Autour du séquençage du génome

Pierrick Bouttier
groupe Algorithmique de l'Irem d'Aix-Marseille

octobre 2011

Table des matières

Chapitre 1

Préambule

Ce document se propose d'aborder quelques algorithmes traitant des données alphanumériques issues de la biologie. Les connaissances requises dans ce domaine sont celles du programme de la série S.
La première partie de ce travail s'inspire largement de l'article *À la recherche de régions codantes*[1] de François Reichenmann
La deuxième partie, consacrée à la comparaison de deux séquences d'ADN, introduit à la programmation dynamique et est inspirée d'un autre article de François Reichenmann [2].

[1] `http://interstices.info/jcms/c_14434`
[2] `http://interstices.info/jcms/c_10593`

Chapitre 2

Recherche de régions codantes

2.1 Introduction

On sait depuis le début des années 1950 que l'information génétique d'un organisme vivant est « mémorisée » dans la molécule d'ADN de ses chromosomes. Cette molécule, constituée de deux brins complémentaires, est un long enchaînement de nucléotides de quatre types différents (distingués par leur base azotée : adénine, thymine, cytosine et guanine) désignés par les lettres A, T, C et G.

Séquencer le génome d'un organisme, c'est établir l'enchaînement de ces nucléotides. Le premier génome entièrement séquencé, celui de la bactérie *Haemophilus influenzae* en 1995, comportait moins de deux millions de nucléotides. Des génomes d'organismes plus évolués peuvent comporter plusieurs milliards de nucléotides.
La molécule d'ADN apparaît ainsi comme un très long texte écrit avec les quatre lettres A, T, C, G. À titre de comparaison un livre de poche de 1 000 pages comporte environ deux millions de caractères...
D'autre part, seules certaines parties de cet enchaînement, appelées *régions codantes* ont un « sens », c'est à dire, qu'elles sont susceptibles de coder une protéine.

Aujourd'hui séquencer le génome d'un organisme ne pose plus de problème technique. Les difficultés commencent avec l'« interprétation » de ce très long texte sans espace, ni point, ni virgule.

Un triplet de nucléotides, appelé codon, correspond à un acide aminé. Il y a 64 (4^3) codons possibles qui correspondent à 20 acides aminés. Des codons différents peuvent donc correspondre à un même acide aminé. C'est la lecture de ces codons dans une région codante qui permettra de déterminer la protéine représentée par cette suite de nucléotides.

La première difficulté est de reconnaître les régions codantes potentielles. Une région codante est une succession de codons non-chevauchants qui commence par un triplet ATG (appelé codon *start*) et s'achève par l'un des trois triplets TAA, TAG ou TGA (appelés codons *stop*). Une autre difficulté est apportée par le fait que le triplet de nucléotides ATG peut marquer un codon *start* mais aussi la présence de l'acide aminé méthionine.

2.2 La recherche de régions codantes

La détermination de régions codantes potentielles dans une séquence d'ADN va commencer par la recherche de deux codons *stop* « en phase », c'est à dire séparés par un nombre de nucléotides multiple de 3. On cherchera ensuite après le premier codon *stop* le premier codon *start* en phase avec ces codons *stop*. Cette recherche conduit à prédire la présence d'une région codante. Dans la pratique, on accumule d'autres indices (longueur de la région codante, présence de motifs connus avant le codon *start*, etc...), pour conforter cette hypothèse. Pour une séquence d'ADN donnée, il y a trois façons différentes de grouper les nucléotides par trois selon que l'on commence par le premier élément, le deuxième ou le troisième. De plus, ne sachant duquel des deux brins complémentaires de la molécule d'ADN cette séquence est issue, il faut considérer la lecture de la séquence de gauche à droite et de droite à gauche.

2.3 Un algorithme de recherche d'une séquence codante potentielle

Dans cette partie, nous nous limiterons à la recherche d'une région codante en effectuant uniquement la lecture de gauche à droite.
La séquence d'ADN est codée sous la forme d'un mot de l'alphabet $\{A, T, C, G\}$. On note N la longueur de cettre séquence et on suppose qu'elle est représentée en mémoire dans un tableau unidimensionnel appelé Seq.
La recherche va se faire en trois étapes :

1. recherche d'un codon *stop*
2. recherche d'un deuxième codon *stop* situé à droite du premier et en phase avec celui-ci (c'est à dire, rappelons-le, séparé par un nombre de nucléotides multiple de 3)
3. recherche d'un codon *start* à partir du premier codon *stop*, situé avant le deuxième codon *stop* et en phase avec ceux-ci

2.3.1 La recherche d'un codon *stop*

Rappelons qu'un codon *stop* est l'un des trois triplets TAA, TAG ou TGA. L'idée de la recherche est simple : on lit la séquence jusqu'à la rencontre d'un caractère 'T' on regarde si le caractère suivant est un 'A' ou un 'G' et, si oui, on examine le caractère suivant pour conclure ou non à la présence d'un codon *stop*. Si la réponse est négative, on reprend la lecture au caractère qui suit le caractère 'T'.

