Alignement optimal de séquences ADN

Thèmes : algorithmique du texte, programmation dynamique, suites récurrentes

L' ADN (acide désoxyribonucléique) est une molécule formée d'une double chaîne de *nucléotides*. Les nucléotides se distinguent par leur base azotée qui existe en 4 versions : l'adénine (A), la thymine (T), la guanine (G) et la cytosine (C). Cette molécule sert de support d'information pour le vivant. D'un point de vue informatique, une *séquence ADN* est donc un texte formé de caractères `A`, `T`, `G`, ou `C`.

Une caractéristique du vivant est que l'ADN subit des *mutations*. Il arrive fréquemment qu'un nucléotide soit remplacé par un autre (on parle de *substitution*), qu'un nucléotide soit supprimé (*suppression*) ou qu'un nouveau nucléotide soit inséré (*insertion*) dans une séquence. Les méthodes informatiques doivent donc s'adapter à cette particularité.

On s'intéresse ici au problème d'alignement de séquences ADN qui permet de comprendre comment une séquence d'ADN a pu être obtenue par mutations d'une autre séquence. Les algorithmes d'alignement sont très importants en biologie : ils permettent de reconstruire des arbres phylogénétiques et de mieux comprendre comment s'est déroulée l'évolution des espèces.

Pour des raisons bio-chimiques, les substitutions n'ont pas toute la même probabilité de survenir. Par exemple, il est bien plus fréquent qu'un `A` soit remplacé par un `G` que par un `T`. Pour quantifier ces différentes probabilités, on introduit généralement une *matrice de coût de substitution*. Nous utiliserons celle-ci :

	A	**G**	**C**	**T**
A	0	2	6	8
G	2	0	9	6
C	6	9	0	1
T	8	6	1	0

Ainsi d'après cette matrice, remplacer un `A` par un `G` coûte 2 tandis que remplacer un `G` par un `C` coûte 9. Remplacer une lettre par elle même n'est pas vraiment une mutation donc le coût est nul dans ce cas.

La matrice de subsitution sera codée en Python à l'aide d'un dictionnaire que l'on pourra considérer ici comme un tableau indexé par des couples de lettres. On donne le début du code à écrire pour construire la matrice :

```
matsub = {}
matsub['A', 'A'] = 0
matsub['A', 'G'] = 2
...
```

Le fichier source Python vous fournit également une fonction `S(k, l)` qui permet de générer une séquence ADN aléatoire de longueur ℓ.

1 Alignement de deux séquences ADN (Needleman-Wunsch, 1970)

Dans cette partie on s'intéresse à l'alignement de deux séquences d'ADN. Considérons par exemple les séquences `GAATTCAGTTA` et `AAGTCAGTTTA`. Une étude attentive des deux séquences permet de constater qu'elles sont similaires, il est donc possible que la seconde séquence soit en fait le résultat de plusieurs mutations sur la première séquence.

Aligner les deux séquences consiste à insérer des symboles `-` (appelés *gap*) dans chacune des séquences. Les *gaps* représentent les *suppressions* et les *insertions* et sont disposés de manière à ce que les séquences se correspondent le mieux possible. Par exemple, on peut considérer l'alignement suivant

```
00 01 02 03 04 05 06 07 08 09 10 11
G  A  A  T  T  C  A  G  -  T  T  A
A  A  G  -  T  C  A  G  T  T  T  A
```

Cet alignement traduit le fait que la seconde séquence peut s'obtenir à partir de la première avec 2 subsitutions (en positions 0 et 2), 1 suppression d'un `T` (en position 3) et 1 insertion d'un *T* (inséré en 8e position).

L'algorithme de Needleman-Wunsch permet de calculer un alignement optimal de deux séquences, c'est-à-dire qui correspond au plus faible coût en mutations. Il repose sur la *programmation dynamique*.

1.1 Coût d'un alignement

Pour donner un *coût* à un alignement, on utilisera la matrice de substitution de la partie précédente et on se fixera arbitrairement un coût de suppression $c_s = 5$ et un coût d'insertion $c_i = 4$.

Soit (s_1, s_2) un couple de séquences, un *alignement* est un couple (r_1, r_2) où r_1 (resp. r_2) a été obtenu à partir de s_1 (resp. s_2) par insertions de symboles de *gaps* et tel que pour tout i, on n'a pas $r_1[i] = r_2[i] = -$ (pas d'alignement de symboles gaps).

