

# Introduction à la BioInformatique

Michel Van Caneghem

Janvier 2004

UE3 : Algorithme et Complexité #6a – ©2004 – Michel Van Caneghem

## Pourquoi s'intéresser à la Biologie Moléculaire

Dans le domaine de la biologie moléculaire on assiste à une explosion des données provenant de séquençage de génomes complets (plus de 3000 animaux).

Quelques tailles de génome (voir pages web) :

- ★ Escherichia coli 4,6 Mb,
- ★ Drosophila melanogaster 137 Mb,
- ★ Homme 3000 Mb,
- ★ Tulipe 30 Gb.,
- ★ Amoeba dubia 670 Gb.

Consultez les pages web des Options de Biologie moléculaire(I et II) de la Licence/maitrise d'Informatique :

UE3 : Algorithme et Complexité #6a – ©2004 – Michel Van Caneghem

1

## Pourquoi s'intéresser à la Biologie Moléculaire (2)

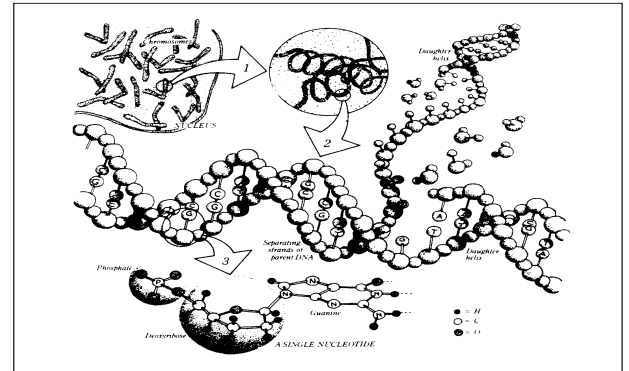
Le domaine de la biologie moléculaire est en pleine révolution. Au coeur de cette révolution, l'informatique joue un rôle central pour :

- ✓ acquisition des données (décoder les régions importantes des génomes)
- ✓ Stockage et diffusion des données
- ✓ Représentation des connaissances
- ✓ exploitation et interprétation des données

UE3 : Algorithme et Complexité #6a – ©2004 – Michel Van Caneghem

2

## Le génome



UE3 : Algorithme et Complexité #6a – ©2004 – Michel Van Caneghem

3

## Le génome (2)

L'ADN est le support de l'hérédité donc de l'information. Il se trouve dans la cellule, principalement dans les **chromosomes**. L'ADN code dans la cellule tous les objets biologiques actifs (protéines, ARN, ...). C'est un peu le programme !!!

L'information de l'ADN est codée avec 4 **bases**. En fait le génome est une grande molécule de plusieurs mètres (2m pour l'homme) composée d'Adénine, Cytosine, Guanine et Thymidine. On peut considérer que c'est un très grand texte sur un vocabulaire de 4 lettres ACGT.

UE3 : Algorithme et Complexité #6a – ©2004 – Michel Van Caneghem

4

## Le génome (3)

Le génome d'une bactérie (Escherichia Coli) à 4.6 Millions de bases et celui de l'homme est composé d'environ 3 Milliards bases [3 Goctets!!], réparties en 23 segments linéaires : les **chromosomes**.

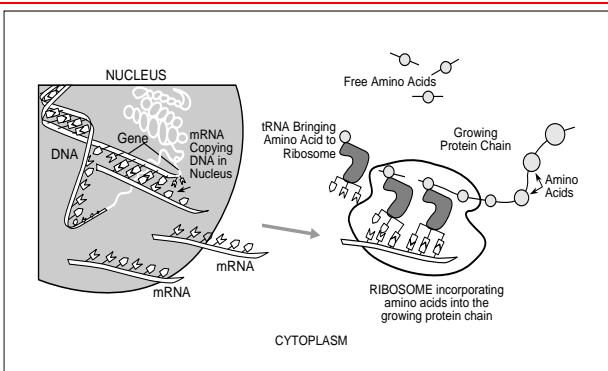
Le génome est composé de parties appelées **gènes** qui codent les composant de bases de la vie, comme les **protéines** qui sont composées **d'acides aminés**.