Si un codon *stop* est trouvé, l'indice du caractère 'T' est mémorisé dans la variable $DebCodStop1$.

Rappelons que N désigne la longueur de la séquence Seq à examiner. Ce qui conduit à l'algorithme suivant :

algorithme 1

```
Arret ← N − 1
i ← 1
Trouve ← faux
tant que (i < Arret) et (non Trouve) faire
    si (Seq[i] ≠ 'T')
            alors
                | i ← i + 1
            sinon
                | si (Seq[i + 1] = 'A')
                |     alors
                |         | si (Seq[i + 2] ='A') ou (Seq[i + 2] ='G')
                |         |     alors
                |         |         | Trouve ← vrai
                |         |         | DebCodStop1 ← i
                |         |     sinon
                |         |         | i ← i + 1
                |     sinon
                |         | si (Seq[i + 1] ='G') et (Seq[i + 2] ='A')
                |         |     alors
                |         |         | Trouve ← vrai
                |         |         | DebCodStop1 ← i
                |         |     sinon
                |         |         | i ← i + 1
si Trouve
    alors
        | Résultat : DebCodStop1
    sinon
        | Résultat : Aucun codon stop trouvé
```

2.3.2 Recherche d'un deuxième codon *stop* en phase avec le premier

On suppose qu'un premier codon *stop* a été trouvé dans la séquence d'ADN et que l'indice du nucléotide 'T' de ce codon est mémorisé dans la variable $DebCodStop1$. Il s'agit maintenant de trouver le premier codon *stop* en phase avec celui qui vient d'être trouvé et situé après celui-ci.

Il suffit donc d'examiner la suite de la séquence à partir de l'indice $DebCodStop1 + 3$, tous les 3 caractères, jusqu'à ce que l'on rencontre un caractère 'T' et d'examiner, comme précédemment, les deux caractères qui suivent. Ce qui conduit à l'algorithme suivant, qui est similaire au précédent dans son principe :

algorithme 2

```
Arret ← N − 1
i ← DebCodStop1 + 3
Trouve ← faux
tant que (i < Arret) et (non Trouve) faire
    si (Seq[i] ≠ 'T')
        alors
            | i ← i + 3
        sinon
            | si (Seq[i + 1] ='A')
            |     alors
            |         | si (Seq[i + 2] ='A') ou (Seq[i + 2] ='G')
            |         |     alors
            |         |         | Trouve ← vrai
            |         |         | DebCodStop2 ← i
            |         |     sinon
            |         |         | i ← i + 3
            |     sinon
            |         | si (Seq[i + 1] ='G') et (Seq[i + 2] ='A')
            |         |     alors
            |         |         | Trouve ← vrai
            |         |         | DebCodStop2 ← i
            |         |     sinon
            |         |         | i ← i + 3
si Trouve
    alors
        | Résultat : DebCodStop2
    sinon
        | Résultat : Aucun codon stop trouvé
```

2.3.3 Introduction de la notion de sous-programme

Comme cela a été souligné précédemment, les algorithmes **1** et **2** sont identiques à deux différences près :

- l'indice du caractère où débute la recherche
- la valeur dont est incrémentée la variable i.

Ces différences concernent des valeurs de variables et non pas la structure même de l'algorithme. De plus, dans l'algorithme de recherche d'une séquence codante, il pourra être nécessaire d'utliser plusieurs fois et alternativement ces deux algorithmes.

On peut concevoir un sous-programme *RechCodStop* qui prend en entrée deux entiers d et p, d indice de début de recherche et p pas d'incrémentation de la variable i, et qui, appliqué à la séquence Seq, renvoie un entier Ind égal à l'indice du début du codon *stop* trouvé s'il y en a un (i.e si $Trouve$ vaut $vrai$) et -1 sinon (cette valeur est un codage

pour signifier que la variable booléenne $Trouve$ a la valeur $faux$).

sous-programme *RechCodStop*(d,p)

$Arret \leftarrow N - 1$
$i \leftarrow d$
$Trouve \leftarrow$ faux
$Ind \leftarrow -1$
tant que ($i < Arret$) et (non $Trouve$) **faire**
  si ($Seq[i] \neq$'T')
    alors
      $i \leftarrow i + p$
    sinon
      si ($Seq[i+1] =$'A')
        alors
          si ($Seq[i+2] =$'A') ou ($Seq[i+2] =$'G')
            alors
              $Trouve \leftarrow$ vrai
              $Ind \leftarrow i$
            sinon
              $i \leftarrow i + p$
        sinon
          si ($Seq[i+1] =$'G') et ($Seq[i+2] =$'A')
            alors
              $Trouve \leftarrow$ vrai
              $Ind \leftarrow i$
            sinon
              $i \leftarrow i + p$

Résultat : Ind

Un script de ce sous programme en langage Python (version 2.6) est donné en annexe (section 4.1.1).

