Gabriel Misrachi
Raphael Montaud

Programming project : From DNA to formation of proteins : how to align sequences ?

Task 1 :

Une première approche, biologiquement pertinente, consiste à chercher la plus longue sous-séquence commune entre nos deux séquences de données. Cette résolution est biologiquement pertinente puisqu'elle permet de passer outre les phénomènes qui peuvent altérer une séquence, d'ADN par exemple, au cours du temps. Les mutations qu'elles soient par substitution, délétion ou insertion ne diminue que d'une unité la taille de la plus longue séquence commune. Ainsi deux séquences de source identique mais mutées au cours du temps ont de bonne chance d'avoir une sous-séquence commune relativement longue.

Voici un premier algorithme naïf pour résoudre ce problème de plus longue-sous séquence commune.

Nous avons en entrée un string s1 de taille n et un string s2 de taille m. On supposera que $n \leq m$ pour simplifier.
Nous allons d'abord décrire un algorithme auxiliaire qu'on appellera test.

Il prend en entrée une paire de string sa et sb où sa est de taille inférieure ou égale à sb. L'algorithme cherche alors si le string sa existe dans le bon ordre dans sb. Il renvoie True si c'est le cas et False sinon.
Exemple : si on lui donne en argument AAL et RAPHAEL, il va trouver l'alignement (-A—A-L) et déduire que AAL existe dans RAPHAEL. Il va renvoyer True.

Cette méthode requiert un temps de calcul linéaire en la taille de sb.
Il suffit alors d'appliquer la méthode test avec les arguments (s,s2) où s parcourt toutes les sous-séquences de s1.
Exemple : si s1 = RAPH, on peut faire le dénombrement de la façon suivante :
-on enlève une lettre dans RAPH. Cela donne les sous-séquences RAP, RAH, RPH, APH. Cela correspond à 1 parmi 4 sous-séquences.
-on enlève deux lettres dans RAPH. Cela donne RA, RH, PH, AH, RP, AP. Cela correspond à 2 parmi 4 sous-séquences.
-etc

On obtient donc $\sum_{k=1}^{n}\binom{n}{k} = 2^n$ sous-séquences de s1 à chercher dans s2. Cela nous donne un temps de calcul total en $O(m * 2^n)$. Ce temps exponentiel n'est clairement pas satisfaisant. C'est pourquoi nous allons proposer un algorithme plus efficace dans Task 2.

Task 2 :

Toutefois, il est possible d'améliorer considérablement les performances de notre algorithme.

Pour optimiser le temps de calcul nous allons nous baser sur les méthodes de dynamic programming.
Tout repose sur la réduction du problème suivante :
On cherche la plus grande sous-séquence commune aux mots $a_1a_2\ldots a_n$ et $b_1b_2\ldots b_m$:
-Si $b_m = a_n$ alors on sait que ces deux caractères doivent être alignés dans la plus grande sous-séquence commune. Donc on peut les retirer tous les deux puis chercher la plus grande sous-séquence commune dans $a_1a_2\ldots a_{n-1}$ et $b_1b_2\ldots b_{m-1}$.
-Sinon il faudra forcément retirer une des deux lettres et elle ne sera alignée avec personne. Donc la plus grande sous-séquence correspond aux deux choix possibles : c'est le max des plus grandes sous-séquences communes de ($a_1a_2\ldots a_n$, $b_1b_2\ldots b_{m-1}$) et ($a_1a_2\ldots a_{n-1}$, $b_1b_2\ldots b_m$).

On en déduit la récurrence suivante :

Si : $a_i = b_j$ alors :
c[i,j] = 1 + c[i-1,j-1]

Sinon :
c[i,j] = max (c[i-1,j], c[i,j-1])

où c[i,j] représente la taille de la plus longue sous-séquence commune des i premières lettres de sa et des j premières lettres de sb.
On peut donc construire un tableau contenant ces valeurs c[i,j] de gauche à droite et de bas en haut. Le calcul se fait alors en $O(n * m)$. Et on peut lire le résultat sur la case c[n,m].
Nous avons donc grandement amélioré l'efficacité du calcul grâce à cette méthode de mémoization.

Task 3 :

On s'intéresse maintenant à ce que l'on appelle la distance d'édition entre deux séquences.
Etant donné deux séquences, la distance d'édition entre celles-ci est le minimum du coût pour passer de l'une à l'autre en appliquant des mutations par substitution, délétion ou insertion. Chaque type de mutation ayant un coût défini au préalable (un coefficient lié à sa fréquence, par exemple).
Bien que les deux problèmes ne portent pas le même nom on peut cependant remarquer qu'il existe une équivalence entre distance minimale d'édition et plus grande sous-séquence commune. Soient s et t deux séquences. On note n la taille de la plus grande séquence. On note p la distance minimale d'édition et u la taille de la plus grande sous séquence commune.
Reprenons l'exemple du sujet. On parlera de s pour la séquence du dessus et de t pour celle du dessous.

