INF421

Abdessamad Ed-dahmouni & Marouane Belhaouz

January 2017

1 Introduction :

En bio-informatique, l'alignement de séquences (sequence alignment) est une manière de représenter deux ou plusieurs séquences de macromolécules biologiques (ADN, ARN ou protéines) les unes sous les autres, de manière à en faire ressortir les régions similaires ou homologues. L'objectif de l'alignement est de disposer les composants (nucléotides ou acides aminés) pour identifier les zones de concordance.

L'alignement sequentiel a plusieurs utilisations importantes en bioinformatique car il permet un certain nombre de prédictions. Par exemple, on peut utiliser l'alignement de séquence pour prédire la ou les fonctions d'une protéine, si on détecte une homologie avec une protéine de fonction connue.

2 Tasks :

2.1 Longest common subsequence :

L'idée de l'algorithme est d'utiliser la programmation dynamique vu en cours. Nous avons deux chaînes a et b (de type string) et nous voulons calculer leur distance d'édition. Pour cela nous définissons la fonction C(i, j) qui calcule la distance entre $a[1 : i]$ et $b[1 : j]$ (les préfixes de a et b de longueurs respectives i et j).

Cette fonction est récursivement définie par :

$$C(i,j) = \begin{cases} 0 & \text{si } i = 0 \\ 0 & \text{si } j = 0 \\ C(i-1, j-1) & \text{si } a_i = b_j \\ \max \begin{cases} C(i, j-1) + 1 \\ C(i-1, j) + 1 \end{cases} & \text{sinon} \end{cases}$$

Afin de récupérer la sous suite maximale, on construit un chemin de (m, n) à (0, 0) dans la table en retrouvant à chaque étape la case qui a donné le maximum.

Complexité : Vu que les opérations à l'interieur de la boucle se font en temps constant et qu'on utilise une matrice de taille $(n+1)*(m+1)$, il est clair que la complexité de l'algorithme est $O(m*n)$ en temps et en espace. Cependant, si on voulait juste calculer la longueur de cette séquence, il est possible d'effectuer le calcul en ne gardant que la ligne précédente et la ligne actuelle en mémoire, ce qui réduit grandement la quantité de mémoire utilisée à $O(m)$.

2.2 Distance edition and alignment of sequences :

Dans cette partie, on utilise une deuxiéme fois la programmation dynamique. On commence par créer une matrice C dont la valeur de $C(i,j)$ correspond au score l'alignement entre $a[1:i]$ et $b[1:j]$. Ensuite, on remplit la matrice récursivement en commençant en haut à gauche.

$$C(i,j) = \begin{cases} j & \text{si } i = 0 \\ i & \text{si } j = 0 \\ C(i-1,j-1) & \text{si } a_i = b_j \\ \min \begin{cases} C(i,j-1)+1 \\ C(i-1,j)+1 \\ C(i-1,j-1)+1 \end{cases} & \text{sinon} \end{cases}$$

Afin de récuper l'alignement optimal, on construit un chemin de (m, n) à (0, 0) dans la table en retrouvant à chaque étape la case qui a donné le min. On distingue dans ce cas trois cas :

- Cas 1 : Le passage de la case C(i,j) vers C(i,j-1) signifie une suppression.

- Cas 2 : Le passage de la case C(i,j) vers C(i-1,j) signifie une insertion.

- Cas 3 : Le passage de la case C(i,j) vers C(i-1,j-1) signifie soit une transformation soit une égalité des deux caractére ($a_i = b_j$).

On remarque ainsi une équivalence entre le nombre minimal d'operations nécessaire pour transformer le mot a en b, et le fait d'insérer des "gaps" tel que le nombre de "gaps" plus le nombre de position où les caractères associés sont différents est minimale.

Complexité : Vu que les opérations à l'interieur de la boucle se font en temps constant et qu'on utilise une matrice de taille $(n+1)*(m+1)$, il est clair que la complexité de l'algorithme est $O(m*n)$ en temps et en espace.

