;
} *SEGMENT;
```

```
#define SEGMENT_GENOME(_S_) ((_S_)->k)
#define SEGMENT_START(_S_)  ((_S_)->i)
#define SEGMENT_END(_S_)    ((_S_)->j)
#define SEGMENT_NEXT(_S_)   ((_S_)->next)

// Allocating a DELTA-segment with parameters k, i, j
static SEGMENT SEGMENT_Allocate(int k, long i, long j) {
  SEGMENT s =
    (SEGMENT) malloc (sizeof(struct segment));
  s->k = k; s->i = i; s->j = j; s->next = NULL;
  return(s);
}
// Freeing a DELTA-segment
static void SEGMENT_Free(SEGMENT s) {
  free(s);
}
// Freeing a list of DELTA-segments that is not empty
static void SEGMENT_List_Free(SEGMENT l) {
  if ( l->next != NULL ) { SEGMENT_List_Free(l->next); }
  SEGMENT_Free(l);
}

// Comparing two segments. Order defined in terms of coordinates (k,i,j):
//  COMP_GR: segment 1 > segment 2
//  COMP_LO: segment 2 > segment 1
//  COMP_EQ: segment 2 = segment 1
#define COMP_GR 1
#define COMP_LO -1
#define COMP_EQ 0
static int SEGMENT_Compare(SEGMENT s1, SEGMENT s2) {
  if      (s1->k > s2->k) return(COMP_GR);
  else if (s1->k < s2->k) return(COMP_LO);
  else if (s1->i > s2->i) return(COMP_GR);
  else if (s1->i < s2->i) return(COMP_LO);
  else if (s1->j > s2->j) return(COMP_GR);
  else if (s1->j < s2->j) return(COMP_LO);
  else                    return(COMP_EQ);
}
// Sorting a list in increasing order. Algorithm: bubble sort.
static SEGMENT SEGMENT_Sort(SEGMENT s) {
  int     stop;                         // Indicates when stopping the sorting
  SEGMENT head, prev1, prev2, current; // Cursors on the segments
  head = s; stop = 0; while ( stop == 0 ) {
```

```
    stop = 1; prev1 = NULL; prev2 = head; current = head->next;
    while ( current != NULL ) {
      if ( SEGMENT_Compare(prev2, current) == COMP_GR ) {
stop          = 0;
prev2->next   = current->next;
current->next = prev2;
if ( prev1 != NULL ) { prev1->next = current; }
else                 { head = current; }
prev1 = current; current = prev2->next;
      }
      else { prev1 = prev2; prev2 = current; current = current->next; }
    }
  }
  return(head);
}

// ****************************************************************************


// ****************************************************************************
// GENE FAMILIES AND GENE TEAMS: DATA STRUCTURE AND BASIC FUNCTIONS
// ----------------------------------------------------------------------------
// IMPORTANT. A gene team is encoded by a list of gene families.

typedef struct family {
  long label;          // Label of the gene family
  // The next fields, the number of occurrences of this family in a cluster,
  // and the number of genomes of a clusters where this family appears
  // are used when a team is associated to a cluster of segments
  int  nb_occurrences;
  int  nb_genomes;
  int  last_genome; // Last genome of a cluster where this family was seen
  struct family *next;
} *FAMILY;

#define FAMILY_LABEL(_F_)  ((_F_)->label)
#define FAMILY_NB_OCC(_F_) ((_F_)->nb_occurrences)
#define FAMILY_NB_GEN(_F_) ((_F_)->nb_genomes)
#define FAMILY_LAST(_F_)   ((_F_)->last_genome)
#define TEAM_NEXT_FAM(_F_) ((_F_)->next)

// Allocation of a family
static FAMILY FAMILY_Allocate(long label, int nb_occurrences, int last_genome) {
  FAMILY f = (FAMILY) malloc(sizeof (struct family));
```

```
  f->label = label;
  f->nb_occurrences = nb_occurrences;
  f->nb_genomes = 1;
  f->last_genome = last_genome;
  f->next = NULL;
  return(f);
}
// Freeing a family
static void FAMILY_Free(FAMILY f) {
  free(f);
}
// Freeing a team
static void TEAM_Free(FAMILY t) {
  if ( t->next != NULL ) TEAM_Free(t->next);
  FAMILY_Free(t);
}

// Sorting a team by decreasing order of occurrences number
// Algorithm: bubble sort.
static FAMILY TEAM_Sort(FAMILY f) {
  int    stop;                        // Indicates when stopping the sorting
  FAMILY head, prev1, prev2, current; // Cursors on the team
  stop = 0; head = f; while ( stop == 0 ) {
    stop = 1; prev1 = NULL; prev2 = head; current = head->next;
    while ( current != NULL ) {
      if ( current->nb_occurrences > prev2->nb_occurrences ) {
stop = 0;
prev2->next   = current->next;
current->next = prev2;
if ( prev1 != NULL ) { prev1->next = current; }
else                 { head = current; }
prev1 = current; current = prev2->next;
      }
      else { prev1 = prev2; prev2 = current; current = current->next; }
    }
  }
  return(head);
}
// ****************************************************************************


// ****************************************************************************
// SEGMENTATION OF THE GENOMES
// ----------------------------------------------------------------------------
```

```
// The segmentation phase consists in computing a list of genomic segments that
// will be used to give a weight to every point between consecutive genes of all
// genomes
// ------------------------------------------------------------------------------

// ------------------------------------------------------------------------------
// USER DEFINED PARAMETER.
static int DELTA;
// A DELTA-segment is a segment of at most DELTA+1 genes.
// It is said to be a conserved DELTA-segment if either its two extremities x and
// y are equal or there is another occurrence of the genes x and y, separated by
// at most DELTA-1 genes.

// Weights of points between consecutive genes.
static int **WEIGHTS;
// The weight of a point is the number of conserved DELTA-segments that contain
// both genes that define this point.
// Weight of the point between genes _I_ and _I_+1 of genome _K_
#define WEIGHT(_K_, _I_) WEIGHTS[(_K_)-1][(_I_)-1]

// ------------------------------------------------------------------------------
// Data structure used to keep track of the neighboring relations encoutered
// between genes when examining all possible DELTA-segments.
// Each record of this linked list (x=(k,i),y=(k,j)) indicates that this couple
// of genes has been encountered in a same DELTA-segment either ONE_TIME or
// SEVERAL_TIME.
typedef struct SEGMENTATION_pair {
  int  k; // Sequence containing the first occurrence of (x,y)
  long i; // Position of x in k in this first occurrence
  long j; // Position of y in k in this first occurrence
  int  v; // Status of the neighboring with (x,y): ONE_TIME or SEVERAL_TIME
  struct SEGMENTATION_pair *next;
} *SEGMENTATION_Pair;
// The two values to indicate the status of a couple of genes (x,y) :
#define ONE_TIME 1 // Encountered one time
#define SEVERAL  2 // Encountered more than one time

// Allocating a new pair with parameters k, i, j.
static SEGMENTATION_Pair SEGMENTATION_Pair_Allocate(int k, long i, long j) {
  SEGMENTATION_Pair p =
    (SEGMENTATION_Pair) malloc(sizeof(struct SEGMENTATION_pair));
  p->k = k; p->i = i; p->j = j; p->v = ONE_TIME; p->next = NULL;
  return(p);
}
```

```
// Freeing a pair
static void SEGMENTATION_Pair_Free(SEGMENTATION_Pair p) {
  free(p);
}
// Freeing a linked list of pair that is not empty
static void SEGMENTATION_Pair_List_Free(SEGMENTATION_Pair l) {
  if ( l->next != NULL ) { SEGMENTATION_Pair_List_Free(l->next); }
  SEGMENTATION_Pair_Free(l);
}

// The main data structure to keep track of these couples (x,y) is an array of
// NB_FAMILIES linked lists of SEGMENTATION_Pairs.
// (1) One supposes that |x| is greater than every cell in the corresponding
// list LINKS[|x|], to avoid the duplication of information.
static SEGMENTATION_Pair *SEGMENTATION_PAIRS_LIST;

// This function adds a pair indicating the first occurrence of a pair (x,y),
// in the list SEGMENTATION_PAIRS_LIST[x-1]where x is the smallest gene among
// the ones in positions i and j of file k, and y the other gene.
static void SEGMENTATION_Add_Pair(int k, long i, long j) {
  SEGMENTATION_Pair c;
  if ( GENE(k, i) > GENE(k, j) ) {
    c = SEGMENTATION_Pair_Allocate(k, i, j);
    c->next = SEGMENTATION_PAIRS_LIST[GENE(k, i)-1];
    SEGMENTATION_PAIRS_LIST[GENE(k, i)-1] = c;
  }
  else {
    c = SEGMENTATION_Pair_Allocate(k, j, i);
    c->next = SEGMENTATION_PAIRS_LIST[GENE(k, j)-1];
    SEGMENTATION_PAIRS_LIST[GENE(k, j)-1] = c;
  }
}

// This function checks if the couple of genes (x,y) located in positions i and j
// of sequence k has already been encountered separated by a distance at most
// DELTA and then is in the list SEGMENTATION_PAIRS_LIST.
// It returns a pointer to the cell of SEGMENTATION_PAIRS_LIST corresponding to
// the first occurrence of this pair if there was one, and NULL is there was none.
static SEGMENTATION_Pair SEGMENTATION_Check_Presence(int k, long i, long j) {
  SEGMENTATION_Pair c; // Current cell of the list SEGMENTATION_PAIRS_LIST[x-1]
  long X, Y, I, J, T;  // Local copies of x, i, y, j
  // First, one makes sure that X > Y (see comment (1) above)
  X = GENE(k, i); I = i; Y = GENE(k, j); J = j;
  if ( X < Y ) { T = X; X = Y; Y = T; T = I; I = J; J = T; }
```

```
  // Next, one looks for an occurrence of the couple (X,Y)
  c = SEGMENTATION_PAIRS_LIST[X-1];
  while ( (c != NULL) && (GENE(c->k, c->j) != Y) ) { c = c->next; }
  return (c);
}

// ------------------------------------------------------------------------------
// The complete list of all conserved DELTA-segments: a list of segments.
static SEGMENT SEGMENTATION_SEGMENTS_LIST;

// Adding a DELTA-segment at the beginning of the list of segments
static SEGMENT SEGMENTATION_Add_Segment(int k, long i, long j) {
  SEGMENT s                  = SEGMENT_Allocate(k, i, j);
  s->next                    = SEGMENTATION_SEGMENTS_LIST;
  SEGMENTATION_SEGMENTS_LIST = s;
}

// Updating the list SEGMENTATION_SEGMENTS_LIST of DELTA-segments by examining
// the segment of genome k that ends at position i and starts at position j.
static void SEGMENTATION_Process_Segment(int k, long i, long j) {
  long x = GENE(k, i);
  long y = GENE(k, j);
  SEGMENTATION_Pair c;  // Initial segment (x,y) if there is one
  // First case, y = x: tandem segment
  if ( y == x ) { SEGMENTATION_Add_Segment(k, j, i); }
  else { // Duplicated segments
    c = SEGMENTATION_Check_Presence(k, i, j);
    // Second case: (x,y) never appeared: one records its initial apparition
    if ( c == NULL ) { SEGMENTATION_Add_Pair(k, i, j); }
    // Third case: (x,y) appeared one time: one adds the initial segment
    else if ( c->v == ONE_TIME ) {
      c->v = SEVERAL;
      if ( c->i > c->j ) { SEGMENTATION_Add_Segment(c->k, c->j, c->i); }
      else               { SEGMENTATION_Add_Segment(c->k, c->i, c->j); }
      SEGMENTATION_Add_Segment(k, j, i);
    }
    // Fourth case: (x,y) appeared several times:
    // the initial segment is already in the list
    else { SEGMENTATION_Add_Segment(k, j, i); }
  }
}

// Processing the list SEGMENTATION_SEGMENTS_LIST of all conserved
// DELTA-segments to weight every point of the genomes.
```

```
static void SEGMENTATION_Compute_Weights() {
  int     k;    // Cursor on all genomes
  long    i, j; // Cursors on the genes of a genome
  SEGMENT s;    // Cursor on the list of conserved DELTA-segments
  SEGMENTATION_PAIRS_LIST =
    (SEGMENTATION_Pair *) calloc(NB_FAMILIES, sizeof(SEGMENTATION_Pair));
  // Computation of conserved DELTA-segments
  for ( k = 1; k <= NB_GENOMES; k++ ) {
    for ( i = 1; i <= SIZE(k); i++ ) {
      for ( j = i-1; (j >0) && (j >= i-DELTA); j-- ) {
SEGMENTATION_Process_Segment(k, i, j);
      }
    }
  }
  // Computation of the weights
  WEIGHTS = (int **)  malloc(NB_GENOMES * sizeof(int *));
  for ( k = 1; k <= NB_GENOMES; k++ ) {
    WEIGHTS[k-1] = (int *) calloc(SIZE(k), sizeof(int));
  }
  s = SEGMENTATION_SEGMENTS_LIST; while ( s != NULL ) {
    int i; // Cursor on the genes of the current segment
    for ( i = s->i; i < s->j; i++ ) { WEIGHT(s->k, i)++; }
    s = s->next;
  }
  SEGMENT_List_Free(SEGMENTATION_SEGMENTS_LIST);
}