C	A	C	T	A	A	G	-	C	A	T	C	A	G	C	-
-	A	-	T	A	G	G	G	C	A	-	-	A	T	C	T

Une édition possible de t vers s est de disposer les séquences comme ci-dessus puis de procéder à une délétion lorsqu'un caractère de t est face à un tiret. Une insertion lorsqu'on a un tiret dans t et un remplacement lorsqu'on a deux caractères différents face à face. Donc on voit qu'une édition possible se fait en n-u opérations donc nécessairement $p \leq n - u$. Par ailleurs si une édition de t vers s est possible en p opérations alors il existe nécessairement une sous-séquence commune de taille n-p soit $n - p \leq u$. On en déduit donc que $u = n - p$. D'où le lien entre distance minimale d'édition et plus grande sous-séquence commune.

Comment alors retrouver l'alignement optimal et pas seulement sa taille ?
Au cours du calcul nous allons mémoriser les décisions qui sont prises. Cela signifie que à chaque fois que l'on calcule une case du tableau c[i,j], on mémorise à partir de quelle direction il a fait son calcul. Si les deux lettres étaient identiques, on mémorise une flèche diagonale. Sinon, si le maximum se situait à sa gauche, on retient une flèche vers la gauche et si le maximum était la case du dessus alors on mémorise une flèche vers le haut.
On obtient alors un tableau de flèches que l'on va pouvoir remonter très simplement. On démarre à la case (m,n) tout en bas à droite. Dans chaque case la flèche indique la direction de la décision optimale. Une flèche vers la gauche indique qu'il faut aligner la dernière lettre de s avec un tiret, une flèche vers le haut indique qu'il faut aligner la dernière flèche de t avec un tiret. Enfin une flèche diagonale signifie qu'il faut aligner les deux lettres considérées. On parcourt alors le tableau entier en suivant les flèches.

On peut alors très simplement retrouver l'alignement idéal.

Task 4 :

Les résolutions effectuées jusqu'à présent visaient surtout les séquences d'ADN puisqu'implicitement on considérait les fréquences de chaque mutation indépendante du caractère muté, i.e de la base azotée. Or, lorsque l'on travaille avec des acides aminés, leurs possibles mutations et leurs proportions varient beaucoup. On peut donc leur affecter des scores différents et ainsi donner plus de poids aux données biologiques. Ces scores sont contenus dans la matrice Blosum 50.
Malgré sa pertinence l'ajout de cet argument biologique ne change presque pas la résolution algorithmique du problème.
Il suffit de reprendre la même récurrence que précédemment en la modifiant légèrement:

c[i,j] est le maximum de :

- c[i-1,j] + score(a_i , -)
- c[i, j-1] + score(- , b_j)
- c[i-1 ,j-1] + score(a_i , b_j)

Il suffit ensuite de reprendre la même structure que précédemment.

Task 5 :

On va à présent modéliser le fait que les délétions ou insertions sont la plupart du temps localisées.
Pour ce faire on introduit des pénalités (gap) de deux types. La première pénalité sanctionne l'ouverture d'une nouvelle séquence de délétions ou d'insertions et la deuxième sanctionne les insertions ou délétions qui interviennent à la suite d'une autre mutation du même type.

Encore une fois, algorithmiquement la différence est mineure.

On introduit les pénalités (gap). Le calcul des scores va alors se faire en parallèle de la mémorisation des flèches. En effet introduire une nouvelle séquence de tirets (gap 1) revient à choisir une direction haut (respectivement bas) alors que la flèche de la case de gauche (resp. haut) pointait dans une autre direction. Une pénalité gap 2 correspond à la poursuite dans une direction haut ou bas. Ce nouveau modèle plus complexe permet de se rapprocher du cas réel ou les tirets isolés sont rares.
Nos résultats montrent effectivement que plus les pénalités sont grandes plus l'algorithme va chercher à éviter les séquences de tirets isolés. Cela permet de se rapprocher des alignements trouvés dans la réalité où les tirets isolés sont rares. Par exemple, sur des séquences de taille environ 300, on obtient 74 pénalités gap1 et 27 pénalités gap 2 pour des valeurs (gap1 = 3, gap2 = 1) mais ces nombres se réduisent à 46 et 29 pour (gap1 = 10, gap2 = 3).

