BIN702 - Série d'exercices #1 : l'alignement de séquences

Manuel Lafond

Exercice 1: Soient les séquences

$$S = ACAATCG$$
$$T = CTCATGC$$

Considérez les scores +2 pour un "match" (deux caractères identiques alignés), et -1 pour toute autre paire de caractères alignés (gaps et mutations).

a. Remplissez la table de programmation dynamique pour calculer l'alignement **global** entre S et T. Donnez ensuite un alignement global.

b. Remplissez la table de programmation dynamique pour calculer l'alignement **local** entre S et T. Donnez ensuite un alignement local.

Solution. Vous pouvez vérifier vous-mêmes vos solutions à l'aide des outils d'alignement en ligne fournis sur la page du cours. □

Exercice 2: Une *sous-chaîne* d'une séquence S est un segment contigu de caractères de S. Par exemple si $S = ACCT$, alors les sous-chaînes possibles sont la chaîne vide, $A, C, C, T, AC, CC, CT, ACC, CCT, ACCT$.
Étant donné deux séquences S et T, le problème de la *plus longue sous-chaîne commune* demande de trouver une sous-chaîne C de S telle que C est aussi une sous-chaîne de T, et telle que C est de longueur maximum parmi les choix possibles.

a. Considérez l'algorithme naïf suivant: pour chaque sous-chaîne S' de S et chaque sous-chaîne T' de T, retenir S' et T' si $S' = T'$. Retourner la paire (S', T') retenue qui était de longueur maximum.

Quelle la complexité en temps de cet algorithme?

Solution. Soient $n = |S|$ et $m = |T|$. On suppose que $n \leq m$. Il y a $\binom{n}{2} = \frac{n(n-1)}{2}$ sous-chaînes de S — ceci est parce que pour former une sous-chaîne, on choisit deux positions: une pour le départ et une pour la fin (le nombre de façons de choisir deux positions est $\binom{n}{2}$). Il y a $\binom{m}{2}$ sous-chaînes de T, et chaque comparaison de sous-chaîne prend un temps qui peut aller jusqu'à $O(n)$. La complexité est $O(\binom{n}{2}\binom{m}{2}n) = O(n^2m^2n) = O(n^3m^2)$. □

b. Donnez un algorithme qui prend un temps $O(nm)$, où $|S| = n$ et $|T| = m$. (plus tard dans le cours, nous verrons que ceci peut se faire en temps $O(n + m)$)

Solution. On remarque que si deux sous-chaînes de longeur k sont identiques, elles correspondent à un alignement de k matches consécutifs (sans gaps ni mutations). On peut donc faire un alignement local en imposant une pénalité infinie à un gap ou à une mutation, et avec un score $+1$ pour un match. L'alignement local sera donc forcé de trouver les deux sous-chaînes les plus longues dont les caractères forment un match.

Donc, vous pouvez simplement exécuter l'algorithme Smith-Waterman pour l'alignement local avec des scores de 1 pour un match (caractères identiques) et un score de $-\infty$ pour un gap ou un mismatch. Le score maximum correspond au nombre maximum de caractères consécutifs qu'on peut matcher. Ceci prendra un temps $O(nm)$. □

Exercice 3: Soit S une séquence. Une *sous-séquence* de S est une séquence que l'on peut obtenir en supprimant des caractères de S. En d'autres termes, une sous-séquence est une séquence de caractères qui apparaissent dans le même ordre que dans S, mais pas nécessairement de façon contigüe dans S. Par exemple, si $S = ACGGTCA$, alors $AGTC$ ou encore $ACTCA$ sont des sous-séquences de S.
Étant donné deux séquences S et T, le problème de la *plus longue sous-séquence commune* demande de trouver une sous-séquence R de S telle que R est aussi une sous-séquence de T, et telle que R est de longueur maximum parmi les choix possibles.

a. Considérez l'algorithme naïf suivant: pour chaque sous-séquence S' de S

et chaque sous-séquence T' de T, retenir S' et T' si $S' = T'$. Retourner la paire (S', T') retenue qui était de longueur maximum.