On peut considérer la cellule comme une usine pour produire des objets (les protéines par exemple) à partir d'un plan : les gènes

UE3 : Algorithme et Complexité #6a – ©2004 – Michel Van Caneghem

5

## Le dogme central



UE3 : Algorithme et Complexité #6a – ©2004 – Michel Van Caneghem

6

## Les protéines

Ce sont de longues molécules composées des 20 acides aminés (Vocabulaire de 20 lettres). Les protéines qui sont composées en moyenne de 300 acides aminés sont les blocs de base de la vie.

Elles sont bâties à partir de l'ADN (ARN), en groupant les bases 3 par 3 (**le code génétique universel**), de manière très précise. (un exemple : La drépanocytose est causée par le simple remplacement d'un A par un T – hémoglobine)

La seule suite des acides aminés permet de construire la protéine.

UE3 : Algorithme et Complexité #6a – ©2004 – Michel Van Caneghem

7

## Les 20 acides aminés

		2nd base in codon					
		U	C	A	G		
1st base in codon	U	Phe Phe Leu Leu	Ser Ser Ser Ser	Tyr Tyr <b>STOP</b> <b>STOP</b>	Cys Cys <b>STOP</b> Trp	U C A G	3rd base in codon
	C	Leu Leu Leu Leu	Pro Pro Pro Pro	His His Gln Gln	Arg Arg Arg Arg	U C A G	
	A	Ile Ile Ile Met	Thr Thr Thr Thr	Asn Asn Lys Lys	Ser Ser Arg Arg	U C A G	
	G	Val Val Val Val	Ala Ala Ala Ala	Asp Asp Glu Glu	Gly Gly Gly Gly	U C A G	

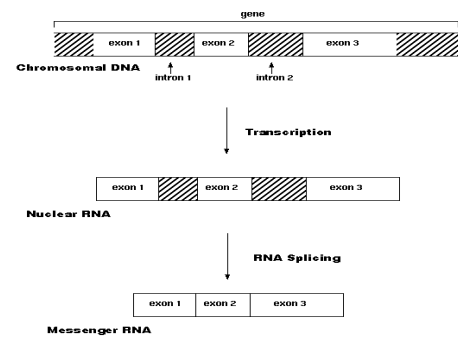
### The Genetic Code

UE3 : Algorithme et Complexité #6a – ©2004 – Michel Van Caneghem

8

## Les exons et les introns

### Transcription of DNA to Messenger RNA



UE3 : Algorithme et Complexité #6a – ©2004 – Michel Van Caneghem

9

## Les gènes

Parmi les 3 Milliards de bases du génome humain, environ 5% sont codant (environ 30 000 gènes). On ne connaît pas le rôle du reste (en particulier des **introns**).

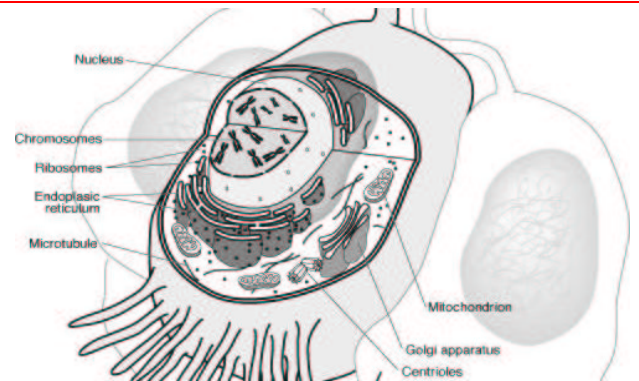
Les gènes des bactéries qui n'ont pas d'introns ont une taille moyenne de 1000 bases.

Les gènes des génomes eucaryotes (avec un noyau) comme dans le génome humain, ont une taille variable, à cause de la présence d'introns (entre 100 et 100000 bases). Les **exons** (la partie codante) sont de la même taille que pour les bactéries. C'est un gros problème de d'identifier les exons.

UE3 : Algorithme et Complexité #6a – ©2004 – Michel Van Caneghem

10

## La cellule



UE3 : Algorithme et Complexité #6a – ©2004 – Michel Van Caneghem

11

## Qu'est ce que la BioInformatique ?

- ♦ Algorithmes sur les séquences d'ADN et de protéines (Alignements), détection des zones codantes,
- ♦ Les banques de données de séquences (XML)
- ♦ Modélisation des structures des protéines
- ♦ Arbres phylogéniques

mais aussi

- ★ Les automates et les machines de Turing (le jeu de la vie)
- ★ La vie artificielle.
- ★ Les ordinateurs biologiques
- ★ Les nanotechnologies...