2.3.4 Recherche d'un codon *start* en phase avec les codons *stop*

On rappelle qu'un codon *start* est un triplet ATG. On suppose qu'on a trouvé deux codons *stop* en phase dans la séquence Seq et dont les indices de début sont respectivement $DebCodStop1$ et $DebCodStop2$. On effectue la recherche entre ces deux codons *stop*. La recherche va donc commencer à l'indice $DebCodStop1 + 3$ et s'achever à l'indice $DebCodStop2 - 3$ avec un pas de 3.

Si un codon *start* est trouvé, l'indice du caractère 'A' est mémorisé dans la variable $DebCodStart$.Ce qui conduit à l'algorithme 3.

Là encore il est préférable de concevoir un sous programme *RechCodStart* qui prend en entrée les indices $d1$ et $d2$ des codons *stop* en phase qui « délimitent » la partie de la

séquence dans laquelle va s'effectuer la recherche et qui renvoie l'indice Ind du début du codon *start* trouvé si $Trouve$ vaut *vrai* et -1 sinon.

algorithme 3

$i \leftarrow DebCodStop1 + 3$
$Arret \leftarrow DebCodStop2 - 2$
$Trouve \leftarrow$ faux
tant que $(i < Arret)$ et (non $Trouve$) **faire**
  si $(Seq[i] = 'A')$ et $(Seq[i+1] = 'T')$ et $(Seq[i+2] = 'G')$
    alors
      $Trouve \leftarrow$ vrai
      $DebCodStart \leftarrow i$
    sinon
      $i \leftarrow i + 3$

si $Trouve$
  alors
    Résultat : $DebCodStart$
  sinon
    Résultat : *Aucun codon start trouvé*

sous-programme $RechCodStart(d1,d2)$

$i \leftarrow d1 + 3$
$Arret \leftarrow d2 - 2$
$Trouve \leftarrow$ faux
$Ind \leftarrow -1$
tant que $(i < Arret)$ et (non $Trouve$) **faire**
  si $(Seq[i] = 'A')$ et $(Seq[i+1] = 'T')$ et $(Seq[i+2] = 'G')$
    alors
      $Trouve \leftarrow$ vrai
      $Ind \leftarrow i$
    sinon
      $i \leftarrow i + 3$

Résultat : Ind

Un script en langage Python est donné en annexe (section 4.1.2).

2.3.5 Recherche d'une séquence codante potentielle

L'objectif est maintenant de trouver une éventuelle séquence codante dans une séquence d'ADN donnée.

L'heuristique a été décrite dans l'introduction de la section 2.3 et consiste en :

1. recherche d'un codon *stop*
2. recherche d'un deuxième codon *stop* situé après le premier et en phase avec celui-ci (c'est à dire, rappelons-le, séparé par un nombre de nucléotides multiple de 3)
3. recherche d'un codon *start* à partir du premier codon *stop*, situé avant le deuxième codon *stop* et en phase avec ceux-ci

L'algorithme de recherche, appliqué à une séquence T donnée, utilise les sous-programmes $RechCodStop(d, p)$ et $RechCodStart(d1, d2)$ décrits aux sections 2.3.3 et 2.3.4. La recherche s'effectue jusqu'à ce qu'une séquence codante ait été trouvée (i.e $ArretRech$ vaut $vrai$) ou que l'on atteigne la fin de la séquence d'ADN à analyser.
La borne d'arrêt est fixée à $N - 7$ pour tenir compte du fait qu'après le premier codon *stop*, on doit trouver un codon *start* et un codon *stop*.

Le résultat est soit le couple d'indices de début et de fin de la séquence codante potentielle trouvée, soit un message d'échec.

algorithme 4

$i \leftarrow 1$
$ArretRech \leftarrow$ faux
tant que $(i < N - 7)$ et (non $ArretRech$) **faire**
  $DebCodStop1 \leftarrow RechCodStop(i, 1)$
  si $(DebCodStop1 = -1)$
    alors
      $ArretRech \leftarrow$ vrai
      Résultat :*Aucune séquence codante trouvée*
    sinon
      $DebCodStop2 \leftarrow RechCodStop(DebCodStop1 + 3, 3)$
      si $(DebCodStop2 = -1)$
        alors
          $i \leftarrow i + 1$
        sinon
          $DebCodStart \leftarrow RechCodStart(DebCodStop1, DebCodStop2)$
          si $DebCodStart = -1$
            alors
              $i \leftarrow i + 1$
            sinon
              Résultat : ($DebCodStart + 3$,$DebCodStop2 - 1$)
              $ArretRech \leftarrow$ vrai

2.3.6 Recherche exhaustive des séquences codantes potentielles dans une séquence d'ADN donnée

Dans la section précédente la recherche s'arrête dès que l'on a trouvé une séquence codante potentielle (ou que l'on a atteint la fin de la séquence d'ADN sans succès).

Il s'agit maintenant de construire un algorithme qui permet de trouver toutes les séquences codantes potentielles d'une séquence d'ADN donnée.

Il suffit, dans l'algorithme 4, d'enlever la condition d'arrêt en cas de succès de la recherche et de mémoriser, sous forme de couple d'entiers, les indices de début et de fin de chaque séquence codante potentielle dans un tableau $SeqCodPot$, indexé par j. L'initialisation de ce tableau au moyen du couple $(0, 0)$ permet de savoir, le cas échéant, que la recherche a été infructueuse.