// ---------------------------------------------------------------------------
// Computing and displaying statistics about a segmentation phase

// Number of segments of a given size: if some segments are longer than
// SEGMENT_MAX, they are counted as segments of size SEGMENT_MAX
static int SEGMENTATION_NB_SEGMENTS[SEGMENT_MAX];
// Number of points of a given weight
static int SEGMENTATION_NB_WEIGHTS[WEIGHT_MAX];

// k = number of the processed genome (from 1 to NB_GENOMES)
// w = minimum weight for two consecutive genes to be in the same segment (>= 0)
static void SEGMENTATION_Compute_Statistics(int k, int w) {
  int i, j, c; // Various cursors
  if ( k == 1 ) {
    for ( c = 0; c < SEGMENT_MAX; c++ ) { SEGMENTATION_NB_SEGMENTS[c] = 0; }
    for ( c = 0; c < WEIGHT_MAX; c++ )  { SEGMENTATION_NB_WEIGHTS[c] = 0; }
  }
```

```
  // Processing the genome k.
  for ( i = 1; i < SIZE(k); i++ ) { SEGMENTATION_NB_WEIGHTS[WEIGHT(k, i)]++; }
  i = 1; while ( i <= SIZE(k) ) {
    j = i; while ( (j < SIZE(k)) && (WEIGHT(k, j) >= w) ) { j++; }
    if ( j-i >= 1 ) {
      // Case of segments that are longer than the max considered size
      if ( j-i+1 > SEGMENT_MAX+2 ) SEGMENTATION_NB_SEGMENTS[SEGMENT_MAX-1]++;
      else                         SEGMENTATION_NB_SEGMENTS[j-i-1]++;
    }
    i = j+1;
  }
}

// k = number of the processed genome (from 1 to NB_GENOMES)
static void SEGMENTATION_Display_Statistics(FILE *F) {
  int c, nb, line;
  int max_line = 10; // Max number of segments info by output line
  fprintf(F, "[LOG] Segmentation of all genomes");
  nb = 0; line = max_line;
  for ( c = 0; c < SEGMENT_MAX; c++ ) {
    if ( SEGMENTATION_NB_SEGMENTS[c] > 0 ) {
      if ( line ==  max_line ) { fprintf(F, "\n[LOG]   "); line = 0; }
      fprintf(F, "[%d,%d] ", c+1, SEGMENTATION_NB_SEGMENTS[c]); line++;
    }
    nb += SEGMENTATION_NB_SEGMENTS[c] * (c+1);
  }
  fprintf(F, "\n[LOG]   Weights distribution \n[LOG]   ");
  for ( c = 0; c < WEIGHT_MAX; c++ ) {
    if ( SEGMENTATION_NB_WEIGHTS[c] > 0 ) {
      fprintf(F, "[%d,%d] ", c, SEGMENTATION_NB_WEIGHTS[c]);
    }
  }
  fprintf(F, "\n");
}
// ****************************************************************************


// ****************************************************************************
// COMPUTATION OF THE SEGMENTS SIMILARITY GRAPH
// ----------------------------------------------------------------------------
// Computing a similarity graph whose vertices are the conserved DELTA-segments
// of the list SEGMENTATION_SEGMENTS_LIST and edges indicate a similarity between
// two segments.
// ----------------------------------------------------------------------------
```

```
// ------------------------------------------------------------------------
// Through all this section, a segment is defined as a consecutive set of genes
// separated by points of weight at least OMEGA.
// THIS GLOBAL VARIABLE HAS BEEN INITIALIZED AND WILL NOT BE IN THIS SECTION.
static int OMEGA;

// ------------------------------------------------------------------------
// Array of size NB_FAMILIES used to count the genes present in a pair of
// segments.
// THIS GLOBAL VARIABLE HAS BEEN ALLOCATED AND WILL NOT BE IN THIS SECTION.
static double *CLUSTERING_OCCURRENCES;

// Number specific to a pair of segments and increase used to define a new pair.
static double CLUSTERING_RANK;
static double CLUSTERING_RANK_INCREASE = 0.00001;

// Ratio of genes of each segment that have to be common to define an edge.
// THIS GLOBAL VARIABLE HAS BEEN INITIALIZED AND WILL NOT BE IN THIS SECTION.
static double CLUSTERING_RATIO;
// Minimum number of genes that have to be common  to define an edge.
// THIS GLOBAL VARIABLE HAS BEEN INITIALIZED AND WILL NOT BE IN THIS SECTION.
static int CLUSTERING_MIN_COMMON;

// ------------------------------------------------------------------------
// Function that computes the number of common genes between the segment of genome
// k1 starting at position i1 and the segment of i2 starting at position i2.
// n1 contains the number of elements of the first segment that appears in the
// second and n2 ... vice-versa ...
// j1 (resp. j2) is the last gene of the segment 1 (resp. 2).
// s1 and e1: part of segment 1 having gene in common with segment 2
static void CLUSTERING_Compare_Segments(int   k1, long  i1,
int   k2, long  i2,
int  *n1, int  *n2,
long *j1, long *j2,
long *s1, long *e1,
long *s2, long *e2) {
  long c1, c2; // Cursors on the two segments
  CLUSTERING_RANK += CLUSTERING_RANK_INCREASE;
  *n1 = *n2 = 0; *s1 = *s2 = *e1 = *e2 = -1;
  // Recording genes of segment 1 and the end j1 of this segment
  c1 = i1; while ( (c1 < SIZE(k1)) && (WEIGHT(k1,c1) >= OMEGA) ) {
    CLUSTERING_OCCURRENCES[GENE(k1,c1++)-1] = CLUSTERING_RANK;
  }
```

```
  CLUSTERING_OCCURRENCES[GENE(k1,c1)-1] = CLUSTERING_RANK;
  *j1 = c1;
  // Recording genes common to segments 1 and 2 and the end j2 of segment 2
  c2 = i2; while ( (c2 < SIZE(k2)) && (WEIGHT(k2,c2) >= OMEGA) ) {
    if ( CLUSTERING_OCCURRENCES[GENE(k2,c2++)-1] == CLUSTERING_RANK ) {
      CLUSTERING_OCCURRENCES[GENE(k2,c2-1)-1] = -CLUSTERING_RANK;
    }
  }
  if ( CLUSTERING_OCCURRENCES[GENE(k2,c2)-1] == CLUSTERING_RANK ) {
    CLUSTERING_OCCURRENCES[GENE(k2,c2)-1] = -CLUSTERING_RANK;
  }
  *j2 = c2;
  // Computing n1, s1 and e1
  for ( c1 = i1; c1 <= *j1; c1++ ) {
    if ( CLUSTERING_OCCURRENCES[GENE(k1,c1)-1] == -CLUSTERING_RANK ) {
      if ( *s1 == -1 ) { *s1 = c1; } *e1 = c1; (*n1)++;
    }
  }
  // Computing n2, s2 and e2
  for ( c2 = i2; c2 <= *j2; c2++ ) {
    if ( CLUSTERING_OCCURRENCES[GENE(k2,c2)-1] == -CLUSTERING_RANK ) {
      if ( *s2 == -1 ) { *s2 = c2; } *e2 = c2; (*n2)++;
    }
  }
}

// Computing the similarity (1 for yes, and 0 for no) between two segments
// (k1,i1,j1) and (k2,i2,j2) given the characteristics of their intersection
// (n1, e1, s1, n2, e2, s2)
static int CLUSTERING_Compute_Similarity(int k1,  long i1, long j1, int n1,
 int k2,  long i2, long j2, int n2,
 long s1, long e1, long s2, long e2) {
  if ( (n1 >= CLUSTERING_MIN_COMMON) && (n2 >= CLUSTERING_MIN_COMMON)  &&
       (((double ) n1) >= (abs(j1-i1)+1)*CLUSTERING_RATIO) &&
       (((double ) n2) >= (abs(j2-i2)+1)*CLUSTERING_RATIO) )
    return(1);
  else
    return(0);
}

// ------------------------------------------------------------------------------
// Data structure for a list of edges of the similarity graph.
// An edge is labelled with the characteristics of the similarity between the two
// segments that define this edge: see parameters of CLUSTERING_Compare_Segments
```

```
// for the comment of each field.

// Note: the order on segments used is the one defined in SEGMENT_Compare.

typedef struct CLUSTERING_edge {
  int  k1; long i1, j1; int  n1; long s1, e1;
  int  k2; long i2, j2; int  n2; long s2, e2;
  struct CLUSTERING_edge *next;
} *CLUSTERING_Edge;
// Allocating an edge
static CLUSTERING_Edge CLUSTERING_Edge_Allocate(int  k1, long i1, long j1,
int  n1, long s1, long e1,
int  k2, long i2, long j2,
int  n2, long s2, long e2 ) {
  CLUSTERING_Edge e = (CLUSTERING_Edge) malloc(sizeof(struct CLUSTERING_edge));
  e->k1 = k1; e->i1 = i1; e->j1 = j1; e->n1 = n1; e->s1 = s1; e->e1 = e1;
  e->k2 = k2; e->i2 = i2; e->j2 = j2; e->n2 = n2; e->s2 = s2; e->e2 = e2;
  e->next = NULL;
  return(e);
}
// Freeing an edge
static void CLUSTERING_Edge_Free(CLUSTERING_Edge e) {
  free(e);
}
// Freeing a non-empty list of edges
static void CLUSTERING_Edge_List_Free(CLUSTERING_Edge l) {
  if ( l->next != NULL ) { CLUSTERING_Edge_List_Free(l->next); }
  CLUSTERING_Edge_Free(l);
}

// ------------------------------------------------------------------------------
// Array of the neighbors in the intersection graph: indexed by genomes and genes
static CLUSTERING_Edge **SIMILARITY_GRAPH;

#define CLUSTERING_GRAPH_NEIGHBORS(_G_,_K_,_I_) ((_G_)[(_K_)-1][(_I_)-1])

// Initializations of the similarity graph
static void CLUSTERING_Graph_Allocate() {
  int  k;
  long i;
  SIMILARITY_GRAPH =
    (CLUSTERING_Edge **) malloc(NB_GENOMES * sizeof(CLUSTERING_Edge *));
  for ( k = 1; k <= NB_GENOMES; k++ ) {
    SIMILARITY_GRAPH[k-1] =
```

```
      (CLUSTERING_Edge *) calloc(SIZE(k), sizeof(CLUSTERING_Edge));
  }
}
static void CLUSTERING_Graph_Free() {
  int  k;
  long i;
  for ( k = 1; k <= NB_GENOMES; k++ ) {
    for ( i = 1; i <= SIZE(k); i++ ) {
      if ( SIMILARITY_GRAPH[k-1][i-1] != NULL ) {
CLUSTERING_Edge_List_Free(SIMILARITY_GRAPH[k-1][i-1]);
      }
    }
    free(SIMILARITY_GRAPH[k-1]);
  }
  free(SIMILARITY_GRAPH);
}

// ------------------------------------------------------------------------------
// Computing the similarity between segments (k1,i1) and (k2,i2), that is
// segments starting in positions (k1,i1) and (k2,i2) -- the end of these
// two segments is implicitely given by the golabl variable OMEGA
// --, with creation of an edge if these two segments are considered as similar.
static void CLUSTERING_Process_Segments(int k1, long i1, int k2, long i2) {
  int  n1, n2;
  long j1, j2;
  long e1, e2, s1, s2;
  // Checking that i1 is the beginning of a segment
  if ( (WEIGHT(k1, i1) < OMEGA) || ((i1 > 1) && (WEIGHT(k1, i1-1) >= OMEGA)) ) {
    return;
  }
  // The same for the second segment
  if ( (WEIGHT(k2, i2) < OMEGA) || ((i2 > 1) && (WEIGHT(k2, i2-1) >= OMEGA)) ) {
    return;
  }
  // Comparing the two segments
  CLUSTERING_Compare_Segments(k1, i1, k2, i2, &n1, &n2, &j1, &j2, &s1, &e1, &s2, &e2);
  if ( CLUSTERING_Compute_Similarity(k1, i1, j1, n1, k2, i2, j2, n2, s1, e1, s2, e2) ) {
    CLUSTERING_Edge e;
    e       = CLUSTERING_Edge_Allocate(k1, i1, j1, n1, s1, e1, k2, i2, j2, n2, s2, e2);
    e->next = SIMILARITY_GRAPH[k1-1][i1-1];
    SIMILARITY_GRAPH[k1-1][i1-1] = e;
    e       = CLUSTERING_Edge_Allocate(k2, i2, j2, n2, s2, e2, k1, i1, j1, n1, s1, e1);
    e->next = SIMILARITY_GRAPH[k2-1][i2-1];
    SIMILARITY_GRAPH[k2-1][i2-1] = e;
```

