PROGRAMMING PROJECT (JACOB LEYGONIE – CLAIRE LASSERRE)

FROM ADN TO FORMATION OF PROTEINS : HOW TO ALIGN SEQUENCES ?

Task 1

La première partie de ce projet consiste à trouver la taille de la plus longue sous-séquence commune entre 2 séquences de lettres : a=a1…an et b=b1…bm qui peuvent représenter l'ADN par exemple.

⇒ $u=u_1…u_k$ est une sous-séquence commune de a et b s'il existe des indices $(i_1…i_k)$ et $(j_1,..,j_k)$ avec 1<=i1<i2<..<ik<=n et 1<=j1<j2<…<jk<=m tels que pour l dans {1…k} $u_l=a_{il}=b_{jl}$.

Le problème de la plus grande sous-séquence commune prend en entrée les 2 séquences a et b et renvoie la longueur de la plus grande sous-séquence (qui n'est pas unique).

```
LongestSubSeq(a, b)
Input : a,b
Outpout : entier l = longueur de la plus grande sous-séquence commune
```

UNE FACON TRES NAÏVE DE RESOUDRE CET ALGORITHME CONSISTE A CREER L'ENSEMBLE DES 2^N SOUS-SEQUENCES DE A

(sous forme de liste chainée par exemple) et ensuite de déterminer si elles sont sous-séquences de b. Pour cela, il suffit de parcourir b et vérifier que tous les éléments de la sous séquence de a considérée y apparaissent (coût en O(m)). Puis il ne garde que la sous-séquence commune de longueur maximale.

- Le code se termine car une fois parcourues les 2^n sous séquences de a l'algorithme s'arrête (et le coût de comparaison est fini).
- L'algorithme compare tous les cas possibles de sous-séquences : il analyse toutes les sous-séquences de a et détermine si elles sont sous-séquences de b. Il trouve donc bien les sous-séquences communes. Puis il détermine la longueur maximale, donc il renvoie bien le résultat attendu.
- Complexité : Le cout de comparaison consiste à parcourir les deux sous-séquences de tailles au pire n et m respectivement. Cette méthode coute donc très cher avec une complexité dans le pire des cas en O (2^n*m).

UNE FAÇON UN PEU MOINS NAÏVE UTILISE UNE PROPRIETE DE RECURRENCE SANS MEMOÏSATION

Pour trouver une propriété d'invariance, examinons ce qu'il advient à tout instant de a[0] :

- a[0] appartient à la plus grande sous-séquence commune
  - ⇒ Il s'apparie avec b[0] (ie a[0]=b[0])-> LongestSubSeq(a, b) = LongestSubSeq(a / a[0], b/b[0])+1
  - ⇒ Il s'apparie avec un élément de b / b[0] -> LongestSubSeq(a, b) = LongestSubSeq(a, b/b[0])
- a[0] n'appartient à la plus grande sous-séquence commune
  - ⇒ LongestSubSeq(a, b) = LongestSubSeq(a/a[0], t)

Le but de l'algorithme est trouver la plus grande sous séquence commune : on prend donc le max des 3 valeurs obtenues. Si a[0]=b[0], la première solution est forcément maximale car elle n'élimine pas un appariement possible.

Une façon naïve de répondre à ce problème n'utilise pas de mémorisation. Il ne retient pas les calculs intermédiaires et va donc effectuer le même calcul de nombreuses fois. En java, il s'écrit :

```
public static int naive(String a, String b){
        int n = a.length();
        int m=b.length();
        if (n==0 || m==0){
                return 0;
        }
        else {
                if (a.charAt(0)==b.charAt(0)) {
                        return 1+naive(a.substring(1),b.substring(1));
                }
                else {
                        return Math.max(naive(a.substring(1),b), naive(a,b.substring(1))) ;
                }
        }
}
```