Il faut aussi, dans le cas où un premier codon *stop* a été trouvé, modifier la valeur de la variable i, afin de pouvoir reprendre la recherche ultérieurement à l'endroit convenable, en lui affectant l'indice du premier nucléotide de ce codon *stop*.

algorithme 5

$i \leftarrow 1$
$ArretRech \leftarrow$ faux
$j \leftarrow 1$
$SeqCodPot[j] \leftarrow (0, 0)$
tant que $(i < N - 7)$ et (non $ArretRech$) **faire**
  $DebCodStop1 \leftarrow RechCodStop(i, 1)$
  si $(DebCodStop1 = -1)$
    alors
      $ArretRech \leftarrow$ vrai
    sinon
      $i \leftarrow DebCodStop1$
      $DebCodStop2 \leftarrow RechCodStop(DebCodStop1 + 3, 3)$
      si $(DebCodStop2 = -1)$
        alors
          $i \leftarrow i + 1$
        sinon
          $DebCodStart \leftarrow RechCodStart(DebCodStop1, DebCodStop2)$
          si $DebCodStart \neq -1$
            alors
              $SeqCodPot[j] \leftarrow (DebCodStart + 3, DebCodStop2 - 1)$
              $j \leftarrow j + 1$
          $i \leftarrow i + 1$

Résultat : $SeqCodPot$

Un script Python est donné en annexe (section 4.1.3).

Chapitre 3

Alignement de séquences d'ADN

3.1 Introduction

Une séquence d'ADN est une succession de nucléotides[1] qui peut être représentée par un mot utilisant les lettres de l'alphabet A,T,C,G.
Comparer ou aligner deux séquences, c'est disposer l'une en dessous de l'autre ces deux chaînes de caractères, auxquelles on aura éventuellement ajouté des « trous », pour mettre en concordance autant qu'il est possible des caractères identiques.

Par exemple, pour les séquences $TCGT$ et$TAGCT$,

$$\begin{array}{ccccc} T & C & G & - & T \\ | & & | & & | \\ T & A & G & C & T \end{array} \quad \text{et} \quad \begin{array}{ccccc} T & - & C & G & - & T \\ | & & & | & & | \\ T & A & - & G & C & T \end{array}$$ sont deux alignements possibles.

Dans l'exemple ci-dessus, un trait vertical entre les deux chaînes, marque l'identité des nucléotides, l'absence de trait une mutation et un tiret une insertion ou une délétion, selon le point de vue que l'on adopte : ce qui apparaît comme une insertion pour une des séquences est une délétion pour l'autre séquence.

Cette recherche d'alignement est utilisée notamment pour déterminer la fonction d'une protéine (séquence de nucléotides) jusque là inconnue en comparant sa séquence à des séquences proches de protéines dont le rôle est connu, ou encore à construire des arbres phylogénétiques entre les espèces.

Pour deux séquences données, il existe en général plusieurs alignements possibles. En outre lorsque l'on recherche la fonction d'une protéine, on va essayer d'aligner sa séquence avec des séquences de protéines connues afin de déterminer quelle est « la plus proche » et d'en inférer la fonction.

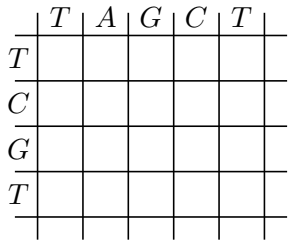
Pour cela on attribue à chaque alignement un coût total. On affecte d'abord un coût élémentaire à chacune des opérations mutation, insertion, délétion. En général le coût d'une mutation est choisi inférieur à ceux d'une insertion ou d'une délétion (qui sont les

[1] ou d'acides aminés

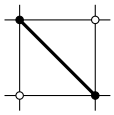
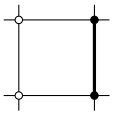
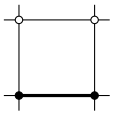
mêmes compte tenu de leur réciprocité signalée précédemment). Dans certains cas, on peut attribuer un coût différent selon le type de mutation (un coût identique pour les mutations A - T et C - G, inférieur à celui des mutations A - C, A - G, T - C et T - G. Le coût total est égal à la somme des opérations nécessaires à l'alignement. La séquence considérée comme la plus proche sera celle dont l'alignement avec la séquence donnée aura le coût le plus faible.