```
  }
}

// ------------------------------------------------------------------------------
// Computing all edges of the intersection graph in the variable SIMILARITY_GRAPH

static void CLUSTERING_Compute_Edges() {
  int k, i, k1, k2, i1, i2;
  // Initialization. Data structures for the comparison of segments
  CLUSTERING_RANK        = 0.00000;
  CLUSTERING_OCCURRENCES = (double *) calloc (NB_FAMILIES, sizeof(double));
  // Computing the edges
  for ( k1 = 1; k1 <= NB_GENOMES; k1++ ) {
    for ( i1 = 1; i1 <= SIZE(k1); i1++ ) {
      for ( i2 = i1+1; i2 <= SIZE(k1); i2++ ) {
CLUSTERING_Process_Segments(k1, i1, k1, i2);
      }
      for ( k2 = k1+1; k2 <= NB_GENOMES; k2++ ) {
for ( i2 = 1; i2 <= SIZE(k2); i2++ ) {
  CLUSTERING_Process_Segments(k1, i1, k2, i2);
}
      }
    }
  }
  free(CLUSTERING_OCCURRENCES);
}
// ******************************************************************************

// ******************************************************************************
// CLUSTERS: DATA STRUCTURE AND BASIC FUNCTIONS
// ------------------------------------------------------------------------------

// ------------------------------------------------------------------------------
// List of clusters.
typedef struct CLUSTERING_list {
  SEGMENT segments;    // List of segments of a cluster
  double  ratio;       // Ratio used for this cluster
  int     weight;      // Weight used for this cluster
  int     duplicated;  // Indicate if this cluster already exists (1 = yes, 0 = no)
  int     nb_genomes;  // Number of genomes covered by the cluster
  int     nb_segments; // Number of segments in the cluster
  int     nb_families; // Number of gene families the cluster
  int     nb_genes;    // Number of genes the cluster
  FAMILY  families;    // Families present in the cluster
```

```
  struct CLUSTERING_list *next; // Next cluster
} *CLUSTERING_List;

// Allocation of a cluster
static CLUSTERING_List CLUSTERING_List_Allocate(double ratio, int weight) {
  CLUSTERING_List c = (CLUSTERING_List) malloc(sizeof(struct CLUSTERING_list));
  c->ratio      = ratio;
  c->weight     = weight;
  c->duplicated = 0;
  c->segments = NULL;
  c->families = NULL;
  c->next     = NULL;
  c->nb_genomes = c->nb_segments = c->nb_families = c->nb_genes = 0;
  return(c);
}
// Freeing a cluster
static void CLUSTERING_Free(CLUSTERING_List c) {
  if ( c->segments != NULL ) { SEGMENT_Free(c->segments); }
  if ( c->families != NULL ) { TEAM_Free(c->families);    }
  free(c);
}
// Freeing a list of clusters
static void CLUSTERING_List_Free(CLUSTERING_List c) {
  if ( c->next != NULL ) { CLUSTERING_List_Free(c->next); }
  CLUSTERING_Free(c);
}

// The first list of clusters: this variable contains all the computed clusters
// induced by the similarity graph
static CLUSTERING_List CLUSTERS_GRAPH;
// The second list of clusters: this variable contains all the computed clusters
// induced by the teams
static CLUSTERING_List CLUSTERS_TEAMS;
// ****************************************************************************


// ****************************************************************************
// COMPUTATION OF THE SET OF GENE CLUSTERS BY A DFS OF THE SIMILARITY GRAPH
// ----------------------------------------------------------------------------
// Each connected component of the similarity graph defines a cluster.

// ----------------------------------------------------------------------------
// Recursive function for the DFS of the graph of intersection of segments
```

```
// Adding the segment (i, j) of genome k to the current cluster,
// as the first segment
static void CLUSTERING_Add_Segment(CLUSTERING_List c, int k, long i, long j) {
  SEGMENT s       = SEGMENT_Allocate(k, i, j);
  SEGMENT_NEXT(s) = c->segments;
  c->segments     = s;
  c->nb_segments++; c->nb_genes += j-i+1;
}

// Data structure to keep track of the visit of segments of a cluster
// CLUSTERING_VISITED[k-1][i-1] != OMEGA IFF either
//  the position i of genome k is not the begining of a segment
//  the position i of genome k begins a segment that has to be added to a cluster
static int **CLUSTERING_VISITED;

static void CLUSTERING_DFS(int k, long i) {
  CLUSTERING_Edge c;
  c = CLUSTERING_GRAPH_NEIGHBORS(SIMILARITY_GRAPH, k, i);
  while ( c != NULL ) {
    if ( CLUSTERING_VISITED[c->k2-1][c->i2-1] != OMEGA ) {
      CLUSTERING_VISITED[c->k2-1][c->i2-1] = OMEGA;
      CLUSTERING_Add_Segment(CLUSTERS_GRAPH, c->k2,  c->i2,  c->j2);
      CLUSTERING_DFS(c->k2, c->i2);
    }
    c = c->next;
  }
}
static void CLUSTERING_Compute_Clusters_DFS() {
  int  k; // Cursor on the different genomes
  long i; // Cursor on the genes of a genome

  CLUSTERING_VISITED  = (int **) malloc(NB_GENOMES * sizeof(int *));
  for ( k = 1; k <= NB_GENOMES; k++ ) {
    CLUSTERING_VISITED[k-1]  = (int *) calloc(SIZE(k), sizeof(int));
    for ( i = 1; i <= SIZE(k); i++ ) { CLUSTERING_VISITED[k-1][i-1] = 0; }
  }
  for ( k = 1; k <= NB_GENOMES; k++ ) {
    for ( i = 1; i <= SIZE(k); i++ ) {
      CLUSTERING_Edge e = CLUSTERING_GRAPH_NEIGHBORS(SIMILARITY_GRAPH, k, i);
      // Neighbors of the segment in position i of genome k: NULL if none or
      // this position does not start a segment
      if ( CLUSTERING_VISITED[k-1][i-1] != OMEGA && e != NULL ) {
// Unvisited segment that should belong to a cluster: it starts a new
// cluster recorded into the global variable CURRENT
```

```
// Creation of the new cluster and insertion at the begining of the
// list of clusters that is given by CURRENT
CLUSTERING_List c;
c        = CLUSTERING_List_Allocate(CLUSTERING_RATIO, OMEGA);
c->next  = CLUSTERS_GRAPH;
CLUSTERS_GRAPH = c;
// Insertion of the segment (k,i) into the current cluster
CLUSTERING_Add_Segment(CLUSTERS_GRAPH, e->k1, e->i1, e->j1);
// One starts the DFS from the segment (k,i)
CLUSTERING_VISITED[k-1][i-1] = OMEGA;
CLUSTERING_DFS(k, i);
      }
    }
  }
  for ( k = 1; k <= NB_GENOMES; k++ ) { free(CLUSTERING_VISITED[k-1]); }
  free(CLUSTERING_VISITED);
}
// ****************************************************************************

// ****************************************************************************
// POSTPROCESSING OF CLUSTERS 1: SORTING
// ----------------------------------------------------------------------------
// Organizing a list of clusters: sorting and removing redundancies.

// Comparing two clusters. Result:
//  COMP_GR: cluster 2 smaller than cluster 1
//  COMP_LO: cluster 1 smaller than cluster 2
//  COMP_EQ: cluster 1 equal to cluster 2: never happens
// Order used: based on the combinatorial structure of segments
static int CLUSTERING_Compare_Clusters(CLUSTERING_List l1, CLUSTERING_List l2) {
  SEGMENT current1, current2; // Cursors on the segemnts of the clusters
  int comp;                   // Result of the comparison of two segments
  // First criterion: number of genomes where the clusters occurs
  if ( l1->nb_genomes < l2->nb_genomes )      { return(COMP_LO); }
  else if ( l1->nb_genomes > l2->nb_genomes ) { return(COMP_GR);}
  // Second criterion: number of segments
  if ( l1->nb_segments < l2->nb_segments )      { return(COMP_LO); }
  else if ( l1->nb_segments > l2->nb_segments ) { return(COMP_GR);}
  // Third criterion: combinatorial structure of segments in clusters
  current1 = l1->segments; current2 = l2->segments;
  while ( (current1 != NULL) && (current2 != NULL) &&
  ((comp = SEGMENT_Compare(current1, current2)) == COMP_EQ) ) {
    current1 = SEGMENT_NEXT(current1); current2 = SEGMENT_NEXT(current2);
  }
```

```
  // Fourth criterion: weight
  if ( comp == COMP_EQ ) {
    if ( l1->weight > l2->weight ) { comp = COMP_GR; l2->duplicated = 1; }
    else if ( l1->weight < l2->weight ) { comp = COMP_LO; l1->duplicated = 1; }
  }
  // Fifth criterion: ratio
  if ( comp == COMP_EQ ) {
    if ( l1->ratio > l2->ratio ) { comp = COMP_GR; l2->duplicated = 1; }
    else if ( l1->ratio < l2->ratio ) { comp = COMP_LO; l1->duplicated = 1; }
  }
  return(comp);
}

// Sorting a list of clusters and removing the repeated clusters.
// The clusters are sorted increasingly according to the order defined in
// CLUSTERING_Compare_Clusters.
// Algorithm: bubble sort.
static CLUSTERING_List CLUSTERS_Sort(CLUSTERING_List c) {
  int          stop; // Used to stop the sorting when clusters are sorted
  int          comp; // Result of the comparison of two segments
  CLUSTERING_List head, prev1, prev2, current; // Cursors on segments of segments
  // "Normalization" of the clusters by sorting their segments
  prev1 = c; while ( prev1 != NULL ) {
    prev1->segments = SEGMENT_Sort(prev1->segments); prev1 = prev1->next;
  }
  // Main bubble sort loop
  stop = 0; head = c; while ( stop == 0 ) {
    stop = 1; prev1 = NULL; prev2 = head; current = head->next;
    while ( current != NULL ) {
      comp = CLUSTERING_Compare_Clusters(prev2, current);
      if ( comp == COMP_LO ) { // prev2 is "greater" than current: they are swaped
stop = 0;
prev2->next = current->next; current->next = prev2;
if ( prev1 != NULL ) { prev1->next = current; }
else                 { head = current; }
prev1 = current; current = prev2->next;
      }
      else { prev1 = prev2; prev2 = current; current = current->next; }
    }
  }
  return(head);
}
// ****************************************************************************
```

```
// ******************************************************************************
// POSTPROCESSING OF CLUSTERS 2: COMPUTING STATISTICS FOR A CLUSTER
// ------------------------------------------------------------------------------
// These statistics are the number of segments and genes (that were already
// computed in CLUSTER_Add_Segment), the number of genomes and the number of
// genes. Moreover for each gene family, the number of occurrences is recorded.