- L'algorithme se termine car à chaque étape on réduit soit a, soit b, soit les 2. a ou b finit donc bien par être de taille nulle et on se trouve dans la condition du premier if (initialisation) qui permet de terminer l'algorithme.
- Au vu de la propriété d'invariance précédente, l'algorithme renvoie bien la longueur de la plus grande sous-séquence.
- Évaluons la complexité de l'algorithme. Notons T(n,m) le temps de calcul de la plus longue sous-séquence pour un mot de taille n et un mot de taille m.
  D'après la relation de récurrence : T(n,m)= 1+T(n-1,m-1) ou T(n,m)=1+ T(n-1,m)+ T(n,m-1). Le second cas étant le plus coûteux. Plaçons-nous dans l'hypothèse la plus coûteuse où seulement des appels récursifs du second type sont exécutés. Il s'agit donc d'évaluer l'ordre de grandeur de la suite U(n,m) définie par :
  U(n,m)=1+U(n,m-1)+U(n-1,m) ;
  U(0,k) =0 ;
  U(k',0)=0
  Il s'agit d'un calcul bien connu. On peut par exemple, par récurrence sur la somme n+m, vérifier que le résultat qui suit est vrai.
  U(n,m)= (n parmi n+m) -1

Task 2

Une façon plus intelligente de résoudre le problème est d'utiliser la programmation dynamique.

- La relation de récurrence nous vient de l'invariant de la task 1 (que l'on ré-écrie en considérant le cas du dernier élément de a et non plus du premier). Pour clarifier le problème on prend en argument les tailles des 2 séquences on a donc :
  ⇒ Cas extrême : a ou b est vide : LongestSubSeq(a, b, n, m) =0
  ⇒ Si a[n-1]=b[m-1] : LongestSubSeq(a, b, n, m) = LongestSubSeq(a/a[n-1]; b\b[m-1]; n-1;m-1) +1
  ⇒ Sinon LongestSubSeq(a; b; n;m) = max(LongestSubSeq(a/a[n-1]; t; n-1;m), LongestSubSeq(a, b/b[m-1];; n;m-1))
  On appelle cette relation (rec2)
- Et la mémorisation se fait à travers une matrice que l'on remplie au fur et à mesure qui permet d'éviter de refaire les mêmes calculs plusieurs fois (de plus le cout d'accès est en temps constant). t(i,j) correspond à la longueur de la plus grande sous-séquence entre a(0…i-1) et b(0…j-1)

NOUS AVONS DEFINI UNE FONCTION AUXILIAIRE DYNAMICAUX

Input : les deux séquences a,b et les indices i,j et la matrice t en cours de construction
Output : Grâce à la relation de récurrence cette fonction remplit la case t(i,j) à partir des autres éléments de la sous matrice de taille i,j.

NOTRE FONCTION PRINCIPALE DYNAMIC

Input : les deux séquences a,b
Output ; la plus grande sous-séquence commune

dynamic(String a, String b) remplit la matrice t grâce à la fonction dynamicAux. t est initialisée en commençant par la 1ere ligne et la 1ere colonne qui sont uniquement constituées de 0 (cas extrême de la relation de récurrence). Puis elle remonte par colonne croissante puis ligne croissante, ce qui est permis par la relation de récurrence de la fonction auxiliaire. Le dernier terme t[n][m] correspond bien au résultat du problème demandé.
Voici un exemple de l'algorithme en cours de route sur deux mots de taille 7 et 8.

	9	10	11	12				
	1	2	3	4	5	6	7	8

En vert l'initialisation

- L'algorithme termine car il s'arrête après avoir parcouru toute la matrice de taille (n+1)*(m+1) (finie).
- L'algorithme renvoie le résultat demandé au vu de la relation de rec2.
- On voit clairement sur l'algorithme Java que la complexité temporelle est en O(nm)) car deux boucles for de longueur n+1 et m+1 respectivement sont imbriquées et, à l'intérieur de ces deux boucles, est appliquée la fonction dynamiqueAux qui a un coût constant. La complexité spatiale est également un O(nm) car on stocke nos données dans une matrice de taille (n+1)*(m+1).

Task3

THEORIE : EQUIVALENCE DES 2 PROBLEMES

équivalence entre

(i) Trouver une façon optimale (le moins d'opérations possibles) pour passer d'un mot s à un mot t.

(ii) Insérer des hyphens entre les lettres de s et celles de t de sorte à minimiser les différences de vis-à-vis entre s et t.