UE3 : Algorithme et Complexité #6a – ©2004 – Michel Van Caneghem

12

## UE3 : Alignements de séquences

Michel Van Caneghem

Janvier 2004

UE3 : Alignements de séquences #6b – ©2004 – Michel Van Caneghem

## Un peu de vocabulaire

- + **chaîne** : liste ordonnée de caractères ou symboles
- + **séquence** : synonyme de chaîne
- + **sous-chaîne** : Une sous-chaîne  $x$  d'une chaîne  $s$  est une chaîne formée de caractères consécutifs de  $s$ . (Exemple : TAC est une sous chaîne de AGTACA, mais pas GAC)
- + **sous-séquence** : Une sous séquence d'une séquence  $s$  est obtenue en enlevant des caractères de  $s$ . (Exemple : GAC est une sous séquence de AGTACA)

Je me suis servi du livre : *Introduction to Computational Molecular Biology* de J.C. Setubal et J. Meidanis (ISBN 0534952623) – 50 € chez Lavoisier

UE3 : Alignements de séquences #6b – ©2004 – Michel Van Caneghem

1

## Intérêt des alignements en biologie

C'est un des outils les plus utilisés en BioInformatique.

**Des séquences similaires ont souvent des origines ou des fonctions similaires.** On peut comparer soit des séquences d'ADN, soit des séquences de protéines.

Attention : comme il y a eu des mutations, des évolutions, ... il y a des "insertions" et des "suppressions" de symboles dans les séquences d'ADN : donc ce ne sont jamais des comparaisons exactes.

UE3 : Alignements de séquences #6b – ©2004 – Michel Van Caneghem

2

## Un problème connu en Informatique

Il s'agit de l'utilitaire diff qui compare 2 fichiers.

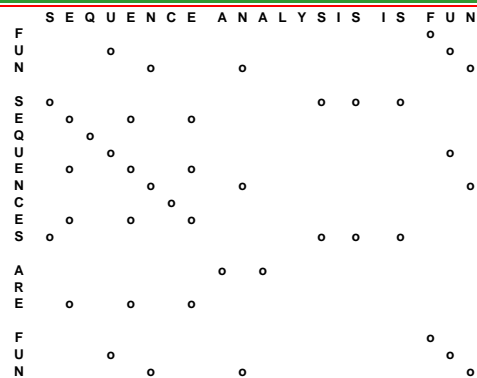
```
diff -y seq1.txt seq2.txt
```

```
A <
C <
G <
C      C
      > A
T      T
G      G
      > T
```

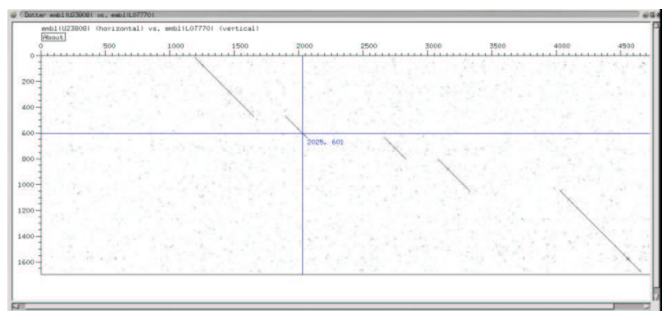
On peut aussi trouver les commandes pour passer de la séquence 1 à la séquence 2

```
6a
T
.
      4a
      A
      .
      1,3d
```

## Dot matrices



## Dot matrices (2)



## Alignement global

Considérons par exemple les deux séquences :  $A = \text{ACGCTG}$  et  $B = \text{CATGT}$  et cherchons à voir comment les aligner en introduisant des "gap" (ou quel est le nombre minimal d'opération de suppression ou d'insertion de caractères pour passer de l'une à l'autre). Par exemple

```
AC--GCTG
-CATG-T-
```

Il faut un critère de jugement du meilleur alignement possible. Il faut donc définir un score pour un alignement.