Quelle la complexité en temps de cet algorithme? (ceci est une façon détournée de vous faire compter le nombre de sous-séquences d'une séquence)

Solution. Soient $n = |S|$ et $m = |T|$. On suppose que $n \leq m$. Il y a 2^n sous-séquences de S, parce que pour chaque caractère de S, on a 2 choix possibles: on l'inclut ou on ne l'inclut pas. Il y a 2^m sous-séquences de T. Chaque comparaison de sous-séquence prend un temps $O(n)$, et donc la complexité est $O(2^n \cdot 2^m \cdot n = O(2^{n+m}n)$. □

b. Donnez un algorithme qui prend un temps $O(nm)$, où $|S| = n$ et $|T| = m$. (n'essayez pas de faire mieux que $O(nm)$, car ce n'est probablement pas possible)

Solution. Exécutez l'algorithme de Needleman-Wunsch pour l'alignement global avec un score de 1 pour un match, 0 pour un gap ou un mismatch. Ceci donnera le nombre maximum de caractères qu'on peut matcher et qui se suivent dans l'ordre, ce qui correspond à une plus longue sous-séquence commune. Ceci prendra un temps $O(nm)$. □

Exercice 4: Donnez un algorithme qui retourne le **nombre** d'alignements globaux optimaux entre deux séquences S et T.

Solution. Calculez la table de programmation dynamique comme d'habitude. Pour simplifier, à chaque cellule $D[i, j]$, mémorisez les cellules adjacentes qui ont donné lieu à la valeur de $D[i, j]$ (les flèches de backtracking).
Remarquez que si vous traitez les cellules comme des sommets et les flèches comme des arêtes, vous obtenez un graphe orienté acyclique. Le nombre d'alignement globaux est égal au nombre de chemins entre $D[n, m]$ et $D[0, 0]$ dans ce graphe. Dans un graphe avec t commets et e arêtes, ceci peut se trouver en temps $O(t + e)$. Dans notre cas, $t \in O(nm)$ et $e \in O(nm)$, et donc ceci peut se faire en temps $O(nm)$.
La procédure ci-bas montre comment calculer le nombre de chemins dans la table de programmation dynamique D. L'idée est qu'à chaque cellule de $D[i, j]$, on calcule le nombre de chemins de $D[n, m]$ à $D[i, j]$, dénoté $nchemins(i, j)$. Cette valeur peut se calculer en fonction de $nchemins(i+1, j)$, $nchemins(i, j+1)$ et $nchemins(i + 1, j + 1)$.

```
fonction getNbChemins(D)
    pour i de n à 0 faire
        pour j de m à 0 faire
            nchemins(i, j) = 0;
            si i = n et j = m alors
                nchemins(n, m) = 1;
            sinon
                si il y a une flèche de D[i + 1, j] à D[i, j] alors
                    nchemins(i, j)+ = nchemins(i + 1, j);
                si il y a une flèche de D[i, j + 1] à D[i, j] alors
                    nchemins(i, j)+ = nchemins(i, j + 1);
                si il y a une flèche de D[i + 1, j + 1] à D[i, j] alors
                    nchemins(i, j)+ = nchemins(i + 1, j + 1);
            fin
        fin
    fin
    return nchemins(0, 0);
```

□

Exercice 5: L'alignement *semi-global* est comme l'alignement global, mais on ne pénalise pas une série de gaps en début et en fin de l'alignement de l'une ou l'autre des séquences. Par exemple, considérez des scores de $+2$ pour un "match" et -1 pour toute autre paire. Supposons qu'on ne pénalise pas les séries de gaps en début et fin de T seulement. Soient

$$S = GCGCGATTATAACGGGGGG$$
$$T = ATTCTATA$$

Un bon alignement semi-global serait

```
GCGCGATTATA-ACGGGGG
-----ATTCTATA------
```

Puisqu'on ne pénalise pas les séries de gap en début/fin de T, le score serait de 10 (2×6 matches, $2 \times (-1)$ mutations/gaps).
Donnez un algorithme en temps $O(nm)$ pour effectuer l'alignement semi-global.