En visualisant un hyphen (-) sur le mot s comme l'insertion dans s de la lettre en face de – dans g, un hyphen sur le mot t comme la délétion dans s de la lettre en face de -, et deux lettres non identiques de s et t en vis-à-vis comme une substitution de la lettre de s par celle de t, la bijection entre les différentes façons de passer de s à t et la façon de placer des hyphens pour avoir des mots de même longueurs est claire, ainsi que l'équivalence.

L'algorithme demandé :

Input : String a,b

Output : tableau avec 2 lignes : première ligne séquence de a modifiée (éléments de a avec hyphen), deuxième ligne séquence de b modifiée (éléments de b avec hyphen) qui minimise le nombre de lettres ou a et b diffèrent est minimal.

NOTRE ALGORITHME AUXILIAIRE ALIGNE MENT_AUX

alignementAux(String a, String b, int i, int j, int[][]t, String[][] res1, String [][] res2) permet de remplir les cases (i,j) de t, res1 et res2.
- t(i,j) est la longueur de la plus grande commune sous-séquence entre a(0…i-1) et b(0…j-1) -> on la remplit de la même façon que précédemment dans dynamic Aux.
- res1(i,j) et res2(i,j) correspondent respectivement à la liste de caractères (hyphen et lettre) de a.substring(i) et b.substring(j) transformés de telle sorte que a et b diffèrent en un minimum de lettre.

Cas 1 : initialisation : remplir la ligne 0 ou colonne 0

Pour la première ligne (i=0), cela correspond à a=liste vide et b quelconque-> on est obligé de transformer a en insérant tous les éléments de b, ce qui correspond à ne mettre que des hyphens sur a et ne rien changer à b :
res1[0][j] = « _____ » (j hyphens)
res2[0][j] = b.substring(0,j);

Pour la première colonne (j=0) réciproquement a est inchangé mais on ne met que des hyphens sur b.
res1[i][0] = a.substring(i)
res2[i][0] = b« _____ » (i hyphens)

Cas 2 : pour les autres cas (i !=0, j !=0)
- Si a (i-1)=b (j-1) //cas2.A.
On apparie a(i-1) et b(j-1) et on effectue l'alignement de a(0…i-2) et b(0…j-2)

- Sinon // cas 2.B, on sait que dans la meilleure sous-séquence ils ne seront pas appariés donc on considère les 3 cas possibles :

⇒ Soit (cas 2.B.1) le fait de considérer a(i-1) permet d'obtenir une meilleure sous-séquence commune entre a(0...i-1) et b(0…j-2) qu'auparavant entre a(0...i-2) et b(0…j-2) donc on essaie d'apparier a(i-1) dans b/b(j-1) et en face de b(j-1) on doit donc placer un hyphen car on doit insérer dans a b(j-1) pour que les séquences finales soient identiques . Finalement cela revient à effectuer l'alignement pour a(0...i-1) et b(0…j-2) et à rajouter un hyphen pour a transformé et b(j-1) pour b transformé.
⇒ Soit (cas 2.B.2) le fait de considérer b(j-1) permet d'obtenir une meilleure sous-séquence commune entre a(0...i-12) et b(0…j-1) qu'auparavant entre a(0...i-2) et b(0…j-2) donc on essaie d'apparier b(j-1) dans a/a(i-1) et en face de a(i-1) on doit donc placer un hyphen car on doit insérer dans b a(i-1) pour que les séquences finales soient identiques . Finalement cela revient à effectuer l'alignement pour b(0...j-1) et a(0…i-2) et à rajouter un hyphen pour b transformé et a(i-1) pour a transformé.
⇒ Soit (cas 2.B.3) ni a(i-1) ni b(j-1) ne permettent d'avoir une meilleure sous séquence et on les met donc face à face dans l'alignement et cela correspondra biologiquement à une substitution.

- Comme on considère tous les cas possibles dans cette boucle, l'algorithme est réalisable (avec comme hypothèse que tous les éléments des 3 matrices des cases inférieures à (i,j) sont déjà déterminés). Il se termine car on diminue progressivement les valeurs de i et j jusqu'à tomber sur les cas limites i=0 et j=0 qui ont été initialisées.
- Il renvoie le résultat demandé au vu de la relation de récurrence que l'on vient d'expliciter.
- La complexité est constante car comme les matrices du rectangle inférieur sont remplies on a accès aux valeurs nécessaires pour la récurrence en temps constant et ensuite on a juste faire quelques comparaisons, concaténations et assignations.