Le *coût de l'alignement* (r_1, r_2) est $\sum_{k=0}^{n-1} \alpha_k$ avec $n = |r_1| = |r_2|$ et

$$\alpha_k = \begin{cases} c_i \text{ si } r_1[k] =' -' \text{ (insertion)} \\ c_s \text{ si } r_2[k] =' -' \text{ (suppresion)} \\ \texttt{matsub}(r_1[k], r_2[k]) \text{ sinon (correspondance ou substitution)} \end{cases}$$

Question 1

Écrire une fonction `cout_alignement(r1 : str, r2 : str) -> int` qui prend en arguments un alignement supposé valide et qui retourne son coût. [Facultatif : verifier la validité des entrées à l'aide d'assertions.]
Donner le coût des alignements suivants :
a. `("AATTGCAT", "AA--GCAT")`
b. `("A-TTGCA-", "AAT--CAT")`
c. `(S(2000, 50), S(3000, 50))`

1.2 Coût d'alignement optimal

Un alignement de deux séquences (s_1, s_2) est *optimal* si son coût est *minimal* parmi l'ensemble des alignements possibles de ces séquences.

On remarque que lorsqu'on forme un alignement optimal, une et une seule des situations suivantes se produit :

1. la dernière lettre X de s_1 est alignée avec la dernière lettre Y de s_2,
2. la dernière lettre de s_1 est supprimée,
3. la dernière lettre de s_2 est une insertion.

Dans chacun de ces cas le reste de l'alignement, c'est-à-dire r_1 et r_2 privés de leur dernier symbole, est aussi un alignement optimal (sinon l'alignement de départ ne l'est pas).

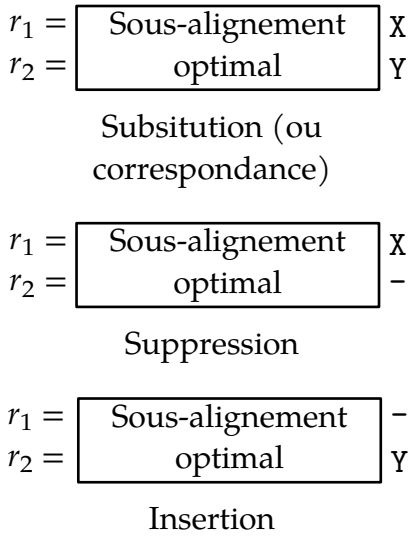

Figure 1 Alignement optimal : différents cas possibles

Ainsi un alignement optimal est formé à partir d'un alignement optimal d'un sous-problème, la programmation dynamique est donc adaptée.

Soit s_1 et s_2 deux séquences de longueurs n et m. Pour $0 \leq i \leq n$ et $0 \leq j \leq n$, on note $T[i, j]$ le coût d'un alignement optimal entre les i premières lettres de s_1 (préfixe s_1 de longueur i) et les j premières lettres de s_2 (préfixe de s_2 de longueur j).

Question 2

Parmi les schémas de programmation dynamique suivants, indiquer celui qui correspond à la méthode proposée :

$$A) \quad \forall i > 0,\ \forall j > 0, \quad T[i,j] = \min \begin{cases} T[i-1,j-1] + matsub[s_1[i-1], s_2[j-1]] \\ T[i-1,j] + c_i \\ T[i,j-1] + c_s \end{cases}$$

$$B) \quad \forall i > 0,\ \forall j > 0, \quad T[i,j] = \max \begin{cases} T[i-1,j-1] + matsub[s_1[i-1], s_2[j-1]] \\ T[i-1,j-1] + c_s \\ T[i-1,j-1] + c_i \end{cases}$$

$$C) \quad \forall i > 0,\ \forall j > 0, \quad T[i,j] = \min \begin{cases} T[i-1,j-1] + matsub[s_1[i-1], s_2[j-1]] \\ T[i-1,j] + c_s \\ T[i,j-1] + c_i \end{cases}$$

$$D) \quad \forall i > 0,\ \forall j > 0, \quad T[i,j] = \min \begin{cases} T[i-1,j-1] + matsub[s_1[i-1], s_2[j-1]] \\ T[i-1,j-1] + c_s \\ T[i-1,j-1] + c_i \end{cases}$$

On remarquera que les conditions initiales sont données par :

$$\begin{cases} T[0,j] = j \times c_i \\ T[i,0] = i \times c_s \end{cases}$$

car dans le premier cas, on n'a pas d'autre choix que d'insérer toutes les lettres de s_2 et dans le second cas il faut nécessairement supprimer toutes les lettres de s_1.