// Adding a family in the list of families of a cluster.
// Families are inserted in increasing order of their label.
// (k,i): position of the gene defining the family
static void STATISTICS_Family_Add(CLUSTERING_List c, int k, long i) {
  FAMILY f, prev; // Cursors on the list of gene families of c
  if ( c->families == NULL ) { // First family encountered
    c->families = FAMILY_Allocate(GENE(k,i), 1, k);
    TEAM_NEXT_FAM(c->families) = NULL;
    c->nb_families = 1;
  }
  else {
    // Looking for the position where should be the current family (k,i)
    f = c->families; prev = NULL;
    while ( f != NULL && FAMILY_LABEL(f) < GENE(k,i) ) {
      prev = f; f = TEAM_NEXT_FAM(f);
    }
    // The family (k,i) is already recorded
    if ( f != NULL && f->label == GENE(k,i) ) {
      FAMILY_NB_OCC(f) += 1;
      if ( FAMILY_LAST(f) != k ) {
FAMILY_NB_GEN(f) += 1; FAMILY_LAST(f) = k;
      }
    }
    // The family (k,i) is not recorded
    else {
      FAMILY x = FAMILY_Allocate(GENE(k,i), 1, k);
      TEAM_NEXT_FAM(x) = f;
      c->nb_families++;
      // It should be recorded as the first family
      if ( f == c->families ) { c->families = x; }
      // It should not be recorded as the first family
      else { TEAM_NEXT_FAM(prev) = x; }
    }
  }
}
```

```
// Array used to record the genomes where a cluster is present (0 means NO)
static int *STATISTICS_GENOMES;
// Computing the statistics of a single cluster
static void STATISTICS_Cluster(CLUSTERING_List c) {
  int     k, i; // Cursors on genes of a cluster
  SEGMENT s;    // Cursors on the segments of a cluster
  for ( k = 1; k <= NB_GENOMES; k++ ) { STATISTICS_GENOMES[k-1] = 0; }
  s = c->segments; while ( s != NULL ) {
    STATISTICS_GENOMES[s->k-1]++;
    for ( i = s->i; i <= s->j; i++ ) { STATISTICS_Family_Add(c, s->k, i); }
    s = SEGMENT_NEXT(s);
  }
  for ( k = 1; k <= NB_GENOMES; k++ ) {
    if ( STATISTICS_GENOMES[k-1] > 0 ) c->nb_genomes++;
  }
}

// Computing the statistics of all clusters
static void STATISTICS_List_Clusters(CLUSTERING_List CLUSTERS) {
  CLUSTERING_List c;
  STATISTICS_GENOMES = (int *) malloc(NB_GENOMES * sizeof(int));
  c = CLUSTERS; while ( c != NULL ) { STATISTICS_Cluster(c); c = c->next; }
  free(STATISTICS_GENOMES);
}
// ****************************************************************************


// ****************************************************************************
// OUTPUT OF CLUSTERS.
// ----------------------------------------------------------------------------

// ----------------------------------------------------------------------------
// Compact output of all clusters
// USER DEFINED PARAMETER. COMPACT OUTPUT FILE.
static FILE *OUTPUT;

static void DISPLAY_List_Clusters(CLUSTERING_List CLUSTERS) {
  CLUSTERING_List c;
  SEGMENT         s;
  long            nb_clusters = 0;
  c = CLUSTERS; while ( c != NULL ) { nb_clusters++; c = c->next; }
  fprintf(OUTPUT, "%d\n", nb_clusters);
  c = CLUSTERS; while ( c != NULL ) {
    fprintf(OUTPUT, "%d\n %d\n %f\n %d\n", c->nb_segments, c->duplicated, c->ratio, c->weight);
```

```
    s = c->segments; while ( s != NULL ) {
      fprintf(OUTPUT, "%d\n%d\n%d\n", SEGMENT_GENOME(s), SEGMENT_START(s), SEGMENT_END(s));
      s = SEGMENT_NEXT(s);
    }
    c = c->next;
  }
}


// ********************************************************************************
// MAIN FUNCTION
// --------------------------------------------------------------------------------

// --------------------------------------------------------------------------------
int main(int argc, char *argv[]) {
  int k, i, w, a, f;
  CLUSTERING_List c;
  double RATIO1, RATIO2, RATIO_STEP;
  int    MIN_COMMON, MIN_OCC, MIN_GENOMES, MIN_FAM;
  int    OMEGA1, OMEGA2, OMEGA_STEP;

  // ------------------------------------------------------------------------------
  // --------- Reading the genomes data -------------------------------------------
  NB_GENOMES = atoi(argv[1]);

  SIZE_GENOMES  = (long *)   malloc(NB_GENOMES * sizeof(long));
  GENOMES       = (long **)  malloc(NB_GENOMES * sizeof(long *));
  SIGNS         = (long **)  malloc(NB_GENOMES * sizeof(long *));
  POS_START     = (long **)  malloc(NB_GENOMES * sizeof(long *));
  POS_END       = (long **)  malloc(NB_GENOMES * sizeof(long *));
  NB_FAMILIES   = 0;

  DELTA      = atoi(argv[1+(2*NB_GENOMES)+1]);
  // LOWEST RATIO DEFINING EDGES OF THE SIMILARITY GRAPH
  RATIO1     = atof(argv[2+(2*NB_GENOMES)+1]);
  // HIGHEST RATIO DEFINING EDGES OF THE SIMILARITY GRAPH
  RATIO2     = atof(argv[3+(2*NB_GENOMES)+1]);
  // INCREASE BETWEEN SUCCESSIVE RATIOS
  RATIO_STEP = atof(argv[4+(2*NB_GENOMES)+1]);
  // MINIMUM NUMBER OF COMMON GENES TO DEFINE AN EDGE
  MIN_COMMON = atoi(argv[5+(2*NB_GENOMES)+1]);
  // LOWEST CONSIDERED OMEGA VALUE
  OMEGA1 = atoi(argv[6+(2*NB_GENOMES)+1]);
  // HIGHEST CONSIDERED OMEGA VALUE
```

```
OMEGA2 = atoi(argv[7+(2*NB_GENOMES)+1]);
// INCREASE BETWEEN SUCCESSIVE OMEGA VALUES
OMEGA_STEP = atoi(argv[8+(2*NB_GENOMES)+1]);
// OUTPUT FILE
OUTPUT = fopen(argv[9+(2*NB_GENOMES)+1], "w");

for ( a = 0; a < argc; a++ ) { fprintf(OUTPUT, "[ARG] %s\n", argv[a]); }

fprintf(OUTPUT, "[LOG]\n");
for ( k = 1; k <= NB_GENOMES; k++ ) {
  fprintf(OUTPUT, "[LOG] Downloading file %s %d %d\n",
       argv[1+2*(k-1)+1], k, atol(argv[1+2*(k-1)+2]));
  INPUT_Read_Genome(argv[1+2*(k-1)+1], k, atol(argv[1+2*(k-1)+2]));
}
fprintf(OUTPUT, "[LOG] Downloading genomes : %d known families\n", NB_FAMILIES);
INPUT_Add_Unlabelled();
fprintf(OUTPUT, "[LOG] Labelling unknown genes : %d families\n", NB_FAMILIES);


// ------------------------------------------------------------------------------
// --------- Segmentation phase -------------------------------------------------
SEGMENTATION_Compute_Weights();

for ( OMEGA = OMEGA1; OMEGA <= OMEGA2; OMEGA += OMEGA_STEP ) {
  fprintf(OUTPUT, "[LOG]\n[LOG] Weight OMEGA = %d \n", OMEGA);
  // --------- Segmentation -----------------------------------------------------
  for ( k = 1; k <= NB_GENOMES; k++ ) {
    SEGMENTATION_Compute_Statistics(k, OMEGA);
  }
  SEGMENTATION_Display_Statistics(OUTPUT);
}

// ------------------------------------------------------------------------------
// ----------- CLustering phase -------------------------------------------------
CLUSTERING_MIN_COMMON = MIN_COMMON;
CLUSTERS_GRAPH        = NULL;
for ( OMEGA = OMEGA1; OMEGA <= OMEGA2; OMEGA += OMEGA_STEP ) {
  // --------- Computing the connected components of this graph ------------------
  for ( CLUSTERING_RATIO  = RATIO1;
CLUSTERING_RATIO <= RATIO2;
CLUSTERING_RATIO += RATIO_STEP ) {
    fprintf(OUTPUT, "[LOG] Computing clusters OMEGA %d RATIO %f\n",
    OMEGA, CLUSTERING_RATIO);
    // --------- Computing the similarity graph ----------------------------------
```

```
      CLUSTERING_Graph_Allocate();
      CLUSTERING_Compute_Edges();
      CLUSTERING_Compute_Clusters_DFS();
      CLUSTERING_Graph_Free();
    }
  }

  // ---------------------------------------------------------------------------
  // ------- Post-processing Clusters ------------------------------------------
  fprintf(OUTPUT, "[LOG] Computing statistics of clusters induced by the graph\n");
  STATISTICS_List_Clusters(CLUSTERS_GRAPH);
  fprintf(OUTPUT, "[LOG] Sorting and cleaning clusters\n");
  CLUSTERS_GRAPH = CLUSTERS_Sort(CLUSTERS_GRAPH);

  // ---------------------------------------------------------------------------
  // ------- Output ------------------------------------------------------------
  fprintf(OUTPUT, "[LOG] Printing clusters\n");
  fprintf(OUTPUT, "[LOG]\n");
  DISPLAY_List_Clusters(CLUSTERS_GRAPH);

  close(OUTPUT);
}
```

### A.2.2 display-clusters-html.c

```
// ---------------------------------------------------------------------------
// HTML display of gene clusters, v1.0
// Cedric Chauve and Karine St-Onge
// LaCIM, Universite du Quebec a Montreal
// May 2005.
// ---------------------------------------------------------------------------

// ***************************************************************************
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
// ***************************************************************************

// ***************************************************************************
// CONSTANTS
// ---------------------------------------------------------------------------

#define ANNOTATION_MAX 300   // Max number of characters in a gene annotation
#define FAM_UNDEF      0     // Number indicating a gene in the input does not
```

```
                            // belong to any family
// ****************************************************************************

// USER DEFINED PARAMETER. OUTPUT FILE.
static FILE *OUTPUT;

// ****************************************************************************
// INPUT: GENOMES AND ANNOTATIONS
// ----------------------------------------------------------------------------
// Display annotations or not
static int ANNOT;
// Display duplicated clusters or not
static int DUPLICATED;

// Number of different gene families
static long NB_FAMILIES;

// Number of genomes
static int  NB_GENOMES;

// Number of genes in each genome
static long *SIZE_GENOMES;
// Size of genome _K_ (_K_ from 1 to NB_GENOMES)
#define SIZE(_K_)  SIZE_GENOMES[(_K_)-1]

// Arrays of sequences of genomes
static long **GENOMES;
// Gene in position _I_ (from 1 to SIZE_GENOMES[_K_])
// of genome _K_ (from 1 to NB_GENOMES)
#define GENE(_K_, _I_) GENOMES[(_K_)-1][(_I_)-1]

// Arrays of signs of genes in genomes
static int **SIGNS;
// Sign of gene in position (from 1 to SIZE_GENOMES[_K_])
// of genome _K_ (from 1 to NB_GENOMES)
#define SIGN(_K_, _I_) SIGNS[(_K_)-1][(_I_)-1]
// The three values for the sign of a gene
#define SIGN_UNDEF 0 // Undefined
#define SIGN_PLUS  1 // Plus
#define SIGN_MINUS 2 // Minus

// Starting positions of genes in genomes
static long **POS_START;
// Starting base of gene in position (from 1 to SIZE_GENOMES[_K_])
```

```
// of genome _K_ (from 1 to NB_GENOMES)
#define START(_K_, _I_) POS_START[(_K_)-1][(_I_)-1]

// Ending positions of genes in genomes
static long  **POS_END;
// Ending base of gene in position (from 1 to SIZE_GENOMES[_K_])
// of genome _K_ (from 1 to NB_GENOMES)
#define END(_K_, _I_) POS_END[(_K_)-1][(_I_)-1]

// Annotations of the genes
static char ***ANNOTATIONS;
// Annotation of gene in position (from 1 to SIZE_GENOMES[_K_])
// of genome _K_ (from 1 to NB_GENOMES)
#define ANNOTATION(_K_, _I_) ANNOTATIONS[(_K_)-1][(_I_)-1]

// Reading file A for the annotation of gene in position i of genome k
// A should be open in mode "w"
// i from 1 to SIZE_GENOMES[k], k from 1 to NB_GENOMES
static void READ_ANNOTATION(int k, long i, FILE *A) {
  int c;
  ANNOTATIONS[k-1][i-1] = (char *) malloc (ANNOTATION_MAX * sizeof(char));
  ANNOTATIONS[k-1][i-1] = fgets(ANNOTATIONS[k-1][i-1], ANNOTATION_MAX-1, A);
  c = 0; while ( ANNOTATIONS[k-1][i-1][c] != '\n' ) { c++; }
  ANNOTATIONS[k-1][i-1] = (char *) realloc (ANNOTATIONS[k-1][i-1],
    (c+2)*sizeof(char));
  ANNOTATIONS[k-1][i-1][c+1] = '\0';
}

// Importing the data from the two files containing a genome
// Maximum length of a line containing the gene information other than annotations
#define LG_STR 200
// genes = name of the file containing the genes informations (number, sign,
//         starting base, ending base)
// annotations = name og the file containing the annotations
// k = number of the genome (from 1 to NB_GENOMES)
// nb_genes = number of genes in the current genome
static void INPUT_Read_Genome(char *genes, char *annotations, int k, int nb_genes) {
  FILE *F;    // File containing the sequence of genes
  FILE *A;    // File with annotations
  int   i;    // Cursor on the lines of the file genes
  int   j, l; // Cursors on a line of the file genes
  long  x;    // The current gene
  // Temporary strings used to read the file genes
  char *gene = (char *) malloc(LG_STR * sizeof(char));
```