DESCRIPTION DE L'ALGORITHME PRINCIPAL ALIGNEMENT

Input : String a,b
Output : a modifié et b modifié sous forme d'un tableau de String (respectivement 1ère et 2ème lignes) qui minimise les points de différences.

Comme pour l'algorithme de la task2, on applique la fonction auxiliaire par ligne et colonne croissante. Comme la 1ère ligne et la 1ère colonne sont connues (comme décrit ci-dessus), on va pouvoir remplir les trois matrices t, res1 et res2 par rectangles de tailles croissantes grce à la fonction alignementAux. L'algorithme renvoie res1[n,m] et res2[n,m] qui correspondent à l'alignement des matrices a et b toutes entières, c'est à dire les séquences a et b modifiés.

- L'algorithme termine car on remplit les matrices res1 et res2 de taille n*m et à chaque itération on effectue les algorithmes auxiliaires qui eux-mêmes terminent.
- L'algorithme renvoie le résultat demandé. Ceci vient de la justesse de l'algorithme auxiliaire.
- On voit clairement sur l'algorithme Java que la complexité temporelle est en O(nm) car deux boucles for de longueur n+1 et m+1 respectivement sont imbriquées et, à l'intérieur de ces deux boucles, est appliquée la fonction alignementAux qui a un coût constant. La complexité spatiale est également un O(nm) car on stocke nos données dans 3 matrices de taille (n+1)*(m+1).

Task4

L'algorithme demandé :

Input : String a,b, matrice Blosum50 qui prend en compte les données biologiques en donnant le score entre 2 caractères possibles (cette matrice est une donnée du problème et ne passe pas en paramètre de la fonction, bien que cela ne change rien à la difficulté du problème.)

Output : tableau avec 2 lignes : en première ligne la séquence de a modifiée (éléments de a avec des hyphens entre les lettres), en deuxième ligne la séquence de b modifiée (éléments de b avec hyphens entre les lettres) qui maximise l'alignement entre a et b sous le joug de la matrice Blosum50.

ALGORITHME AUXILIAIRE SCOREAUX

Input : String a, String b, int i, int j, int[][]t, String[][] res1, String [][] res2
Output = remplir la case (i,j) de t, res1, res2.

Cas 1 : initialisation : remplir la ligne 0 ou colonne 0
Pour la première ligne (i=0), cela correspond à a=liste vide et b quelconque-> on est obligé de transformer a en insérant tous les éléments de b, ce qui correspond à ne mettre que des hyphens sur a et ne rien changer à b :
res1[0][j] = « _____ » (j hyphens)
res2[0][j] = b.substring(0,j);
t[0][j] = somme des scores des appariements (b(k), -) pour k dans (0…j-1)

Pour la première colonne (j=0) réciproquement, a est inchangé mais on ne met que des hyphens sur b.
res1[i][0] = a.substring(i)
res2[i][0] = b« _____ » (i hyphens)
t[i][0] = somme des scores des appariements (a(k), -) pour k dans (0…i-1)

3 cas possibles, dans l'alignement final

- a[i-1] et b[j-1] sont appariés
- a[i-1] est apparié avec un -
- b[j-1] est apparié avec un -

Pour chacun des cas, on calcul le score (depuis Blosum50) de ces appariements singuliers et on en fait la somme avec le cas récursif sous-jacent.
Puis on se place le meilleur de ces trois cas et on remplit t, res1 et res2 en fonction (voire commentaires).

- Comme on considère tous les cas possibles dans cette boucle, l'algorithme est réalisable (avec comme hypothèse que tous les éléments des 3 matrices du rectangle inférieur de (i,j) sont déterminés). Il se termine car on diminue progressivement les valeurs de i et j jusqu'à avoir i ou j=0, correspondant à l'initialisation.
- Il renvoie le résultat demandé au vu de la relation de récurrence que l'on vient d'expliciter car on considère tous les cas possibles et on renvoie celui qui maximise le score à l'étape (i,j). Par récurrence on aura bien le score maximal à la fin de l'exécution.
- La complexité est constante car comme les matrices du carré inférieur sont remplies on a accès aux valeurs nécessaires pour la récurrence en temps constant et ensuite on a juste faire quelque comparaisons, concaténations et assignations.