Question 3

Écrire une fonction `cout_optimal(s1 : str, s2 : str) -> int` prenant en entrée une paire de séquences et retournant le coût d'un alignement optimal.
Donner le coût d'alignement optimal des séquence suivantes.

a. `("GGTTCA", "GGTCA")`
b. `("CATTCACATCTTTAGCA", "TTTCAGATCTCTATGCA")`
c. `(S(2000, 50), S(3000, 50))`

INDICATIONS

On met en œuvre la programmation dynamique : on construit une matrice `T` de dimensions $(n+1) \times (m+1)$ et on la remplit à l'aide des conditions initiales données et de la relation de réccurrence trouvée à la question 6. La valeur que l'on cherche est $T[n, m]$.

1.3 Reconstruction d'un alignement optimal

Question 4

Écrire une fonction `alignement_optimal(s1 : str, s2 : str) -> (str, str)` prenant en arguments une paire de séquence et retournant un alignement optimal de ces deux séquences.
Donner des alignements optimaux de :
a. `("GGTTCA", "GGTCA")`
b. `("CATTCACATCTTTAGCA", "TTTCAGATCTCTATGCA")`
c. `(S(2000, 50), S(3000, 50))`

INDICATIONS

Il s'agit de reconstruire la solution obtenue par programmation dynamique à la question précédente. Pour cela, on pourra reprendre le code de la question 7 et ajouter une matrice U de mêmes dimensions que T qui permet de sauvegarder le choix effectué à chaque étape :

- $U[i, j] = 1$ si $T[i, j]$ a été obtenu par substitution (cas 1)
- $U[i, j] = 2$ si $T[i, j]$ a été obtenu par suppression (cas 2)
- $U[i, j] = 3$ si $T[i, j]$ a été obtenu par insertion (cas 3)

Une fois les matrices T et U construites par programmation dynamique. On reforme l'alignement optimal à l'aide de la matrice U, en partant de la case $U[n, m]$ et en remontant jusqu'au cas initial. L'alignement est ainsi reconstruit de la droite vers la gauche (des derniers symboles aux premiers symboles).

1.4 Calcul par mémoïsation

Dans cette partie on calcule le coût d'un alignement optimal par mémoïsation.

Question 5

Écrire une fonction `cout_optimal2(s1 : str, s2 : str) -> int` prenant en entrée une paire de séquences et retournant le coût d'un alignement optimal en utilisant une programmation dynamique de haut en bas, par mémoïsation.
Vérifier que l'on obtient les mêmes valeurs que pour `cout_optimal` sur les séquences suivantes :
a. `("GGTTCA", "GGTCA")`
b. `("CATTCACATCTTTAGCA", "TTTCAGATCTCTATGCA")`
c. `(S(2000, 50), S(3000, 50))`

INDICATIONS

On déclare un dictionnaire servant à mémoriser les valeurs $T[i, j]$ déjà calculées. On définit une fonction interne qui renvoie la valeur de $T[i, j]$, soit en se servant du résultat mémorisée s'il est disponible, soit en le calculant récursivement.

Question 6

Modifier votre code pour qu'il affiche en fin de calcul le nombre de valeurs mémorisées dans le dictionnaire. La mémoïsation a-t-elle été bénéfique pour ce problème ?

1.5 Nombre d'alignements optimaux

Lorsqu'on a reconstruit un alignement optimal, on a reconstruit une seule solution parmi tous les alignements optimaux existants. On peut se poser la question de savoir combien il existe de solutions optimales, c'est-à-dire, combien il y a-t-il d'alignements possédant un coût minimal ?

Question 7

Écrire une fonction `nb_solutions(s1 : str, s2 : str)` prenant en entrée une paire de séquences et retournant le nombre d'alignements optimaux.

INDICATIONS

Adapter la fonction `cout_optimal` pour qu'elle calcule également une matrice N dans laquelle $N[i, j]$ contient le nombre de solutions optimales pour le sous-problème (i, j).