## Alignement global (2)

Par exemple si :

- ☆ Une égalité vaut 2
- ☆ Une différence vaut -1
- ☆ Un gap vaut -1

alors l'alignement :

```
- A C G C T G
C A T G - T -
-1 2 -1 2 -1 2 -1 ==> 2
```

Il faut s'intéresser également au score des différences entre deux lettres (certaines lettres peuvent être plus proche les unes des autres).

## Alignement global (3)

On cherche à aligner de manière optimale deux chaînes de caractères :  $S$  et  $T$ .

On notera  $\sigma(a, b)$  le score obtenu en alignant le caractère  $a$  avec le caractère  $b$ . Le score du gap est  $\sigma(*, -) = \sigma(-, *) = -gap$

Soit  $V(i, j)$  le score optimal de l'alignement de  $S_1 \dots S_i$  avec  $T_1 \dots T_j$  avec ( $0 \leq i \leq n, 0 \leq j \leq m$ ).

**ATTENTION** le premier caractère de chaque chaîne est un caractère rajouté, par exemple "-", donc, le tableau  $V$  a comme taille  $(n + 1) \times (m + 1)$ . Alors  $V$  a les propriétés suivantes :

## Algorithme global avec gap constant

On peut remarquer (*principe de la programmation dynamique*) que si veut calculer l'alignement optimal entre  $S_1 \dots S_i$  avec  $T_1 \dots T_j$  alors on sait qu'il provient :

- soit d'un alignement optimal entre  $S_1 \dots S_{i-1}$  et  $T_1 \dots T_{j-1}$  auquel on a rajouté  $S_i$  et  $T_j$ .
- soit d'un alignement optimal entre  $S_1 \dots S_{i-1}$  et  $T_1 \dots T_j$  auquel on a rajouté  $S_i$  et un gap après  $T_j$ .
- soit d'un alignement optimal entre  $S_1 \dots S_i$  et  $T_1 \dots T_{j-1}$  auquel on a rajouté gap après  $S_i$  et  $T_j$ .

Il suffit alors de prendre le meilleur des trois. On aboutit aux formules :

## Algorithme avec gap constant (2)

$$\begin{aligned}
 V(0, 0) &= 0 \\
 V(i, 0) &= \sum_{k=1}^i \sigma(S_k, -) \\
 V(0, j) &= \sum_{k=1}^j \sigma(-, T_k) \\
 V(i, j) &= \max \begin{cases} V(i-1, j-1) + \sigma(S_i, T_j) \\ V(i-1, j) + \sigma(S_i, -) \\ V(i, j-1) + \sigma(-, T_j) \end{cases}
 \end{aligned}$$

### Algorithme avec gap constant (3)

A = ACGCTG,      égalité = 2, diff = -1  
 B = CATGT          gap = -1

	C	A	T	G	T
	0	-1	-2	-3	-4
A	-1	-1	1	0	-1
C	-2	1	0	0	-1
G	-3	0	0	-1	2
C	-4	-1	-1	-1	1
T	-5	-2	-2	1	0
G	-6	-3	-3	0	3

### Algorithme avec gap constant (4)

La complexité pour calculer le score est  $O(mn)$  en temps et  $O(mn)$  en mémoire (il est facile de voir que l'on a besoin que d'une ligne à la fois donc complexité  $O(m)$  en mémoire).

Pour reconstruire l'alignement il suffit de partir de  $V(m, n)$  et de remonter en arrière pour chercher d'où vient la valeur que l'on a obtenue et ainsi de suite de manière récursive. La complexité est de  $O(m + n)$  si on connaît la matrice.

Si on aligne deux séquences de 10000 éléments alors il faut une matrice de 100 000 000 d'éléments ce qui pose un problème. Hors nous avons besoin de cette matrice pour reconstruire l'alignement. Il y a une méthode que l'on verra en TD pour avoir une complexité pour l'espace seulement en  $O(m)$ .