```
  char *info = (char *) malloc(LG_STR * sizeof(char));

  F = fopen(genes, "r");
  A = fopen(annotations, "r");

  for ( i = 1; i <= nb_genes; i++ ) {
    // Reading the gene number
    gene = fgets(gene, LG_STR, F);
    j = 0; while ( gene[j] != ' ' ) { j++; }
    info = strncpy(info, gene, j); info[j] = '\0';
    GENE(k,i) = atol(info);
    // Reading the gene sign
    j++; if ( gene[j] == '-' ) { SIGN(k,i) = SIGN_MINUS; }
    else                       { SIGN(k,i) = SIGN_PLUS; }
    // Reading the starting and ending positions
    j+= 2; l = j; while ( gene[j] != ' ' ) { j++; }
    info = strncpy(info, gene, j); info[j] = '\0';
    START(k,i) = atol(info+l);
    info = strcpy(info, gene);
    END(k,i) = atol(info+j+1);
    if ( START(k,i) > END(k,i) ) {
      l = START(k,i); START(k,i) = END(k,i); END(k,i) = l;
    }
    if ( GENE(k,i) > NB_FAMILIES ) { NB_FAMILIES = GENE(k,i); }
    // Reading the annotation of the current gene
    if ( ANNOT ) READ_ANNOTATION(k, i, A);
  }
  free(gene); free(info);
}

// If some genes are numbered by 0, one gives to each of them a new and
// unique numbers: each of them defines a family with one member.
static void INPUT_Add_Unlabelled() {
  int k, i; // Cursors on the genes
  for ( k = 1; k <= NB_GENOMES; k++ ) {
    for ( i = 1; i <= SIZE(k); i++ ) {
      if ( GENE(k, i) == FAM_UNDEF ) { GENE(k,i) = ++NB_FAMILIES; }
    }
  }
}
// ****************************************************************************

// ****************************************************************************
// SEGMENTS OF GENES: DATA STRUCTURE AND BASIC FUNCTIONS
```

```
// ------------------------------------------------------------------------------

// Data structure for a segment, defined on a genome k, between genes in positions
// i and j.
typedef struct segment {
  int  k; // genome
  long i; // i = start, j = end
  long j; // j > i
} *SEGMENT;

#define SEGMENT_GENOME(_S_) ((_S_)->k)
#define SEGMENT_START(_S_)  ((_S_)->i)
#define SEGMENT_END(_S_)    ((_S_)->j)

// Allocating a DELTA-segment with parameters k, i, j
static SEGMENT SEGMENT_Allocate(int k, long i, long j) {
  SEGMENT s =
    (SEGMENT) malloc (sizeof(struct segment));
  s->k = k; s->i = i; s->j = j;
  return(s);
}
// Freeing a DELTA-segment
static void SEGMENT_Free(SEGMENT s) {
  free(s);
}
// ********************************************************************************


// ********************************************************************************
// GENE FAMILIES AND GENE TEAMS: DATA STRUCTURE AND BASIC FUNCTIONS
// ------------------------------------------------------------------------------
// IMPORTANT. A gene team is encoded by a list of gene families.

typedef struct family {
  long label;          // Label of the gene family
  // The next fields, the number of occurrences of this family in a cluster,
  // and the number of genomes of a clusters where this family appears
  // are used when a team is associated to a cluster of segments
  int  nb_occurrences;
  int  nb_genomes;
  int  last_genome; // Last genome of a cluster where this family was seen
  struct family *next;
} *FAMILY;
```

```
#define FAMILY_LABEL(_F_)  ((_F_)->label)
#define FAMILY_NB_OCC(_F_) ((_F_)->nb_occurrences)
#define FAMILY_NB_GEN(_F_) ((_F_)->nb_genomes)
#define FAMILY_LAST(_F_)   ((_F_)->last_genome)
#define TEAM_NEXT_FAM(_F_) ((_F_)->next)

// Allocation of a family
static FAMILY FAMILY_Allocate(long label, int nb_occurrences, int last_genome) {
  FAMILY f = (FAMILY) malloc(sizeof (struct family));
  f->label = label;
  f->nb_occurrences = nb_occurrences;
  f->nb_genomes = 1;
  f->last_genome = last_genome;
  f->next = NULL;
  return(f);
}
// Freeing a family
static void FAMILY_Free(FAMILY f) {
  free(f);
}
// Freeing a team
static void TEAM_Free(FAMILY t) {
  if ( t->next != NULL ) TEAM_Free(t->next);
  FAMILY_Free(t);
}
static FAMILY TEAM_Sort(FAMILY f) {
  int    stop;                        // Indicates when stopping the sorting
  FAMILY head, prev1, prev2, current; // Cursors on the team
  stop = 0; head = f; while ( stop == 0 ) {
    stop = 1; prev1 = NULL; prev2 = head; current = head->next;
    while ( current != NULL ) {
      if ( (current->nb_genomes > prev2->nb_genomes) ||
   ((current->nb_genomes == prev2->nb_genomes) &&
    (current->nb_occurrences > prev2->nb_occurrences)) ){
        stop = 0;
        prev2->next   = current->next;
        current->next = prev2;
        if ( prev1 != NULL ) { prev1->next = current; }
        else                 { head = current; }
        prev1 = current; current = prev2->next;
      }
      else { prev1 = prev2; prev2 = current; current = current->next; }
    }
  }
```

```
  return(head);
}
// ****************************************************************************


// ****************************************************************************
// CLUSTERS: DATA STRUCTURE AND BASIC FUNCTIONS
// ----------------------------------------------------------------------------

// ----------------------------------------------------------------------------
// List of clusters.
typedef struct cluster {
  SEGMENT *segments;   // List of segments of a cluster
  double  ratio;       // Min ratio used for this cluster
  int     weight;      // Weight used for this cluster
  int     duplicated;  // Indicate if this cluster already exists (1 = yes, 0 = no)
  int     nb_genomes;  // Number of genomes covered by the cluster
  int     nb_segments; // Number of segments in the cluster
  int     nb_families; // Number of gene families the cluster
  int     nb_genes;    // Number of genes the cluster
  FAMILY  families;    // Families present in the cluster
  double  score_comb;  // Combinatorial significance
} *CLUSTER;

// Allocation of a cluster
static CLUSTER CLUSTER_Allocate(double ratio, int weight, int dup, int nb_seg) {
  CLUSTER c      = (CLUSTER) malloc(sizeof(struct cluster));
  c->ratio       = ratio;
  c->weight      = weight;
  c->duplicated  = dup;
  c->segments    = (SEGMENT *) malloc(nb_seg * sizeof(SEGMENT));
  c->nb_segments = nb_seg;
  c->families    = NULL;
  c->nb_genomes  = c->nb_families = c->nb_genes = 0;
  c->score_comb  = 0.0;
  return(c);
}
// Freeing a cluster
static void CLUSTER_Free(CLUSTER c) {
  if ( c->segments != NULL ) { free(c->segments); }
  if ( c->families != NULL ) { TEAM_Free(c->families);    }
  free(c);
}
// ****************************************************************************
```

```
// ****************************************************************************
// INPUT: READING CLUSTERS
// ----------------------------------------------------------------------------

// GLOBAL ARRAY OF CLUSTERS
static CLUSTER *ARRAY_CLUSTERS;
// NUMBER OF CLUSTERS
static long NB_CLUSTERS;

static void CLUSTERS_Read(FILE *F) {
  char  *line = (char *) malloc (LG_STR * sizeof(char));
  int    i, j, nb_seg, weight, dup;
  int    k;
  long   i1, j1;
  double ratio;

  do {
    line = fgets(line, LG_STR, F);
    //fprintf(OUTPUT, "%s<br>\n", line);
  } while ( line[0] == '[' );
  //fprintf(OUTPUT, "\n<hr>\n");

  NB_CLUSTERS = atoi(line);
  ARRAY_CLUSTERS = (CLUSTER *) calloc(NB_CLUSTERS, sizeof(CLUSTER));

  for (i = 0; i < NB_CLUSTERS; i++ ) {
    line = fgets(line, LG_STR, F); nb_seg = atoi(line);
    line = fgets(line, LG_STR, F); dup    = atoi(line);
    line = fgets(line, LG_STR, F); ratio  = atof(line);
    line = fgets(line, LG_STR, F); weight = atoi(line);
    ARRAY_CLUSTERS[i] = CLUSTER_Allocate(ratio, weight, dup, nb_seg);
    for ( j = 0; j < nb_seg; j++ ) {
      line = fgets(line, LG_STR, F); k  = atoi(line);
      line = fgets(line, LG_STR, F); i1 = atoi(line);
      line = fgets(line, LG_STR, F); j1 = atoi(line);
      ARRAY_CLUSTERS[i]->segments[j] = SEGMENT_Allocate(k, i1, j1);
    }
  }
}

// ****************************************************************************
// POSTPROCESSING OF CLUSTERS: COMPUTING STATISTICS FOR A CLUSTER
```

```
// ------------------------------------------------------------------------------
// These statistics are the number of segments and genes (that were already
// computed in CLUSTER_Add_Segment), the number of genomes and the number of
// genes. Moreover for each gene family, the number of occurrences is recorded.

// Adding a family in the list of families of a cluster.
// Families are inserted in increasing order of their label.
// (k,i): position of the gene defining the family
static void STATISTICS_Family_Add(CLUSTER c, int k, long i) {
  FAMILY f, prev; // Cursors on the list of gene families of c
  if ( c->families == NULL ) { // First family encountered
    c->families = FAMILY_Allocate(GENE(k,i), 1, k);
    TEAM_NEXT_FAM(c->families) = NULL;
    c->nb_families = 1;
  }
  else {
    // Looking for the position where should be the current family (k,i)
    f = c->families; prev = NULL;
    while ( f != NULL && FAMILY_LABEL(f) < GENE(k,i) ) {
      prev = f; f = TEAM_NEXT_FAM(f);
    }
    // The family (k,i) is already recorded
    if ( f != NULL && f->label == GENE(k,i) ) {
      FAMILY_NB_OCC(f) += 1;
      if ( FAMILY_LAST(f) != k ) {
FAMILY_NB_GEN(f) += 1; FAMILY_LAST(f) = k;
      }
    }
    // The family (k,i) is not recorded
    else {
      FAMILY x = FAMILY_Allocate(GENE(k,i), 1, k);
      TEAM_NEXT_FAM(x) = f;
      c->nb_families++;
      // It should be recorded as the first family
      if ( f == c->families ) { c->families = x; }
      // It should not be recorded as the first family
      else { TEAM_NEXT_FAM(prev) = x; }
    }
  }
}

// Array used to record the genomes where a cluster is present (0 means NO)
static int *STATISTICS_GENOMES;
// Computing the statistics of a single cluster
```

```
static void STATISTICS_Cluster(CLUSTER c) {
  int k, i, s; // Cursors on genes of a cluster and on segments
  for ( k = 1; k <= NB_GENOMES; k++ ) { STATISTICS_GENOMES[k-1] = 0; }
  for ( s = 0; s < c->nb_segments; s++ ) {
    STATISTICS_GENOMES[c->segments[s]->k-1]++;
    for ( i = c->segments[s]->i; i <= c->segments[s]->j; i++ ) {
      STATISTICS_Family_Add(c, c->segments[s]->k, i);
      c->nb_genes++;
    }
  }
  for ( k = 1; k <= NB_GENOMES; k++ ) {
    if ( STATISTICS_GENOMES[k-1] > 0 ) c->nb_genomes++;
  }
  c->score_comb = ((double) c->nb_genes) / ((double) c->nb_families);
}

// Computing the statistics of all clusters
static void STATISTICS_Clusters() {
  int c;
  STATISTICS_GENOMES = (int *) malloc(NB_GENOMES * sizeof(int));
  for ( c = 0; c < NB_CLUSTERS; c++ ) { STATISTICS_Cluster(ARRAY_CLUSTERS[c]); }
  free(STATISTICS_GENOMES);
}
// ****************************************************************************


// ****************************************************************************
// OUTPUT OF CLUSTERS.
// ----------------------------------------------------------------------------

// ----------------------------------------------------------------------------
// Variable recording the number of the current cluster
static int OUTPUT_NB_CLUSTER;

static int    NB_SEGS;
static int    NB_FAMS;
static int    NB_GENES;
static double SCORE;