ALGORITHME PRINCIPAL

Score(String a, String b)

Input : String a, String b, la matrice Blosum50 est également une donnée globale (on ne l'a met pas en argument).

Output : a modifié et b modifié sous forme d'un tableau de String (respectivement 1ère et 2ème lignes) qui maximise le score de l'alignement.

On applique la fonction auxiliaire par ligne et colonne croissante. Comme la 1ère ligne et la 1ère colonne sont connues (comme décrit ci dessus), on va pouvoir remplir les trois matrices t, res 1 et res 2 par rectangles de tailles croissantes grâce à la fonction ScoreAux.
L'algorithme renvoie res1[n,m] et res2[n,m] qui correspondent à l'alignement des matrices a et b toutes entières.

- L'algorithme termine car on parcourt les matrices de taille (n+1)*(m+1) et à chaque

itération on effectue les algorithmes auxiliaires qui eux-mêmes terminent.
- L'algorithme renvoie le résultat demandé. Ceci vient de la justesse de l'algorithme auxiliaire.
- On voit clairement sur l'algorithme Java que la complexité temporelle est en O(nm) car deux boucles for de longueur n+1 et m+1 respectivement sont imbriquées et, à l'intérieur de ces deux boucles, est appliquée la fonction ScoreAux qui a un coût constant. La complexité spatiale est également un O(nm) car on stocke nos données dans 3 matrices de taille (n+1)*(m+1)

Task5

L'algorithme est le même que Score(String a, String b) mais il prend en compte une pénalité d'écart d'ouverture pour chaque nouvel événement d'insertion et de suppression et une pénalité d'écart croissante pour chaque insertion ou suppression suivante qui se produit au même endroit.

fonction auxilliaire ScorePenaltyAux
Input = séquences a,b, indices i,j et les matrices t, res2, res2, pen, lettres et les coûts de pénalités open et incr
Output : remplit
- t, res1 et res2 selon les mêmes principes que pour ScoreAux
- pen[i][j][0]=true; a n'a pas encore commencé
- pen[i][j][1]=true; b n'a pas encore commencé
- pen[i][j][2]=false; a est en phase d'increasing
- pen[i][j][3]=false ; b est en phase d'increasing
- lettres[i][j][0] = nombres de lettres contenues dans a modifiées
- lettres[i][j][0] = nombres de lettres contenues dans b modifiées

(lettres est utile pour savoir quand le mot est « en finition », au sens où un hyphen ne signifiera plus suppression ou insertion, et qu'il ne faudra pas comptabiliser les pénalités)

On initialise les 3 matrices pour i =0 et j =0.

3 cas possibles, dans l'alignement final
- a[i-1] et b[j-1] sont appariés
  ⇒ pas de pénalité, on calcule simplement le score.
- a[i-1] est apparié avec un – .
  ⇒ sans pénalité si b n'a pas commencé ou si b termine (pen[i-1][j][1] || (lettres[i-1][j][1]==m) ==true).
  ⇒ avec la pénalité incr si b est en phase d'increasing (pen[i-1][j][3]==true).
  ⇒ avec la pénalité open si b correspond à un premier hyphen (pas dans la phase de début, ni de fin), soit le cas restant.
- b[j-1] est apparié avec un –.
  ⇒ sans pénalité si a n'a pas commencé ou si a termine (pen[i][j-1][0] || (lettres[i][j-1][0]==n)==true.
  ⇒ avec la pénalité incr si a est en phase d'increasing (pen[i][j-1][2]==true).
  ⇒ avec la pénalité open si a correspond à un premier hyphen (pas dans la phase de début, ni de fin) .

On calcule les 3 scores en et on prend le maximum qui est le plus probable biologiquement
Puis on se place dans ce cas là et on remplit t, res1 et res2 en fonction (voire commentaires) et on ajuste les valeurs de pen et lettres.