### Algorithme avec gap constant (5)

\_ACGCTG  
 \_|\_|\_  
 CATG\_T\_

ACGCTG\_  
 \_|\_|\_  
 \_C\_ATGT

ACGCTG\_  
 \_|\_|\_  
 \_CA\_TGT

### Algorithme local

(pas à programmer pour le devoir) Etant donné deux séquences, on cherche 2 sous-séquences ayant le score maximal. En fait on s'aperçoit que l'on peut reprendre le même programme que le précédent avec quelques modifications :

- On cherche le meilleur score dans toute la matrice
- dans la formule qui calcule  $V(i, j)$  il faut rajouter une quatrième ligne au  $max$  : 0
- la première ligne et la première colonne sont initialisées à zéro.

Dans ce cas l'entrée  $V(i, j)$  correspond au meilleur alignement possible d'un suffixe de  $S[1..i]$  et d'un suffixe de  $T[1..j]$ .

### Algorithme semi-global

(à programmer pour le devoir) Voici deux alignements de séquences :

CAGCA-CTTGGATTCTCGG      CAGCA-CTTGGATTCTCGG  
 ---CAGCGTGG-----      CAGC-----G-T---GG

Le premier est ce que l'on souhaite, le second est optimal. Conclusion : on souhaiterait que les gaps en tête et en queue de séquence ne comptent pas. Il suffit de modifier légèrement le programme d'alignement global :

- ★ le score est le maximum dans la dernière ligne et la dernière colonne.
- ★ Il faut initialiser la matrice de scores pour la première ligne et la première colonne à zéro

### Algorithme avec pénalité de gap quelconque

Quand il y a dans le génome des suppressions ou des inclusions, cela ne marche pas par unité, mais plutôt par bloc. Une suite de  $n$  gaps devrait donc être beaucoup moins pénalisante, que  $n$  gaps individuels. On va donc introduire une fonction de pénalité qui dépend de la longueur du gap.

Si cette fonction est quelconque, alors la programmation dynamique donne un algorithme dont la complexité est de l'ordre de  $O(n^3)$  comparaisons.

Si cette fonction est linéaire (ou plutôt affine), alors on garde une complexité en  $O(n^2)$ . Mais pour cela il faut introduire deux matrices supplémentaires.

### Algorithme avec gap linéaire

(a programmer) On cherche à aligner de manière optimale deux chaînes de caractères :  $S$  et  $T$ .

On notera  $\sigma(a, b)$  le score obtenu en alignant le caractère  $a$  avec le caractère  $b$ .

On notera  $g(i) = W_g + iW_s$  le cout d'un gap de longueur  $i$ . Ce cout est bien sur négatif ( $W_g$  et  $W_s$  sont négatifs).

Soit  $V(i, j)$  le score optimal de l'alignement de  $S_1 \dots S_i$  avec  $T_1 \dots T_j$  avec  $(0 \leq i \leq n, 0 \leq j \leq m)$ . On ajoute deux tableaux  $E(i, j)$  (gap verticaux) et  $F(i, j)$  (gap horizontaux).

Alors  $G, E, F$  ont le sens suivant :

### Algorithme avec gap linéaire (2)

\*  $G(i, j)$  : le meilleur alignement entre  $S[1..i]$  et  $T[1..j]$  avec  $S(i)$  qui est aligné avec  $T(j)$ .

S.....Si  
 T.....Tj

\*  $E(i, j)$  : le meilleur alignement entre  $S[1..i]$  et  $T[1..j]$  avec un gap qui suit  $S(i)$  et qui est aligné avec  $T(j)$ .

S.....Si - - -  
 T.....Tj

\*  $F(i, j)$  : le meilleur alignement entre  $S[1..i]$  et  $T[1..j]$  avec  $S(i)$  qui est aligné sur un gap après  $T(j)$ .

S.....Si  
 T.....Tj - - -

## Algorithme avec gap linéaire (3)

$$G(0, 0) = 0$$

$$G(i, 0) = -\infty \quad E(0, j) = -\infty \quad E(i, 0) = W_g + iW_s$$

$$G(0, j) = -\infty \quad F(i, 0) = -\infty \quad F(0, j) = W_g + jW_s$$

alors :

$$V(i, j) = \max\{G(i, j), E(i, j), F(i, j)\}$$

et :

$$G(i, j) = \sigma(S_i, T_j) + \max \begin{cases} G(i-1, j-1) \\ E(i-1, j-1) \\ F(i-1, j-1) \end{cases}$$