// Output of a single segment
#define STR_SIGN(_K_,_J_) (SIGN((_K_), (_J_)) == SIGN_MINUS ? "-" : "+")
static void DISPLAY_Segment(SEGMENT s) {
  int c; // Cursor on the gene sof a segment
  fprintf(OUTPUT, " <b>[%d]</b> ", s->i);
```

```
  for ( c = SEGMENT_START(s); c <= SEGMENT_END(s); c++ ) {
    fprintf(OUTPUT, "%s%d ",
    STR_SIGN(SEGMENT_GENOME(s), c),
    GENE(SEGMENT_GENOME(s), c));
  }
  fprintf(OUTPUT, " <b>[%d] ...</b>", s->j);
}

// Output of the annotations of a segment
static void DISPLAY_Annotations_Segment(SEGMENT s) {
  int c; // Cursor on the gene sof a segment
  for ( c = SEGMENT_START(s); c <= SEGMENT_END(s); c++ ) {
    fprintf(OUTPUT, "<b><font color=\"green\">[C.A]</b> [%d-%d]</font> %s<br>\n",
    s->k,
    GENE(SEGMENT_GENOME(s), c),
    ANNOTATION(SEGMENT_GENOME(s), c));
  }
}

// Output of a complete cluster
// nb_max_segments = nb max of segments allowed for a cluster.
// Above, clusters are not displayed
// nb_min_oc = nb min of occurrences to display a family
static void DISPLAY_Cluster(CLUSTER c) {
  int    s, k;
  FAMILY f;
  // Output of general informations
  fprintf(OUTPUT, "<b><font color=\"blue\">[C.H]</b></font>\n");
  fprintf(OUTPUT, " CLUSTER %d (OMEGA %d RHO %f) ",
  OUTPUT_NB_CLUSTER++, c->weight, c->ratio);
  fprintf(OUTPUT, "%d SEGMENTS %d GENES %d FAMILIES %d GENOMES\n<br>\n",
  c->nb_segments, c->nb_genes, c->nb_families, c->nb_genomes);
  // Output of gene families
  fprintf(OUTPUT, "<b><font color=\"yellow\">[C.F]</font></b>    ");
  c->families = TEAM_Sort(c->families);
  f = c->families; while ( f != NULL ) {
    if ( FAMILY_NB_GEN(f) > 1 &&
 ((double) FAMILY_NB_GEN(f)) > ((double) c->nb_genomes)/2.0 ) {
      fprintf(OUTPUT, "[%d-%d-%d] ", FAMILY_LABEL(f), FAMILY_NB_OCC(f), FAMILY_NB_GEN(f));
    }
    f = TEAM_NEXT_FAM(f);
  }
  //fprintf(OUTPUT, "\n<br>\n<b><font color=\"blue\">[C.H]</font></b>");
  // Output of segments and annotations
```

```
  k = c->segments[0]->k-1;
  for ( s = 0; s < c->nb_segments; s++ ) {
    if ( c->segments[s]->k > k ) {
      k = c->segments[s]->k;
      fprintf(OUTPUT, "\n<br>\n<font color=\"red\"><b>[C.S] [%d]</b></font> ", c->segments[s]->k);
    }
    DISPLAY_Segment(c->segments[s]);
  }
  fprintf(OUTPUT, "\n<br>\n");
  if ( ANNOT )
    for ( s = 0; s < c->nb_segments; s++ ) { DISPLAY_Annotations_Segment(c->segments[s]); }
}

// Output of all clusters
static void DISPLAY_List_Clusters() {
  int c;
  OUTPUT_NB_CLUSTER = 1;
  for ( c = 0; c < NB_CLUSTERS; c++ ) {
    if ( ARRAY_CLUSTERS[c]->nb_segments <= NB_SEGS  &&
 ARRAY_CLUSTERS[c]->nb_families <= NB_FAMS  &&
 ARRAY_CLUSTERS[c]->nb_genes    <= NB_GENES &&
 ARRAY_CLUSTERS[c]->score_comb  >= SCORE    &&
 (ARRAY_CLUSTERS[c]->duplicated == 0 ||
  (ARRAY_CLUSTERS[c]->duplicated == 1 && DUPLICATED == 1)) ) {
      DISPLAY_Cluster(ARRAY_CLUSTERS[c]);
      fprintf(OUTPUT, "<hr>\n");
    }
  }
}
// ****************************************************************************


// ****************************************************************************
// MAIN FUNCTION
// ----------------------------------------------------------------------------

// ----------------------------------------------------------------------------
int main(int argc, char *argv[]) {
  int   k, i, w, a, f, nb_clusters;
  FILE *INPUT;

  // ----------------------------------------------------------------------------
  // --------- Reading the genomes data -----------------------------------------
  NB_GENOMES = atoi(argv[1]);
```

```
  NB_SEGS    = atoi(argv[2+(3*NB_GENOMES)]);
  NB_FAMS    = atoi(argv[3+(3*NB_GENOMES)]);
  NB_GENES   = atoi(argv[4+(3*NB_GENOMES)]);
  SCORE      = atof(argv[5+(3*NB_GENOMES)]);
  INPUT      = fopen(argv[6+(3*NB_GENOMES)], "r");
  OUTPUT     = fopen(argv[7+(3*NB_GENOMES)], "w");
  ANNOT      = atoi(argv[8+(3*NB_GENOMES)]);
  DUPLICATED = atoi(argv[9+(3*NB_GENOMES)]);

  SIZE_GENOMES  = (long *)   malloc(NB_GENOMES * sizeof(long));
  GENOMES       = (long **)  malloc(NB_GENOMES * sizeof(long *));
  SIGNS         = (int **)   malloc(NB_GENOMES * sizeof(int *));
  POS_START     = (long **)  malloc(NB_GENOMES * sizeof(long *));
  POS_END       = (long **)  malloc(NB_GENOMES * sizeof(long *));
  if ( ANNOT ) ANNOTATIONS   = (char ***) malloc(NB_GENOMES * sizeof(char **));
  NB_FAMILIES   = 0;

  fprintf(OUTPUT, "<html>\n<body>\n");
//   for ( a = 0; a < argc; a++ ) {
//     fprintf(OUTPUT, "%s ", argv[a]);
//   }
//   fprintf(OUTPUT, "<hr>\n");

  for ( k = 1; k <= NB_GENOMES; k++ ) {
    SIZE(k)          = atol(argv[1+3*(k-1)+3]);
    GENOMES[k-1]     = (long *)  malloc (SIZE(k) * sizeof(long));
    SIGNS[k-1]       = (int *)   malloc (SIZE(k) * sizeof(int));
    POS_START[k-1]   = (long *)  malloc (SIZE(k) * sizeof(long));
    POS_END[k-1]     = (long *)  malloc (SIZE(k) * sizeof(long));
    if ( ANNOT ) ANNOTATIONS[k-1] = (char **) malloc (SIZE(k) * sizeof(char *));
    INPUT_Read_Genome(argv[1+3*(k-1)+1], argv[1+3*(k-1)+2], k, SIZE(k));
  }
  INPUT_Add_Unlabelled();

  // ------------------------------------------------------------------------
  // --------- Reading clusters -----------------------------------------------
  CLUSTERS_Read(INPUT);

  // ------------------------------------------------------------------------
  // ------- Statistics -----------------------------------------------------
  STATISTICS_Clusters();

  // ------------------------------------------------------------------------
  // ------- Output ---------------------------------------------------------
```

```
  DISPLAY_List_Clusters();
  fprintf(OUTPUT, "</body>\n</html>");
  fclose(OUTPUT);
}
```

### A.2.3 display-clusters-txt.c

```
// -------------------------------------------------------------------------
// TEXT display of gene clusters, v1.0
// Cedric Chauve and Karine St-Onge
// LaCIM, Universite du Quebec a Montreal
// May 2005.
// -------------------------------------------------------------------------

// *************************************************************************
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
// *************************************************************************

// *************************************************************************
// CONSTANTS
// -------------------------------------------------------------------------

#define ANNOTATION_MAX 300   // Max number of characters in a gene annotation
#define FAM_UNDEF      0     // Number indicating a gene in the input does not
                             // belong to any family
// *************************************************************************

// *************************************************************************
// INPUT: GENOMES AND ANNOTATIONS
// -------------------------------------------------------------------------
// Display annotations or not
static int ANNOT;
// Display duplicated clusters or not
static int DUPLICATED;

// Number of different gene families
static long NB_FAMILIES;

// Number of genomes
static int  NB_GENOMES;

// Number of genes in each genome
```

```
static long *SIZE_GENOMES;
// Size of genome _K_ (_K_ from 1 to NB_GENOMES)
#define SIZE(_K_)  SIZE_GENOMES[(_K_)-1]

// Arrays of sequences of genomes
static long **GENOMES;
// Gene in position _I_ (from 1 to SIZE_GENOMES[_K_])
// of genome _K_ (from 1 to NB_GENOMES)
#define GENE(_K_, _I_) GENOMES[(_K_)-1][(_I_)-1]

// Arrays of signs of genes in genomes
static int **SIGNS;
// Sign of gene in position (from 1 to SIZE_GENOMES[_K_])
// of genome _K_ (from 1 to NB_GENOMES)
#define SIGN(_K_, _I_) SIGNS[(_K_)-1][(_I_)-1]
// The three values for the sign of a gene
#define SIGN_UNDEF 0 // Undefined
#define SIGN_PLUS  1 // Plus
#define SIGN_MINUS 2 // Minus

// Starting positions of genes in genomes
static long **POS_START;
// Starting base of gene in position (from 1 to SIZE_GENOMES[_K_])
// of genome _K_ (from 1 to NB_GENOMES)
#define START(_K_, _I_) POS_START[(_K_)-1][(_I_)-1]

// Ending positions of genes in genomes
static long  **POS_END;
// Ending base of gene in position (from 1 to SIZE_GENOMES[_K_])
// of genome _K_ (from 1 to NB_GENOMES)
#define END(_K_, _I_) POS_END[(_K_)-1][(_I_)-1]

// Annotations of the genes
static char ***ANNOTATIONS;
// Annotation of gene in position (from 1 to SIZE_GENOMES[_K_])
// of genome _K_ (from 1 to NB_GENOMES)
#define ANNOTATION(_K_, _I_) ANNOTATIONS[(_K_)-1][(_I_)-1]

// Reading file A for the annotation of gene in position i of genome k
// A should be open in mode "w"
// i from 1 to SIZE_GENOMES[k], k from 1 to NB_GENOMES
static void READ_ANNOTATION(int k, long i, FILE *A) {
  int c;
  ANNOTATIONS[k-1][i-1] = (char *) malloc (ANNOTATION_MAX * sizeof(char));
```

```
  ANNOTATIONS[k-1][i-1] = fgets(ANNOTATIONS[k-1][i-1], ANNOTATION_MAX-1, A);
  c = 0; while ( ANNOTATIONS[k-1][i-1][c] != '\n' ) { c++; }
  ANNOTATIONS[k-1][i-1] = (char *) realloc (ANNOTATIONS[k-1][i-1],
    (c+2)*sizeof(char));
  ANNOTATIONS[k-1][i-1][c+1] = '\0';
}

// Importing the data from the two files containing a genome
// Maximum length of a line containing the gene information other than annotations
#define LG_STR 200
// genes = name of the file containing the genes informations (number, sign,
//         starting base, ending base)
// annotations = name og the file containing the annotations
// k = number of the genome (from 1 to NB_GENOMES)
// nb_genes = number of genes in the current genome
static void INPUT_Read_Genome(char *genes, char *annotations, int k, int nb_genes) {
  FILE *F;    // File containing the sequence of genes
  FILE *A;    // File with annotations
  int   i;    // Cursor on the lines of the file genes
  int   j, l; // Cursors on a line of the file genes
  long  x;    // The current gene
  // Temporary strings used to read the file genes
  char *gene = (char *) malloc(LG_STR * sizeof(char));
  char *info = (char *) malloc(LG_STR * sizeof(char));

  F = fopen(genes, "r");
  A = fopen(annotations, "r");

  for ( i = 1; i <= nb_genes; i++ ) {
    // Reading the gene number
    gene = fgets(gene, LG_STR, F);
    j = 0; while ( gene[j] != ' ' ) { j++; }
    info = strncpy(info, gene, j); info[j] = '\0';
    GENE(k,i) = atol(info);
    // Reading the gene sign
    j++; if ( gene[j] == '-' ) { SIGN(k,i) = SIGN_MINUS; }
    else                       { SIGN(k,i) = SIGN_PLUS; }
    // Reading the starting and ending positions
    j+= 2; l = j; while ( gene[j] != ' ' ) { j++; }
    info = strncpy(info, gene, j); info[j] = '\0';
    START(k,i) = atol(info+l);
    info = strcpy(info, gene);
    END(k,i) = atol(info+j+1);
    if ( START(k,i) > END(k,i) ) {
```