3.2 Représentation d'un alignement

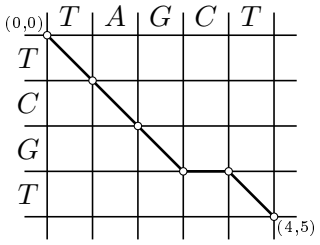
On peut représenter l'alignement d'une séquence $Seq1$ de longueur n et d'une séquence $Seq2$ de longueur p au moyen d'une grille à $(n+1)$ lignes horizontales et $(p+1)$ lignes verticales. Chaque ligne représente un caractère de la séquence $Seq1$ et chaque colonne un caractère de la séquence $Seq2$. Par exemple, pour les séquences $TCGT$ et $TAGCT$, on obtient la grille suivante :

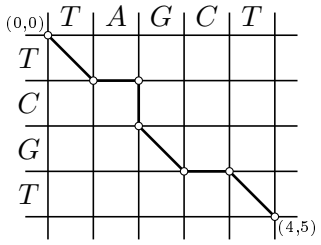


Chacune des trois opérations mutation (ou identité), délétion et insertion peut être représentée par un trait dans une cellule de la grille selon le code suivant :

identité - mutation  délétion  insertion 

Les deux alignements $\begin{array}{ccccccc} T & C & G & - & T \\ | & & | & & | \\ T & A & G & C & T \end{array}$ et $\begin{array}{ccccccc} T & - & C & G & - & T \\ | & & & | & & | \\ T & A & - & G & C & T \end{array}$ peuvent être représentés graphiquement par un chemin allant du nœud $(0,0)$ au nœud $(4,5)$:





À chaque alignement correspond un chemin du nœud $(0,0)$ au nœud (n,p) de la grille et réciproquement.

3.3 Recherche d'un alignement optimal

Dans la suite de cette section, on attribue un coût de 2 pour une mutation et un coût de 3 pour une insertion-délétion.

3.3.1 Un algorithme naïf

Une méthode pour déterminer un alignement optimal, c'est à dire au coût minimal, est de calculer le coût de chacun des alignements possibles et de retenir celui - ou ceux - qui a - ou qui ont - le coût le plus faible. En pratique, ceci n'est pas réalisable car le nombre d'alignements est trop élevé[2].

3.3.2 Une remarque sur les chemins optimaux

Considérons un chemin optimal $\mathcal{C}$ allant du nœud $(0,0)$ au nœud (r,s) d'une grille $(n+1) \times (p+1)$, où $0 < r \leq n$ et $0 < s \leq p$, et supposons qu'il soit composé de k segments horizontaux, verticaux ou en diagonale. Le chemin $\mathcal{C}_1$ composé des $(k-1)$ premiers segments du chemin $\mathcal{C}$ est lui aussi optimal.
En effet, s'il n'en était pas ainsi, il existerait un chemin $\mathcal{C}_2$, de coût strictement inférieur à celui de $\mathcal{C}_1$, joignant le nœud $(0,0)$ à l'avant dernier nœud du chemin $\mathcal{C}$. En ajoutant le coût du dernier segment du chemin $\mathcal{C}$, commun aux chemins $\mathcal{C}_1$ et $\mathcal{C}_2$, on obtiendrait un chemin allant du nœud $(0,0)$ au nœud (r,s) de coût strictement inférieur à celui du chemin $\mathcal{C}$, ce qui contredirait l'hypothèse d'optimalité faite sur ce chemin $\mathcal{C}$.

La recherche d'un chemin optimal conduisant du nœud $(0,0)$ au nœud (r,s) se ramène donc à la recherche des chemins optimaux conduisant aux nœuds permettant d'atteindre le nœud (r,s) en une étape. Cette remarque peut être appliquée, à nouveau, à ces derniers nœuds, et ainsi de suite... On pourrait calculer « récursivement » le coût minimal pour atteindre chaque nœud de la grille, mais en procédant ainsi on serait amené à calculer plusieurs fois les mêmes coûts. Pour éviter cet inconvénient et des calculs inutiles, on va utiliser l'algorithme décrit ci-après.

3.3.3 Principe d'un algorithme de recherche d'un alignement optimal

L'algorithme va se dérouler en deux temps[3] :
- une première étape où l'on va calculer le coût minimal pour atteindre chacun des nœuds de la grille à partir du nœud $(0,0)$
- une deuxième étape où l'on va déterminer l'alignement optimal entre les deux séquences en effectuant « une lecture inverse » du tableau précédent

[2]Dans le cas d'une grille $(n+1) \times (p+1)$, on montre qu'il y a $\sum_{i=0}^{min(n,p)} \frac{(n+p-i)!}{i!(n-i)!(p-i)!}$ alignements différents possibles

[3]cet algorithme est une version simplifiée de celui connu sous le nom d'algorithme de Needleman et Wunsch, élaboré en 1970 et qui s'appliquait à l'origine à l'alignement de séquences de protéines.

3.3.4 Construction de la matrice des coûts minimaux

Les coûts minimaux pour atteindre les nœuds de la grille vont être mémorisés dans une matrice C de dimension $(n+1) \times (p+1)$. Les indices seront pris à partir de 0 pour être en cohérence avec ceux des nœuds de la grille.