## Algorithme avec gap linéaire (4)

$$E(i, j) = \max \begin{cases} E(i, j-1) + W_s \\ F(i, j-1) + W_g + W_s \\ G(i, j-1) + W_g + W_s \end{cases}$$

$$F(i, j) = \max \begin{cases} F(i-1, j) + W_s \\ E(i-1, j) + W_g + W_s \\ G(i-1, j) + W_g + W_s \end{cases}$$

et bien sûr le score max est donné par :

$$V(m, n) = \max\{E(m, n), F(m, n), G(m, n)\}$$

## PAM-250

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V
A	2																			
R	-2	6																		
N	0	0	2																	
D	0	-1	2	4																
C	-2	-4	-4	-5	12															
Q	0	1	1	2	-5	4														
E	0	-1	1	3	-5	2	4													
G	1	-3	0	1	-3	-1	0	5												
H	-1	2	2	1	-3	3	1	-2	6											
I	-1	-2	-2	-2	-2	-2	-3	-2	5											
L	-2	-3	-3	-4	-6	-2	-3	-4	-2	2	6									
K	-1	3	1	0	-5	1	0	-2	0	-2	-3	5								
M	-1	0	-2	-3	-5	-1	-2	-3	-2	4	0	6								
F	-4	-4	-4	-6	-4	-5	-5	-5	-2	1	2	-5	0	9						
P	1	0	-1	-1	-3	0	-1	-1	0	-2	-3	-1	-2	-5	6					
S	1	0	1	0	0	-1	0	1	-1	-1	-3	0	-2	-3	1	2				
T	1	-1	0	0	-2	-1	0	0	-1	0	-2	0	-1	-3	0	1	3			
W	-6	2	-4	-7	-8	-5	-7	-7	-3	-5	-2	-3	-4	0	-6	-2	-5	17		
Y	-3	-4	-2	-4	0	-4	-4	-5	0	-1	-4	-2	7	-5	-3	-3	0	10		
V	0	-2	-2	-2	-2	-2	-1	-2	4	2	-2	2	-1	-1	0	-6	-2	4		

## BLAST

**Basic Local Alignment Search Tool** : Etant donné une séquence, le but est de trouver rapidement des séquences similaires dans une grande base de données de séquences.

Les méthodes d'alignement précédentes sont trop coûteuses.

Pour cela on recherche des sous-mots (sans-gaps) dans la séquence initiale ayant le plus grand score possibles d'alignement avec des mots appartenant à des séquences de la base de données et ceci avec une méthode heuristique.

## BLAST(2)

1. A partir de la séquence initiale on construit la liste de tous les mots de longueur  $w$  (par exemple 4 pour les protéines) ayant au moins un score de  $T$  avec un mots de longueur  $w$  de la séquence initiale.
  2. Puis on cherche dans la base de donnée toutes les séquences ayant un mot de longueur  $w$  appartenant a cette liste (une méthode est d'utiliser une table de hash-code).
  3. Chaque fois que l'on a trouvé un tel mot on essaye de l'étendre à gauche et a droite pour améliorer son score.
  4. On ne garde que les meilleurs scores
- Un analyse statistique sérieuse permet de ne garder que ce qui est statistiquement significatif.

## Le Devoir 3 : la méthode de Hirschberg

L'objectif de ce devoir est de faire un programme qui permet de comparer deux séquences d'ADN ("human chromosome 12p13 (222930 bp) and its syntenic region in mouse chromosome 6" (227538 bp)).

l'algorithme classique construit autour de la programmation dynamique ne peut pas marcher dans de tels cas. En effet il utilise un capacité mémoire de l'ordre de  $O(n^2)$  si  $n$  est la taille des chaînes à comparer. Dans le cas qui nous intéresse,  $n$  vaut environ 230000. Il faut donc une mémoire de  $230000 * 230000 * 4 = 211600$  Moctets = 211 Go.

J'ai donc choisit d'utiliser un algorithme basé sur la méthode de Hirschberg qui utilise un espace mémoire en  $O(n)$  pour un temps de calcul un peu plus lent que l'algorithme classique. La description de l'algorithme est ici :

.../~vancan/mait/documents/adn/page58-61.pdf