Task 6 :

On cherche maintenant à résoudre notre problème de sous-séquence commune localement. C'est à dire que l'on ne souhaite plus aligner les séquences entières mais seulement les parties pertinentes de celles-ci, les protéines par exemple.

Pour calculer le meilleur alignement local faire varier les indices i, j , ip , jp (les indices des sous-séquences de s et t sur lesquelles on fait tourner task 5) de façon à tester tous les scores optimaux des sous-mots de s et t et de trouver les indices qui produisent le meilleur score.

Task 7 :

Après toutes ces versions exactes de recherche du meilleur alignement antre deux séquences, on se propose d'implémenter une solution non exacte mais moins coûteuse en temps de calcul. Dans la task 7, étant donné une séquence g d'acides aminés et un entier k, nous allons travailler sur Wg l'ensemble

des sous-mots de taille k de g. Le but est d'établir des matchs parfaits entre les séquences mutées "acceptables" des mots de Wg et les sous-mots de l'autre séquence. L'acceptabilité d'une mutation d'un mot est définie grâce à un critère seuil sur son score obtenu avec Blosum 50. Ce seuil est proportionnel au score du mot originel de coefficient de proportionnalité th. Voici la résolution algorithmique de ce problème.

Soit w un mot de Wg. Il existe 20^k mots de k lettres qu'on peut construire avec l'alphabet de Blosum50 (20 correspondant à la taille de l'alphabet utilisé).
Un premier algorithme naïf (Task7bis dans le code) consiste à tester le score de tous ces mots alignés avec w et de retenir ceux dont le score respecte le critère th*score(w,w). Cette méthode n'est vraiment pas efficace et son temps de calcul augmente exponentiellement avec k.
Une autre approche est de calculer les scores à l'aide d'un arbre.
On créé une classe Arbre_Calcul dédiée au calcul de ces scores. Prenons un exemple. On cherche les mots de 4 lettres qui ont un bon score avec RAPH. On créé un arbre dont les enfants vont correspondre à des caractères de l'alphabet. Dans la première génération, on calcule le score de R avec chacune des lettres de l'alphabet. Puis à partir de ces scores on va pouvoir calculer la suite. Par exemple le score score(RAP,RAX) est égal au score score(RA,RA) + score(P,X) donc on a pas besoin de recalculer score(RA,XA). Cela permet de gagner du temps. Mais l'optimisation essentielle est que l'on coupe les branches « mortes ». Pour cela on va évaluer le score du meilleur descendant possible de la branche. Par exemple si l'on a déjà calculé score(RA,XA) alors le meilleur cas possible pour un descendant de XA est d'avoir le reste des lettres en commun avec RAPH. C'est-à-dire que le meilleur descendant de XA est XAPH. Cependant, si l'on voit que score(XAPH,RAPH) ne respecte pas le critère th*score(RAPH,RAPH) alors on coupe cette branche et on ne prend pas la peine de calculer ses descendants.
Le fait d'effectuer les calculs sous forme d'arbres permet de diviser par k le nombre d'additions à faire. Il est difficile d'évaluer le temps économisé en coupant des branches. En effet dans le pire des cas (th=0), on effectuera tous les calculs donc on n'a rien gagné par rapport à la méthode naïve. En revanche, dans le meilleur des cas (th=1), l'algorithme ne parcourt que le chemin qui l'amène directement à la seule solution (RAPH) car il n'y a « pas le droit à l'erreur » : une simple lettre qui diffère et le meilleur descendant ne respecte plus le critère. On sent donc que plus th est proche de 1 et plus notre algorithme se montrera efficace. Nous l'avons donc testé pour plusieurs valeurs de k et th pour évaluer son efficacité.
Nous présentons ici les ratios de temps de calcul entre les deux méthodes pour différentes valeurs de k et th. (Ces essais ont été effectués avec des séquences de taille environ 300).

	th = 0,5	th = 0,7	th = 0,9
k = 3	15%	13%	2%
k = 4	3,60%	0,80%	0,30%
k = 5	1%	0,10%	0,02%

On observe donc comme prévu que les ratios sont décroissants en k et en th.

Remarque : Beaucoup de graines ne se trouvent finalement pas dans t, et on a donc calculé un certain nombre de scores pour rien. D'après nos calculs il serait en fait beaucoup plus rapide de parcourir t à la recherche de sous mots acceptables tant que l'on reste sur des tailles raisonnables de chaines de caractères.

Task 8 :

Dans la task 8 on essaye d'étendre les matchs de taille k, de nouveau en utilisant un argument de score seuil. C'est à dire qu'on étend si le score de l'extension augmente. Finalement, on garde les extensions qui ont un score supérieur à un seuil proportionnel au score du sous mot de g initial avec lui-même.