Chaque nœud de la grille, distinct du nœud $(0,0)$, peut être atteint en une étape à partir d'au plus trois nœuds de la grille :

figure 1

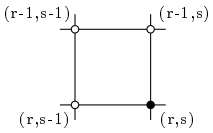


figure 2

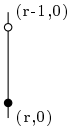


figure 3

S'il s'agit d'un nœud situé sur la première ligne horizontale de la grille (cf *figure 1*), celui-ci ne peut être atteint qu'à partir du nœud situé à sa gauche et le coût minimal pour l'atteindre sera calculé à partir du coût minimal du précédent. On procède de la même façon s'il s'agit d'un nœud situé sur la première ligne verticale de la grille (cf *figure 3*).
Sinon, si l'on note $C_{r-1,s-1}$, $C_{r-1,s}$ et $C_{r,s-1}$ les coûts optimaux pour arriver, respectivement, aux nœuds $(r-1, s-1)$, $(r-1, s)$ et $(r, s-1)$ (cf *figure 2*), le coût d'un chemin optimal pour arriver au nœud (r, s) sera égal au minimum des trois nombres $C_{r-1,s} + 3$, $C_{r,s-1} + 3$ et $C_{r-1,s-1}$ + 2 (ou + 0 si l'on a affaire à une identité de nucléotides).

On va donc calculer, ligne par ligne, le coût optimal pour atteindre chacun des nœuds de la grille en partant du nœud $(0,0)$ après avoir complété la première ligne et la première colonne de la matrice.

Pour faciliter la recherche d'un chemin optimal à partir de la matrice C au cours de la seconde étape de l'algorithme, on construit un deuxième tableau $Chem$ de mêmes dimensions où l'on va mémoriser dans la cellule $Chem_{r,s}$, au moyen d'un code, l'adresse du nœud dont le coût minimal a permis de calculer le coût minimal pour atteindre le nœud (r, s).
Précisément :

- $Chem_{0,0}$ = 'o'
- $Chem_{r,s}$ = 'i', pour insertion, si $C_{r,s} = C_{r,s-1} + 3$
- $Chem_{r,s}$ = 'd', pour délétion, si $C_{r,s} = C_{r-1,s} + 3$
- $Chem_{r,s}$ = 'm', pour mutation ou identité de nucléotides, sinon

Un algorithme de construction de la matrice C et du tableau $Chem$ est décrit ci-dessous (les séquences $Seq1$ et $Seq2$ sont mémorisées dans des tableaux à une dimension R et S) et la variable mut contient la valeur 0 ou la valeur 2 selon que les caractères à aligner sont identiques ou sont différents.

algorithme 6

$C_{0,0} \leftarrow 0$
$Chem_{0,0} \leftarrow$ 'o'
pour j **de** 1 **à** p **faire**
  $C_{0,j} \leftarrow 3 \times j$
  $Chem_{0,j} \leftarrow$ 'i'
pour i **de** 1 **à** n **faire**
  $C_{i,0} \leftarrow 3 \times i$
  $Chem_{i,0} \leftarrow$ 'd'
pour i **de** 1 **à** n **faire**
  pour j **de** 1 **à** p **faire**
    si $(R[i] = S[j])$
      alors
        $mut \leftarrow 0$
      sinon
        $mut \leftarrow 2$
    Calcul de $C_{i,j}$ et de $Chem_{i,j}$
Résultat : C et $Chem$

Le calcul de $C_{i,j}$ et celui de $Chem_{i,j}$ se fait ainsi :

si $C_{i-1,j-1} + mut \leq C_{i,j-1} + 3$
  alors
    si $C_{i-1,j-1} + mut \leq C_{i-1,j} + 3$
      alors
        $C_{i,j} \leftarrow C_{i-1,j-1} + mut$
        $Chem_{i,j} \leftarrow$ 'm'
      sinon
        $C_{i,j} \leftarrow C_{i-1,j} + 3$
        $Chem_{i,j} \leftarrow$ 'd'
  sinon
    si $C_{i,j-1} \leq C_{i-1,j}$
      alors
        $C_{i,j} \leftarrow C_{i,j-1} + 3$
        $Chem_{i,j} \leftarrow$ 'i'
      sinon
        $C_{i,j} \leftarrow C_{i-1,j} + 3$
        $Chem_{i,j} \leftarrow$ 'd'

Remarque : les inégalités dans les tests de comparaison ont été choisies de façon à privilégier les mutations (ou identités de nucléotides) sur les délétions et insertions, et les insertions sur les délétions.
Un script Python (sous forme de fonction *constrmat*) de cet algorithme est donné en annexe(section 4.2.1).

Cet algorithme appliqué aux séquences $TCGT$ et$TAGCT$ produit les matrices C et $Chem$ ci-dessous :

0	3	6	9	12	15
3	0	3	6	9	12
6	3	2	5	6	9
9	6	5	2	5	8
12	9	8	5	4	5

o	i	i	i	i	i
d	m	i	i	i	m
d	d	m	m	m	i
d	d	m	m	i	m
d	m	m	d	m	m

3.3.5 Détermination d'un alignement optimal par « lecture inverse » de la matrice des coûts

Les matrices C et $Chem$ étant calculées, il s'agit maintenant de déterminer un alignement optimal par « lecture inverse » de la matrice $Chem$.

Les séquences alignées vont être mémorisées dans deux tableaux à une dimension $SeqR$ et $SeqS$. Rappelons que les séquences alignées sont composées des mêmes lettres que les séquences à aligner et, éventuellement, de trous dans l'une et/ou l'autre séquence. Ces tableaux ont une même longueur, variable selon les séquences à aligner, comprise entre $max(n, p)$ et $n + p$.