```
      l = START(k,i); START(k,i) = END(k,i); END(k,i) = l;
    }
    if ( GENE(k,i) > NB_FAMILIES ) { NB_FAMILIES = GENE(k,i); }
    // Reading the annotation of the current gene
    if ( ANNOT ) READ_ANNOTATION(k, i, A);
  }
  free(gene); free(info);
}

// If some genes are numbered by 0, one gives to each of them a new and
// unique numbers: each of them defines a family with one member.
static void INPUT_Add_Unlabelled() {
  int k, i; // Cursors on the genes
  for ( k = 1; k <= NB_GENOMES; k++ ) {
    for ( i = 1; i <= SIZE(k); i++ ) {
      if ( GENE(k, i) == FAM_UNDEF ) { GENE(k,i) = ++NB_FAMILIES; }
    }
  }
}
// ****************************************************************************

// ****************************************************************************
// SEGMENTS OF GENES: DATA STRUCTURE AND BASIC FUNCTIONS
// ----------------------------------------------------------------------------

// Data structure for a segment, defined on a genome k, between genes in positions
// i and j.
typedef struct segment {
  int  k; // genome
  long i; // i = start, j = end
  long j; // j > i
} *SEGMENT;

#define SEGMENT_GENOME(_S_) ((_S_)->k)
#define SEGMENT_START(_S_)  ((_S_)->i)
#define SEGMENT_END(_S_)    ((_S_)->j)

// Allocating a DELTA-segment with parameters k, i, j
static SEGMENT SEGMENT_Allocate(int k, long i, long j) {
  SEGMENT s =
    (SEGMENT) malloc (sizeof(struct segment));
  s->k = k; s->i = i; s->j = j;
  return(s);
}
```

```
// Freeing a DELTA-segment
static void SEGMENT_Free(SEGMENT s) {
  free(s);
}
// ********************************************************************************


// ********************************************************************************
// GENE FAMILIES AND GENE TEAMS: DATA STRUCTURE AND BASIC FUNCTIONS
// --------------------------------------------------------------------------------
// IMPORTANT. A gene team is encoded by a list of gene families.

typedef struct family {
  long label;          // Label of the gene family
  // The next fields, the number of occurrences of this family in a cluster,
  // and the number of genomes of a clusters where this family appears
  // are used when a team is associated to a cluster of segments
  int  nb_occurrences;
  int  nb_genomes;
  int  last_genome; // Last genome of a cluster where this family was seen
  struct family *next;
} *FAMILY;

#define FAMILY_LABEL(_F_)  ((_F_)->label)
#define FAMILY_NB_OCC(_F_) ((_F_)->nb_occurrences)
#define FAMILY_NB_GEN(_F_) ((_F_)->nb_genomes)
#define FAMILY_LAST(_F_)   ((_F_)->last_genome)
#define TEAM_NEXT_FAM(_F_) ((_F_)->next)

// Allocation of a family
static FAMILY FAMILY_Allocate(long label, int nb_occurrences, int last_genome) {
  FAMILY f = (FAMILY) malloc(sizeof (struct family));
  f->label = label;
  f->nb_occurrences = nb_occurrences;
  f->nb_genomes = 1;
  f->last_genome = last_genome;
  f->next = NULL;
  return(f);
}
// Freeing a family
static void FAMILY_Free(FAMILY f) {
  free(f);
}
// Freeing a team
```

```
static void TEAM_Free(FAMILY t) {
  if ( t->next != NULL ) TEAM_Free(t->next);
  FAMILY_Free(t);
}
static FAMILY TEAM_Sort(FAMILY f) {
  int    stop;                        // Indicates when stopping the sorting
  FAMILY head, prev1, prev2, current; // Cursors on the team
  stop = 0; head = f; while ( stop == 0 ) {
    stop = 1; prev1 = NULL; prev2 = head; current = head->next;
    while ( current != NULL ) {
      if ( (current->nb_genomes > prev2->nb_genomes) ||
   ((current->nb_genomes == prev2->nb_genomes) &&
    (current->nb_occurrences > prev2->nb_occurrences)) ){
        stop = 0;
        prev2->next   = current->next;
        current->next = prev2;
        if ( prev1 != NULL ) { prev1->next = current; }
        else                 { head = current; }
        prev1 = current; current = prev2->next;
      }
      else { prev1 = prev2; prev2 = current; current = current->next; }
    }
  }
  return(head);
}
// ****************************************************************************


// ****************************************************************************
// CLUSTERS: DATA STRUCTURE AND BASIC FUNCTIONS
// ----------------------------------------------------------------------------

// ----------------------------------------------------------------------------
// List of clusters.
typedef struct cluster {
  SEGMENT *segments;   // List of segments of a cluster
  double  ratio;       // Min ratio used for this cluster
  int     weight;      // Weight used for this cluster
  int     duplicated;  // Indicate if this cluster already exists (1 = yes, 0 = no)
  int     nb_genomes;  // Number of genomes covered by the cluster
  int     nb_segments; // Number of segments in the cluster
  int     nb_families; // Number of gene families the cluster
  int     nb_genes;    // Number of genes the cluster
  FAMILY  families;    // Families present in the cluster
```

```
  double  score_comb;  // Combinatorial significance
} *CLUSTER;

// Allocation of a cluster
static CLUSTER CLUSTER_Allocate(double ratio, int weight, int dup, int nb_seg) {
  CLUSTER c      = (CLUSTER) malloc(sizeof(struct cluster));
  c->ratio       = ratio;
  c->weight      = weight;
  c->duplicated  = dup;
  c->segments    = (SEGMENT *) malloc(nb_seg * sizeof(SEGMENT));
  c->nb_segments = nb_seg;
  c->families    = NULL;
  c->nb_genomes  = c->nb_families = c->nb_genes = 0;
  c->score_comb  = 0.0;
  return(c);
}
// Freeing a cluster
static void CLUSTER_Free(CLUSTER c) {
  if ( c->segments != NULL ) { free(c->segments); }
  if ( c->families != NULL ) { TEAM_Free(c->families);    }
  free(c);
}
// ****************************************************************************


// ****************************************************************************
// INPUT: READING CLUSTERS
// ----------------------------------------------------------------------------

// GLOBAL ARRAY OF CLUSTERS
static CLUSTER *ARRAY_CLUSTERS;
// NUMBER OF CLUSTERS
static long NB_CLUSTERS;

static void CLUSTERS_Read(FILE *F) {
  char  *line = (char *) malloc (LG_STR * sizeof(char));
  int    i, j, nb_seg, weight, dup;
  int    k;
  long   i1, j1;
  double ratio;

  do {
    line = fgets(line, LG_STR, F);
  } while ( line[0] == '[' );
```

```
  NB_CLUSTERS = atoi(line);
  ARRAY_CLUSTERS = (CLUSTER *) calloc(NB_CLUSTERS, sizeof(CLUSTER));

  for (i = 0; i < NB_CLUSTERS; i++ ) {
    line = fgets(line, LG_STR, F); nb_seg = atoi(line);
    line = fgets(line, LG_STR, F); dup    = atoi(line);
    line = fgets(line, LG_STR, F); ratio  = atof(line);
    line = fgets(line, LG_STR, F); weight = atoi(line);
    ARRAY_CLUSTERS[i] = CLUSTER_Allocate(ratio, weight, dup, nb_seg);
    for ( j = 0; j < nb_seg; j++ ) {
      line = fgets(line, LG_STR, F); k  = atoi(line);
      line = fgets(line, LG_STR, F); i1 = atoi(line);
      line = fgets(line, LG_STR, F); j1 = atoi(line);
      ARRAY_CLUSTERS[i]->segments[j] = SEGMENT_Allocate(k, i1, j1);
    }
  }
}

// ****************************************************************************
// POSTPROCESSING OF CLUSTERS: COMPUTING STATISTICS FOR A CLUSTER
// ----------------------------------------------------------------------------
// These statistics are the number of segments and genes (that were already
// computed in CLUSTER_Add_Segment), the number of genomes and the number of
// genes. Moreover for each gene family, the number of occurrences is recorded.

// Adding a family in the list of families of a cluster.
// Families are inserted in increasing order of their label.
// (k,i): position of the gene defining the family
static void STATISTICS_Family_Add(CLUSTER c, int k, long i) {
  FAMILY f, prev; // Cursors on the list of gene families of c
  if ( c->families == NULL ) { // First family encountered
    c->families = FAMILY_Allocate(GENE(k,i), 1, k);
    TEAM_NEXT_FAM(c->families) = NULL;
    c->nb_families = 1;
  }
  else {
    // Looking for the position where should be the current family (k,i)
    f = c->families; prev = NULL;
    while ( f != NULL && FAMILY_LABEL(f) < GENE(k,i) ) {
      prev = f; f = TEAM_NEXT_FAM(f);
    }
    // The family (k,i) is already recorded
    if ( f != NULL && f->label == GENE(k,i) ) {
```

```
      FAMILY_NB_OCC(f) += 1;
      if ( FAMILY_LAST(f) != k ) {
FAMILY_NB_GEN(f) += 1; FAMILY_LAST(f) = k;
      }
    }
    // The family (k,i) is not recorded
    else {
      FAMILY x = FAMILY_Allocate(GENE(k,i), 1, k);
      TEAM_NEXT_FAM(x) = f;
      c->nb_families++;
      // It should be recorded as the first family
      if ( f == c->families ) { c->families = x; }
      // It should not be recorded as the first family
      else { TEAM_NEXT_FAM(prev) = x; }
    }
  }
}

// Array used to record the genomes where a cluster is present (0 means NO)
static int *STATISTICS_GENOMES;
// Computing the statistics of a single cluster
static void STATISTICS_Cluster(CLUSTER c) {
  int k, i, s; // Cursors on genes of a cluster and on segments
  for ( k = 1; k <= NB_GENOMES; k++ ) { STATISTICS_GENOMES[k-1] = 0; }
  for ( s = 0; s < c->nb_segments; s++ ) {
    STATISTICS_GENOMES[c->segments[s]->k-1]++;
    for ( i = c->segments[s]->i; i <= c->segments[s]->j; i++ ) {
      STATISTICS_Family_Add(c, c->segments[s]->k, i);
      c->nb_genes++;
    }
  }
  for ( k = 1; k <= NB_GENOMES; k++ ) {
    if ( STATISTICS_GENOMES[k-1] > 0 ) c->nb_genomes++;
  }
  c->score_comb = ((double) c->nb_genes) / ((double) c->nb_families);
}

// Computing the statistics of all clusters
static void STATISTICS_Clusters() {
  int c;
  STATISTICS_GENOMES = (int *) malloc(NB_GENOMES * sizeof(int));
  for ( c = 0; c < NB_CLUSTERS; c++ ) { STATISTICS_Cluster(ARRAY_CLUSTERS[c]); }
  free(STATISTICS_GENOMES);
}
```

```
// ****************************************************************************


// ****************************************************************************
// OUTPUT OF CLUSTERS.
// ----------------------------------------------------------------------------

// ----------------------------------------------------------------------------
// Variable recording the number of the current cluster
static int OUTPUT_NB_CLUSTER;

static int    NB_SEGS;
static int    NB_FAMS;
static int    NB_GENES;
static double SCORE;


// USER DEFINED PARAMETER. OUTPUT FILE.
static FILE *OUTPUT;

// Output of a single segment
#define STR_SIGN(_K_,_J_) (SIGN((_K_), (_J_)) == SIGN_MINUS ? "-" : "+")
static void DISPLAY_Segment(SEGMENT s) {
  int c; // Cursor on the gene sof a segment
  fprintf(OUTPUT, " [%d] ", s->i);
  for ( c = SEGMENT_START(s); c <= SEGMENT_END(s); c++ ) {
    fprintf(OUTPUT, "%s%d ",
    STR_SIGN(SEGMENT_GENOME(s), c),
    GENE(SEGMENT_GENOME(s), c));
  }
  fprintf(OUTPUT, " [%d] ...", s->j);
}

// Output of the annotations of a segment
static void DISPLAY_Annotations_Segment(SEGMENT s) {
  int c; // Cursor on the gene sof a segment
  for ( c = SEGMENT_START(s); c <= SEGMENT_END(s); c++ ) {
    fprintf(OUTPUT, "[C.A] [%d-%d] %s",
    s->k,
    GENE(SEGMENT_GENOME(s), c),
    ANNOTATION(SEGMENT_GENOME(s), c));
  }
}
```