- Comme on considère tous les cas possibles dans cette boucle, l'algorithme est réalisable (avec comme hypothèse que tous les éléments des 3 matrices du rectangle inférieur de (i,j) sont connus). Il se termine car on diminue progressivement les valeurs de i et j jusqu'à avoir i ou j=0 qu'on a initialisés.
- Il renvoie le résultat demandé au vu de la relation de récurrence que l'on vient d'expliciter car on considère tous les cas possibles et on renvoie celui qui maximise le score à l'étape (i,j). Par récurrence on aura bien le score maximal à la fin de l'exécution.
- La complexité est constante car comme les matrices du carré inférieur sont remplies on a accès aux valeurs nécessaires pour la récurrence en temps constant et ensuite on a juste faire quelque comparaisons, concaténations et assignations.

ALGORITHME PRINCIPAL SCOREPENALTY

Input : les 2 séquences à analyser et les coûts de pénalité.
Output : a modifié et b modifié sous forme d'un tableau de String (respectivement 1ère et 2ème lignes) qui maximise le score de l'alignement en prenant en compte les pénalités.

Comme précédemment, on applique la fonction auxiliaire par ligne et colonne croissante. Comme la 1ere ligne et la 1ère colonne sont connues (comme décrit ci dessus), on va pouvoir remplir les trois matrices t, res 1, res 2, pen et lettres par rectangles de tailles croissantes grâce à la fonction ScorePenaltyAux. L'algorithme termine car on parcourt les matrices de taille n*m et à chaque itération on effectue les algorithmes auxiliaires qui eux-mêmes terminent. L'algorithme renvoie le résultat demandé. Ceci vient de la justesse de l'algorithme auxiliaire.
On voit clairement sur l'algorithme Java que la complexité temporelle est en O(nm) car deux boucles for de longueur n+1 et m+1 respectivement sont imbriquées et, à l'intérieur de ces deux boucles, est appliquée la fonction ScorePenaltyAux qui a un coût constant. La complexité spatiale est également un O(n*m) car on stocke nos données dans 3 matrices de taille (n+1)*(m+1), et 2 matrice de tailles (n+1)*(m+1)*4 et (n+1)*(m+1)*2.

Task6

Le but ici est d'essayer d'aligner seulement certaines parties des séquences et non pas les séquences en entier. On cherche donc les deux sous-séquences de a et b qui vont permettre le meilleur score (au sens qu'il est calculé par notre fonction ScorePenalty grâce à la fonction auxiliaire ScorePenaltyAux).

ALGORITHME PRINCIPAL LOCALALIGNEMENT2

Input : les 2 séquences à analyser et les couts de pénalité
Output : un alignement local entre a et b

Nous avons commencé par réaliser un ALGORTIHME TRES NAIF en terme de complexité spatiale. Je vais décrire l'algorithme plus intelligent LocalAlignment2 (String a, String b, int incr, int open).

Notre algorithme calcule tous les scores d'alignements locaux entre a(i…i2) et b(j…j2). Pour i,i2,j,j2 fixés, il calcule le score des sous séquences a(i…i2), b(j…j2) grâce à la fonction ScorePenaltyAux qui s'appliquent aux sous-séquences de a et b qui commencent respectivement en i et en j (notées ai et bj). On a donc accès à t[i2][j2], s'il est supérieur que le maximum courant noté opt, alors on attribue à opt cette valeur et on retient dans les 2 séquences loc1 et loc2 les 2 alignements de ai et bj. En bouclant sur tous les i, j, i2,j2 accessibles on a dans opt le score de l'alignement maximal et dans loc1 et loc2 l'alignement en tant que tel. L'algorithme termine car il effectue 4 boucles de longueurs finies et dans chaque itération on applique ScorePenaltyAux qui termine.
Le résultat est celui attendu par définition de l'énoncé puisqu'on a parcouru tous les cas possibles pour i,j,i2,j2 et on ne retient que celui correspondant au score maximale.
La complexité est en $O(n^2*m^2)$. C'est flagrant sur le code Java car on a 4 boucles for imbriquées et chaque itération effectue 5 affectations, 1 comparaison et la fonction ScorePenaltyAux qui est elle-même en temps constant. En terme d'espace, on stocke nos données t, res1, res2, pen, lettres. Ces matrices sont initialisées au départ puis on les réutilise à chaque fois pour les nouveaux calculs (pour une nouvelle valeur de (i,j)). La complexité spatiale est donc en O(n*m).