Le principe en est le suivant : supposons, qu'après k étapes de la « lecture inverse », on soit arrivé dans la cellule $Chem_{i,j}$ du tableau $Chem$. On examine le caractère contenu dans celle-ci :

- s'il s'agit d'un 'm', le i^{ieme} caractère de la séquence R est aligné avec le j^{ieme} caractère de la séquence S (autrement dit $SeqR[k+1]$ prend la valeur $R[i]$ et $SeqS[k+1]$ prend la valeur $S[j]$) et on va à la cellule $Chem_{i-1,j-1}$ du tableau $Chem$ (déplacement en diagonale correspondant à une mutation)

- s'il s'agit d'un 'd', le i^{ieme} caractère de la séquence R est aligné avec un '$-$' et on va à la cellule $Chem_{i-1,j}$ du tableau $Chem$ (déplacement vertical correspondant à une délétion)

- s'il s'agit d'un 'i', le j^{ieme} caractère de la séquence S est aligné avec un '$-$' et on va à la cellule $Chem_{i,j-1}$ du tableau $Chem$ (déplacement horizontal correspondant à une insertion)

L'algorithme commence par l'examen de la cellule $Chem_{n,p}$.

Une fois la « lecture inverse terminée », les séquences contenues dans les tableaux $SeqR$ et $SeqS$ sont rangées « à l'envers ». Il faut donc en renverser l'ordre des caractères avant de les afficher.

Ces indications conduisent à l'algorithme suivant :

algorithme 7

$i \leftarrow n$
$j \leftarrow p$
$k \leftarrow 1$
tant que $i \neq 0$ ou $j \neq 0$
  si $Chem_{i,j} =$ 'm'
    alors
      $SeqR[k] \leftarrow R[i]$
      $SeqS[k] \leftarrow S[j]$
      $i \leftarrow i - 1$
      $j \leftarrow j - 1$
    sinon
      si $Chem_{i,j} =$ 'i
        alors
          $SeqR[k] \leftarrow$ '-'
          $SeqS[k] \leftarrow S[j]$
          $j \leftarrow j - 1$
        sinon
          $SeqR[k] \leftarrow R[i]$
          $SeqS[k] \leftarrow$ '-'
          $i \leftarrow i - 1$
  $k \leftarrow k + 1$
Renverser l'ordre des caractères de $SeqR$ et $SeqS$
Résultat : $SeqR$ et $SeqS$

Un script Python de cet algorithme, utilisant la fonction *constrmat* est donné en annexe(section 4.2.2).

Dans l'exemple précédent, on va parcourir succcessivement les cases en gras du tableau ci-dessous en partant de l'angle inférieur droit, ce qui conduit, après inversion de l'ordre des nucléotides, à l'alignement optimal suivant :

0	i	i	i	i	i
d	**m**	i	i	i	m
d	d	**m**	m	m	i
d	d	m	**m**	**i**	m
d	m	m	d	m	**m**

$$\begin{array}{ccccc} T & C & G & - & T \\ | & & | & & | \\ T & A & G & C & T \end{array}$$

Chapitre 4

Annexes

On trouvera dans les pages qui suivent les scripts en langage Python (version 2.6) des algorithmes décrits précédemment.

4.1 Recherche de régions codantes

4.1.1 Sous programme de recherche d'un codon *stop*

Cette fonction[1] est appelée au moyen de l'instruction $RechCodStop(T, N, d, p)$.

```
# -*- coding: utf-8 -*-
# T désigne la sequence d'ADN à explorer
# N désigne la longueur de la séquence T
# d désigne l'indice de début de recherche
# p désigne le pas de lecture de la séquence (1 ou 3)
# Cette fonction retourne l'indice du premier caractère
#du codon start trouvé s'il existe et -1 sinon.
#
def RechCodStop(T,N,d,p):
     Arret=N-2
     i=d
     Trouve=False
     Ind=-1
     while i<Arret and not Trouve:
          if T[i]!='t':
               i=i+p
          else:
               if T[i+1]=='a':
                    if T[i+2]=='a' or T[i+2]=='g':
                         Trouve=True
                         Ind=i
                    else
                         i=i+p
               else:
                    if T[i+1]=='g' and T[i+2]=='a':
                         Trouve=True
                         Ind=i
                    else:
                         i=i+p
     return Ind
```

[1] en langage Python les sous programmes sont nommés fonctions

4.1.2 Sous programme de recherche d'un codon *start*

Cette fonction est appelée par la commande $RechCodStart(T, d1, d2)$.

```
# -*- coding: utf-8 -*-
# T désigne la séquence d'ADN à explorer
# d1 désigne l'indice de début du premier codon stop
# d2 désigne l'indice de début du deuxième codon stop
# Cette fonction retourne l'indice du premier caractère
# du codon start trouvé s'il existe et -1 sinon.
#
def RechCodStart(T,d1,d2):
    i=d1+3
    Arret=d2-2
    Trouve=False
    Ind=-1
    while i<Arret and not Trouve:
         if T[i]=='a' and T[i+1]=='t' and T[i+2]=='g':
              Trouve=True
              Ind=i
         else:
              i=i+3
    return Ind
```