```
// Output of a complete cluster
// nb_max_segments = nb max of segments allowed for a cluster.
// Above, clusters are not displayed
// nb_min_oc = nb min of occurrences to display a family
static void DISPLAY_Cluster(CLUSTER c) {
  int    s, k;
  FAMILY f;
  // Output of general informations
  fprintf(OUTPUT, "[C.H]\n[C.H] ");
  fprintf(OUTPUT, " CLUS. %d RATIO %f WEIGHT %d DUP. %d ",
  OUTPUT_NB_CLUSTER++, c->ratio, c->weight, c->duplicated);
  fprintf(OUTPUT, "SEG. %d GENES %d FAM. %d GENOMES %d SCORE %f\n",
  c->nb_segments, c->nb_genes, c->nb_families, c->nb_genomes, c->score_comb);
  // Output of gene families
  fprintf(OUTPUT, "[C.H]    ");
  c->families = TEAM_Sort(c->families);
  f = c->families; while ( f != NULL ) {
    if ( FAMILY_NB_GEN(f) > 1 &&
 ((double) FAMILY_NB_GEN(f)) > ((double) c->nb_genomes)/2.0 ) {
      fprintf(OUTPUT, "[%d-%d-%d] ", FAMILY_LABEL(f), FAMILY_NB_OCC(f), FAMILY_NB_GEN(f));
    }
    f = TEAM_NEXT_FAM(f);
  }
  fprintf(OUTPUT, "\n[C.H]    ");
  // Output of segments and annotations
  k = c->segments[0]->k-1;
  for ( s = 0; s < c->nb_segments; s++ ) {
    if ( c->segments[s]->k > k ) {
      k = c->segments[s]->k;
      fprintf(OUTPUT, "\n[C.S] [%d] ", c->segments[s]->k);
    }
    DISPLAY_Segment(c->segments[s]);
  }
  fprintf(OUTPUT, "\n");
  if ( ANNOT )
    for ( s = 0; s < c->nb_segments; s++ ) { DISPLAY_Annotations_Segment(c->segments[s]); }
}

// Output of all clusters
static void DISPLAY_List_Clusters() {
  int c;
  OUTPUT_NB_CLUSTER = 1;
  for ( c = 0; c < NB_CLUSTERS; c++ ) {
    if ( ARRAY_CLUSTERS[c]->nb_segments <= NB_SEGS  &&
```

```
 ARRAY_CLUSTERS[c]->nb_families <= NB_FAMS  &&
 ARRAY_CLUSTERS[c]->nb_genes    <= NB_GENES &&
 ARRAY_CLUSTERS[c]->score_comb  >= SCORE    &&
 (ARRAY_CLUSTERS[c]->duplicated == 0 ||
  (ARRAY_CLUSTERS[c]->duplicated == 1 && DUPLICATED == 1)) ) {
      DISPLAY_Cluster(ARRAY_CLUSTERS[c]);
      //      fprintf(OUTPUT, "\n");
    }
  }
}
// ****************************************************************************


// ****************************************************************************
// MAIN FUNCTION
// ----------------------------------------------------------------------------

// ----------------------------------------------------------------------------
int main(int argc, char *argv[]) {
  int   k, i, w, a, f, nb_clusters;
  FILE *INPUT;

  // ----------------------------------------------------------------------------
  // --------- Reading the genomes data -----------------------------------------
  NB_GENOMES = atoi(argv[1]);
  NB_SEGS    = atoi(argv[2+(3*NB_GENOMES)]);
  NB_FAMS    = atoi(argv[3+(3*NB_GENOMES)]);
  NB_GENES   = atoi(argv[4+(3*NB_GENOMES)]);
  SCORE      = atof(argv[5+(3*NB_GENOMES)]);
  INPUT      = fopen(argv[6+(3*NB_GENOMES)], "r");
  OUTPUT     = fopen(argv[7+(3*NB_GENOMES)], "w");
  ANNOT      = atoi(argv[8+(3*NB_GENOMES)]);
  DUPLICATED = atoi(argv[9+(3*NB_GENOMES)]);

  SIZE_GENOMES  = (long *)   malloc(NB_GENOMES * sizeof(long));
  GENOMES       = (long **)  malloc(NB_GENOMES * sizeof(long *));
  SIGNS         = (int **)   malloc(NB_GENOMES * sizeof(int *));
  POS_START     = (long **)  malloc(NB_GENOMES * sizeof(long *));
  POS_END       = (long **)  malloc(NB_GENOMES * sizeof(long *));
  if ( ANNOT ) ANNOTATIONS   = (char ***) malloc(NB_GENOMES * sizeof(char **));
  NB_FAMILIES   = 0;

  fprintf(OUTPUT, "[ARG] ");
  for ( a = 0; a < argc; a++ ) {
```

```
    fprintf(OUTPUT, "%s ", argv[a]);
  }
  fprintf(OUTPUT, "\n");

  for ( k = 1; k <= NB_GENOMES; k++ ) {
    SIZE(k)          = atol(argv[1+3*(k-1)+3]);
    GENOMES[k-1]     = (long *)  malloc (SIZE(k) * sizeof(long));
    SIGNS[k-1]       = (int *)   malloc (SIZE(k) * sizeof(int));
    POS_START[k-1]   = (long *)  malloc (SIZE(k) * sizeof(long));
    POS_END[k-1]     = (long *)  malloc (SIZE(k) * sizeof(long));
    if ( ANNOT ) ANNOTATIONS[k-1] = (char **) malloc (SIZE(k) * sizeof(char *));
    INPUT_Read_Genome(argv[1+3*(k-1)+1], argv[1+3*(k-1)+2], k, SIZE(k));
  }
  INPUT_Add_Unlabelled();

  // ------------------------------------------------------------------------
  // --------- Reading clusters ---------------------------------------------
  CLUSTERS_Read(INPUT);

  // ------------------------------------------------------------------------
  // ------- Statistics -------------------------------------------------------
  STATISTICS_Clusters();

  // ------------------------------------------------------------------------
  // ------- Output -----------------------------------------------------------
  DISPLAY_List_Clusters();
  fclose(OUTPUT);
}
```

# BIBLIOGRAPHIE


Altschul S. *et al.* Basic Local Alignment Search Tool, *Journal of Molecular Biology*, Vol. 215, No. 3, p. 403-410, 1990.

Baldi, P et Brunak S. Bioinformatics : the machine learning approach, *MIT Press*, p. 194-195, 2001.

Bateman A. *et al.* The Pfam Protein Families Database, *Nucleic Acids Research*, Vol. 28, No. 1, p. 263-266, 2000.

Bergeron A. *et al.* Gene teams : a new formalization of gene clusters for comparative genomics, *Computational Biology and Chemistry*, Vol. 27, No 1, p. 59-67, 2003.

Bergeron A. *et al.* The Algorithmic of Gene Teams, *WABI*, p. 464-476, 2002.

Bernot A. Analyse de génomes, Transcriptomes, et Protéomes, *Dunod*, p. 222, 2001.

Blanchette M. A comparative analysis method for detecting binding sites in coding regions, *RECOMB*, p. 57-66, 2003.

Blanchette M. et Tompa M. Discovery of regulatory elements by a computational method for phylogenetic footprinting, *Genome Research* Vol. 12, No. 5, p. 739-748, 2002.

Boffelli D. *et al.* Intraspecies sequence comparisons for annotating genomes, *Genome Research*, Vol. 14, No. 12, p. 2406-2411, 2004.

Bourque G. *et al.* Reconstructing the Genomic Architecture of Ancestral Mammals : Lessons From Human, Mouse, and Rat Genomes, *Genome Research*, Vol. 14, No. 4, p. 507-516, 2004.

Chen Y. *et al.* Operon prediction by comparative genomics : an application to the Synechococcus sp. WH8102 genome, *Nucleic Acids Research*, Vol. 32, No. 7, p. 2147-2157, 2004.

Cordoliani H. Les acides nucléiques, *Nathan*, p. 128, 1994.

Crick F. et Watson J. Molecular structure of Nucleic Acids, *Nature*, Vol. 171, p. 737-738, 1953.

Eddy S. R. Non-coding RNA genes and the modern RNA world, *Nature Reviews on Genetics*, Vol. 2, No. 12, p. 919-929, 2001.

Fitch W. M. Homology a personal view on some of the problems, *Trends Genet*, Vol. 16, No. 5, p. 227-231, 2000.

Gusfield D. Algorithms on Strings, Trees and Sequences : Computer Science and Computational Biology, *Cambridge University Press*, p. 540, 1997.

Hastings M. et Krainer A. Pre-mRNA splicing in the new millenium, *Current Opinion in Cell Biology*, Vol. 13, p. 302309, 2001.

He X. et Goldwasser M. Identifying Conserved Gene Clusters in the Presence of Orthologous Groups, *RECOMB*, p. 272-280, 2004.

Lerat E. *et al.* From gene tree to organismal phylogeny in prokaryotes : the case of $\gamma$-Proteobacteria, *PLoS Biology*, Vol. 1, No. 1, p. 101-109, 2003.

Margulies E. H. *et al.* Identification and Characterization of Multi-Species Conserved Sequences, *Genome Research*, Vol. 13, No. 12, p. 2507-2518, 2003.

Modrek B. et Lee C. A genomic view of alternative splicing, *Nature Genetics*, Vol. 30, No. 1, p. 1319, 2002.

Neapolitan R. et Naimipour K. Foundations of Algorithms using C++ Pseudo Code, Third Edition, *Jones and Bartle Publishers*, p. 633, 2005.

Olivier Jaillon *et al.* Genome duplication in the teleost fish Tetraodon nigroviridis reveals the early vertebrate proto-karyotype, *Nature*, Vol. 431, No. 7011, p. 946-957, 2004.

Pasek S. *et al.* Identification of genomic features using microsyntenies of domains : domain teams, *Genome Research*, Vol. 15, No. 6, p. 867-74, 2005.

Philippe H. et Douady C. J. Horizontal gene transfer and phylogenetics, *Current Opinion in Cell Biology*, Vol. 6, No. 5, p. 498-505, 2003.

Roskin K. M. *et al.* Scoring two-species local alignments to try to statistically separate neutrally evolving from selected dna segments, *RECOMB*, p. 257-266, 2003.

Salgado H. *et al.* RegulonDB (version 4.0) : Transcriptional Regulation, Operon Organization and Growth Conditions in Escherichia coli K-12, *Nucleic Acids Research*, Vol. 32, p. 303-306, 2004.

Sankoff D. Rearrangements and chromosomal evolution, *Current Opinion in Genetics and Development*, Vol. 13, No. 6, p. 583-587, 2003.

Sankoff D. Short inversions and conserved gene clusters, *Bioinformatics*, Vol. 18, No. 10, p. 1305-1308, 2002.

Sankoff D. *et al.* The distribution of inversion lengths in prokaryotes, *RECOMB*, Vol. 3388, p. 97-108, 2005.

Singer G. at al. Clusters of co-expressed genes in mammalian genomes are conserved by natural selection, *Molecular Biology and Evolution*, Vol. 22, No. 3, p. 767-775, 2004.

Tatusov R. *et al.* The COG database : a tool for genome-scale analysis of protein functions and evolution, *Nucleic Acids Research*, Vol. 28, No. 1, p. 33-36, 2000.

Waterston R. H. *et al.* Initial sequencing and comparative analysis of the mouse genome, *Nature*, Vol. 420, No. 6915, p. 520-562, 2002.

Wieczorek D. *et al.* The Rat a-Tropomyosin Gene Generates a Minimum of Six Different mRNAs Coding for Striated, Smooth, and Nonmuscle Isoforms byAlternative Splicing, *Molecular and cellular biology*, Vol. 8, No. 2, p. 679-694, 1988.

Xie G. *et al.* Ancient origin of the tryptophan operon and the dynamics of evolutionary change, *Microbiology and Molecular Biology Reviews*, Vol 67, No. 3, p. 303-342, 2003.

COGNITOR, http ://www.ncbi.nlm.nih.gov/COG/old/xognitor.html

Colibri, http ://genolist.pasteur.fr/Colibri/

Gresham High School, http ://ghs.gresham.k12.or.us/science/ps/sci/ibbio/chem/nucleic/chpt15/chpt15.htm

Le navigateur de génome de l'université de Californie de Santa Cruz USCS (genome browser), http ://genome.ucsc.edu/

Lexique EncycloBio, http ://www.cite-sciences.fr/lexique/

NCBI, http ://www.ncbi.nlm.nih.gov/

PSSM, http ://apps.bioneq.qc.ca/twiki/bin/view/Basedeconnaissances/PSSM

SIGSCAN, http ://thr.cit.nih.gov/molbio/signal/

Subtilist, http ://genolist.pasteur.fr/SubtiList/

TRANSTERM, http ://www.tigr.org/software/transterm.html