Si vous voulez vous amuser, considérez tous les cas où on ne veut pas pénaliser les gaps en début et/ou fin de S, et/ou les gaps en début et/ou fin de T.

Solution.
Rappelons que dans la table $V[i, j]$ de l'alignement global, les rangées représentent les caractères de S et les colonnes les caractères de T. On modifie le Needleman-Wunsch classique de la façon suivante.

- Pour permettre des gaps en début de T sans pénalité, on modifie les conditions initiales $V[i, 0] = 0$ pour tout $0 \leq i \leq n$ (donc la colonne 0 est initialisée à 0). L'effet sera qu'on pourra commencer à aligner le premier caractère de T à partir de n'importe où sans pénalité. Ceci correspond à permettre un nombre arbitraire de gaps en début de T.
- Pour permettre des gaps en fin de T, on prend la valeur maximum de la dernière colonne m et on fait le backtracking à partir de cette valeur (au lieu de prendre $V[n, m]$).
- Pour permettre des gaps en début de S, on initialise la première rangée à 0.
- Pour mermettre des gaps en fin de S, on prend le maximum de la dernière colonne.

□

Exercice 6: Considérez l'alignement avec une *pénalité affine* pour des suites de gap. C'est-à-dire, on vous donne des constantes h et s telles que le coût d'un gap de taille k est $h + sk$. Ici, h est le coût d'ouverture du gap, et s le coût d'extension du gap.
Étant donné deux chaînes S et T, h, s et une matrice de mutations, donnez un algorithme en temps $O(nm)$ qui calcule l'alignement global avec une pénalité affine.
(note : cette question n'est pas facile, mais encore une fois je recommande la programmation dynamique)

Solution.
On va maintenir trois tables de programmation dynamique.

- $V[i, j]$ est le meilleur score d'alignement entre $S[1..i]$ et $T[1..j]$, comme d'habitude;

- $G_S[i,j]$ est le meilleur score d'alignement entre $S[1..i]$ et $T[1..j]$ avec la restriction que $S[i]$ est aligné avec un gap;
- $G_T[i,j]$ est le meilleur score d'alignement entre $S[1..i]$ et $T[1..j]$ avec la restriction que $T[j]$ est aligné avec un gap;

L'idée est que pour calculer $V[i,j]$, au lieu de parcourir toute la rangée i et toute la colonne j pour évaluer toutes les quantités de gaps, on va déléguer ce calcul à $G_S[i,j]$ et $G_T[i,j]$. Puisque la fonction de pénalité est linéaire, $G_S[i,j]$ pourra se fier directement à $G_S[i-1,j]$ pour se calculer et $G_T[i,j]$ pourra se fier à $G_T[i,j-1]$. Vous trouverez de plus amples justifications sur la nécessité d'avoir trois tables en ligne, par exemple sur `https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/gaps.pdf`.
L'initialisation va comme suit:

- $V[0,0] = 0$;
- $V[0,j] = -(h+sj)$ pour tout $1 \leq j \leq m$ et $V[i,0] = -(h+si)$ pour tout $1 \leq i \leq n$;
- $G_S[0,j] = -\infty$ pour tout $1 \leq j \leq m$ et $G_S[i,0] = -(h+si)$ pour tout $1 \leq i \leq n$;
- $G_T[0,j] = -(h+sj)$ pour tout $1 \leq j \leq m$ et $G_T[i,0] = -\infty$ pour tout $1 \leq i \leq n$.

et la récurrence générale est

- $G_S[i,j] = \max(G_S[i-1,j] - s, V[i-1,j] - (h+s))$: soit on continue un gap déjà entrepris, ou on en démarre un nouveau;
- $G_T[i,j] = \max(G_T[i,j-1] - s, V[i,j-1] - (h+s))$;
- $V[i,j] = \max(G_S[i,j], G_T[i,j], V[i-1,j-1] + score(S[i], T[j]))$.

□