4.1.3 Recherche exhaustive des séquences codantes potentielles

Voici le script en langage Python de l'algorithme qui utilise les deux fonctions précédentes.

```
# -*- coding: utf-8 -*-
# recherche exhaustive des séquences codantes potentielles
# j représente le nombre de séquences trouvées
#
from RechCodStop import*
from RechCodStart import*
#
# entrer le nom du fichier texte  entre '  '  et avec l'extension .txt
#
fichier=input('nom du fichier à  analyser:')
ofi=open(fichier,'r')
Seq= ofi.read()
N=len(Seq)
i=0
ArretRech=False
j=0
SeqCodPot=[]
while i<N-8 and not ArretRech:
     d1=RechCodStop(Seq,N-9,i,1)
     if d1==-1:
         ArretRech=True
     else:
         i=d1
         d2=RechCodStop(Seq,N,d1+3,3)
         if d2==-1:
             i=i+1
         else:
             d3=RechCodStart(Seq,d1,d2)
             if d3!=-1:
                 j=j+1
                 SeqCodPot.append((d3+3,d2-1))
                 i=i+1
if j==0 :
     print 'Aucune séquence codante potentielle trouvée'
else:
     for i in range(1,j+1):
         l=SeqCodPot[i][1]-SeqCodPot[i][0]+1
         print SeqCodPot[i][0],SeqCodPot[i][1],'longueur de la CDS: ',l
         # CDS pour Coding Sequence
print 'nombre de séquences trouvées: ',j
```

4.2 Alignement de séquences d'ADN

4.2.1 Construction de la matrice des coûts minimaux

Script Python de la fonction *constrmat(X,x,Y,y)* :

```
    # -*- coding: utf-8 -*-
# X première séquence, x longueur de la séquence X
# Y deuxième séquence, y longueur de la deuxième séquence
def constrmat(X,x,Y,y):
# création des matrices M et Chem
    M=[0]*(x+1)
    Chem=[0]*(x+1)
    for j in range(x+1):
        M[j]=[0]*(y+1)
        Chem[j]=[0]*(y+1)
# initialisation de la première ligne
    for j in range(1,y+1):
        M[0][j]=3*j
        Chem[0][j]='i'
# initialisation de la première colonne
    for i in range(1,x+1):
        M[i][0]=3*i
        Chem[i][0]='d'
# construction de la matrice
    for i in range(1,x+1):
        for j in range(1,y+1):
            if X[i-1]==Y[j-1]:
                mut=0
            else:
                mut=2
            if M[i-1][j-1]+mut<=M[i][j-1]+3:
                if M[i-1][j-1]+mut<= M[i-1][j]+3:
                    M[i][j]=M[i-1][j-1]+mut
                    Chem[i][j]='m'
                else:
                    M[i][j]=M[i-1][j]+3
                    Chem[i][j]='d'
            else:
                if M[i][j-1]<=M[i-1][j]:
                    M[i][j]=M[i][j-1]+3
                    Chem[i][j]='i'
                else:
                    M[i][j]=M[i-1][j]+3
                    Chem[i][j]='d'
    return [M,Chem]
```

4.2.2 Détermination d'un alignement optimal par lecture inverse de la matrice des coûts

Script Python de l'algorithme de recherche d'un alignement optimal :

```
# -*- coding: utf-8 -*-
# algorithme alignement global
from math import*
from constrmat import*
#
R=input('séquence 1:')
S=input('séquence 2:')
n=len(R)
p=len(S)
print R, S
# calcul de la matrice des coûts MatCouts et
# de la matrice Chem
MatCouts=constrmat(R,n,S,p)[0]
Chem=constrmat(R,n,S,p)[1]
# calcul d'un chemin optimal
SeqR=[]
SeqS=[]
Symb=[] # liste de symboles d'alignement
j=p
i=n
while Chem[i][j]!=0:
    if Chem[i][j]=='m':
         if MatCouts[i][j]==MatCouts[i-1][j-1]:
              Symb.insert(0,'|') # caractères identiques
         else:
              Symb.insert(0,' ')
         SeqR.insert(0,R[i-1])
         SeqS.insert(0,S[j-1])
         i=i-1
         j=j-1
    else:
         if Chem[i][j]=='i':
              SeqR.insert(0,'-')
              SeqS.insert(0,S[j-1])
              Symb.insert(0,' ')
              j=j-1
         else:
              SeqR.insert(0,R[i-1])
              SeqS.insert(0,'-')
              Symb.insert(0,' ')
              i=i-1
```

```
# impression de la matrice des coûts et de l'alignement
for i in range(n+1):
    print MatCouts[i]
    print Chem[i]
print ' ' # saut de ligne
print SeqR
print Symb
print SeqS
```

Remarque : le programme comporte la construction d'un troisième tableau *Symb* contenant les symboles d'alignement pour améliorer la lecture du résultat obtenu, ainsi que l'édition des matrices *MatCouts* et *Chem*.