2.3 Substitution matrices :

Il s'avére que le probléme précédent n'est qu'un cas particulier d'un problème plus géneral. On modifie maintenant la norme de calcul de l'alignement en introduisant les matrices de similarité. Ces matrices sont utilisées en bioinformatique pour réaliser des alignements de séquences biologiques reliées évolutivement. Elles permettent de donner un score de similarité ou de ressemblance entre deux acides aminés. Dans notre cas, on utilise la matrice "Blosum 50".

On utilise encore une fois la programmation dynamique en modifiant légerement l'algorithme précédent. Dans ce cas, le score total de l'alignement correspond à la somme des scores de chaque position.

On remplit la matrice récursivement en partant de la position (0,0).

$$C(i,j) = \begin{cases} j.Score(b_{j-1}, -) & \text{si } i = 0 \\ i.Score(a_{i-1}, -) & \text{si } j = 0 \\ \max \begin{cases} C(i, j-1) + Score(b_{j-1}, -) \\ C(i-1, j) + Score(a_{i-1}, -) \\ C(i-1, j-1) + Score(a_{i-1}, b_{j-1}) \end{cases} & \text{sinon} \end{cases}$$

Afin de récuper l'alignement optimal, on construit un chemin de (m, n) à (0, 0) dans la table en retrouvant à chaque étape la case qui a donné le max. C'est ce qu'on appelle le "Backtracking".

Complexité : Vu que les opérations à l'interieur de la boucle se font en temps constant et qu'on utilise une matrice de taille $(n+1) * (m+1)$, il est clair que la complexité de l'algorithme est $O(m * n)$ en temps et en espace.

2.4 Affine penalty :

Il s'avére qu'il est plus intéressant de supprimer ou d'ajouter un caractére dans le le même endroit plutôt que dans plusieurs endroits différents. On introduit ainsi une pénalité linéaire (opening gap, increasing gap) qui sera ajoutée au score total. On cherche dans ces nouvelles conditions un alignement de score total maximal.

Pour cela, on introduit trois matrices : C, A et B.

-C(i,j) correspond au score maximale de l'alignement entre $a[1:i]$ et $b[1:j]$ telle que a_i et b_j sont alignés à la meme position.

-A(i,j) correspond au score maximale de l'alignement entre $a[1:i]$ et $b[1:j]$ telle que a_i correspond à une fossé.

-B(i,j) correspond au score maximale de l'alignement entre $a[1:i]$ et $b[1:j]$ telle que b_j correspond à une fossé.

On remplit la matrice récursivement en partant de la position (0,0).

$$A(i,j) = \begin{cases} -\infty & \text{si } i = 0 \\ OP + i.IN & \text{si } j = 0 \\ \max \begin{cases} C(i-1, j) + OP + IN + Score(a_{i-1}, -) \\ A(i-1, j) + IN + Score(a_{i-1}, -) \\ B(i-1, j) + OP + IN + Score(a_{i-1}, -) \end{cases} \end{cases}$$

$$B(i,j) = \begin{cases} OP + i.IN & \text{si } i = 0 \\ -\infty & \text{si } j = 0 \\ \max \begin{cases} C(i,j+1) + OP + IN + Score(b_{j-1}, -) \\ A(i,j+1) + OP + IN + Score(b_{j-1}, -) \\ B(i,j+1) + IN + Score(b_{j-1}, -) \end{cases} \end{cases}$$

$$C(i,j) = \begin{cases} -\infty & \text{si } i = 0 \\ -\infty & \text{si } j = 0 \\ \max \begin{cases} C(i-1,j-1) + Score(a_{i-1}, b_{j-1}) \\ A(i-1,j-1) + Score(a_{i-1}, b_{j-1}) \\ B(i-1,j-1) + Score(a_{i-1}, b_{j-1}) \end{cases} \end{cases}$$

Pour le "BackTracking", on commence par la case qui représente le $\max(C(m,n), A(m,n), B(m,n))$ et on oscille entre les trois matrices.

Complexité : Vu que les opérations à l'intérieur de chaque boucle se font en temps constant et qu'on utilise trois matrices de taille $(n+1) * (m+1)$, il est clair que la complexité de l'algorithme est $O(m * n)$ en temps et en espace.

2.5 Local Alignment :

Les méthodes d'alignement peuvent soit essayer d'aligner les séquences sur la totalité de leur longueur, soit se restreindre à des régions limitées dans lesquelles la similarité est forte. C'est l'alignement local. On garde les memes matrices A, B et C et on modifie légerement l'algorithme précédent.

$$A(i,j) = \begin{cases} -\infty & \text{si } i = 0 \\ OP + i.IN & \text{si } j = 0 \\ \max \begin{cases} C(i-1,j) + OP + IN + Score(a_{i-1}, -) \\ A(i-1,j) + IN + Score(a_{i-1}, -) \\ B(i-1,j) + OP + IN + Score(a_{i-1}, -) \end{cases} \end{cases}$$

$$B(i,j) = \begin{cases} OP + i.IN & \text{si } i = 0 \\ -\infty & \text{si } j = 0 \\ \max \begin{cases} C(i,j+1) + OP + IN + Score(b_{j-1}, -) \\ A(i,j+1) + OP + IN + Score(b_{j-1}, -) \\ B(i,j+1) + IN + Score(b_{j-1}, -) \end{cases} \end{cases}$$

$$C(i,j) = \begin{cases} -\infty & \text{si } i = 0 \\ -\infty & \text{si } j = 0 \\ \max \begin{cases} C(i-1,j-1) + Score(a_{i-1}, b_{j-1}) \\ A(i-1,j-1) + Score(a_{i-1}, b_{j-1}) \\ B(i-1,j-1) + Score(a_{i-1}, b_{j-1}) \\ 0 \end{cases} \end{cases}$$

Pour le "BackTracking", on commence par la case qui représente l'élement maximal de la matrice C, on oscille entre les matrices A, B et C et on s'arrete à la case (i,j) telle que $C(i,j) = 0$. L'idée de l'algorithme est que l'alignement local de score maximal commencera et finira toujours par une transformation.

Complexité : Vu que les opérations à l'interieur de la boucle se font en temps constant et qu'on utilise trois matrices de taille $(n+1) * (m+1)$, il est clair que la complexité de l'algorithme est $O(m * n)$ en temps et en espace.

2.6 BLAST :

Pour des questions de complexité en temps, il est mieux d'utiliser un algorithme d'approximation au lieu de notre algorithme exact, l'algorithme d'approximation proposé ici est BLAST(Basic Local Alignment Search Tool).

Pour implémenter l'algorithme de BLAST, on écrit deux fonctions auxiliaires : *generate*() et *indices*().
La fonction *generate* a pour paramètres :
input : une chaîne de charactères qui contient toutes les lettres acceptables.
g : reprèsente le nouveau gène.
th : paramètre qui permet de calculer le seuil du score.
N : la longueur des mots à générer.

Cette fonction génère tous les mots de longueur N qui ont un score supérieur au seuil correspondant, et ceci en appelant la fonction récursive qui a le même nom, qui ajoute à chaque pas de la récursion une lettre à la fin du mot *str* en cours de construction. L'ensemble des mots renvoyés par cette fonction, noté $\mathcal{S}_g$ (*Seeds*) est utilisé ensuite par la fonction *indices*.

La structure d'arbre de mot-clés (*keywordtree*) utilisée pour stocker $\mathcal{S}_g$ a pour avantage la réduction de la compexité en espace, car pour tous les mots qui ont le même préfixe, ce dernier est stocké une seule fois dans la mémoire.

La fonction *indices* renvoie ensuite les indices du début de coïncidence entre un sous-mot de t et un mot de $\mathcal{S}_g$ en parcourant les sous-mots de t et en les ajoutant à la liste à revoyer s'ils appartiennent à l'arbre représentant $\mathcal{S}_g$.

Finalement, la fonction $BLAST$ utilise la liste des indices générées par la fonction *indices* et cherche les sous-mots correspondants de g et t, l'alignement de ces deux sous-mots est ensuite progréssivement prolongé à gauche et à droite tant que l'extrémité d'une des séquences n'est pas atteinte et le score ne décroît pas. cette fonction renvoie à la fin la liste des alignements qui ont un score supérieur au seuil $th_l.s_g$. La classe Align est une simple structure de données qui contient les triplets (premier indice, longueur, score), utilisée pour représenter les alignements trouvés.