On pourrait se demander s'il n'est pas possible d'améliorer la complexité temporelle, en renforçant l'initialisation (tous les mots vides avec n'importe quel sous-séquence de a ou b définissent d'emblée un unique appariement/score possible), et le nombre de données stockées. Néanmoins, puisque l'on ne fait jamais appel deux fois à la même valeur dans les relations de récurrence dans l'algorithme ci-dessus, cela ne paraît pas être forcément être une bonne idée. Le seul point délicat dans l'algorithme et que l'on réinitialise de nombreuses fois les colonnes et lignes (nm fois), ce qui induit à chaque fois un coût en O(n+m). Cela ne modifie néanmoins pas la complexité. Nous ne pensons donc pas qu'il existe une technique de programmation dynamique significativement plus efficace en temps et en espace.

Task 7

Algorithme global demandé :

Input : Séquences g, t et un seuil th
Output : tous les indices de t qui correspondent à un appariement parfait entre un élément de Sg et un sous-mot de t

QUESTION THEORIQUE : COMMENT A-T-ON STOCKE G ?

On a choisi un Keyword tree pour stocker Sg. L'idée est que chaque nœud a un nombre de fils compris entre 0 et le nombre de lettres de l'alphabet considéré. Toutes les branches ont la même profondeur, k, et la présence d'une branche signifie la présence du mot obtenue en lisant les lettres de la racine à la feuille de cette branche dans le mot g. Ainsi, l'arbre possède un nombre de nœuds majorés par |Alphabet|^k. Ceci impliquera en particulier que décider si un sous-mot de t est dans Sg s'effectuera en temps constant. En pratique néanmoins, on voit que cette constante est élevée.

RETOURNARBRE

La façon efficace de stocker tous les sous-mots de g de taille k est de mettre le dictionnaire= {sous-mots de taille k dans g} sous forme d'arbre. Nous justifierons ce choix dans l'explication de la section suivante. On a donc fait une fonction auxiliaire retournarbre(String g, int k) qui retourne cet arbre (défini au sens de la classe arbre.java) à partir de la séquence mère g et de la taille k des sous-mots à considérer. La complexité est O(nk). En effet, on effectue une boucle de taille (n-k+A1) et à chaque itération on ajoute une nouvelle sous séquence de taille k (de telle sorte qu'on les a toutes). La fonction ajout a un coût k.

ESTDANSSG

Nous n'avons pas créé l'ensemble Sg décrit dans l'énoncé mais notre fonction estdansSg permet de déterminer si un mot de taille k, noté tk, est dans l'ensemble Sg. L'algorithme parcourt les branches de l'arbre. Lorsqu'il descend sur une branche, il calcule la valeur du score entre la lettre de la branche et la lettre de tk correspondante. Il l'ajoute à la valeur courante du score total. À chaque feuille est donc calculé un score d'alignement entre tk et le mot de g correspondant à la descente de la branche parvenant à la feuille. Finalement, il faut que tk et le mot de g considéré soient assez proches selon le critère que le score final >= autoscore de tk avec lui-même. Si tel est le cas le booléen renvoyé est true car tk appartient a Sg. L'algorithme :

- Termine car il parcourt tous les éléments de l'arbre de mots de g qui est finie car g est de longueur finie .
- Il retourne bien un booléen qui n'est vrai que si un des sous-mots de g est proche de tk au sens du critère pour Sg.
- Complexité : Le choix de l'arbre se justifie ici. En effet, le fait d'avoir stocker les sous-mots de g sous forme d'arbre permet d'avoir une complexité pour cette fonction estdansSg en O(k*taille de l'alphabet). Si on avait simplement parcouru g qui est une String on aurait eu un cout O(k*taille (g)), ce qui est bien moins intéressant.
- Remarquons néanmoins que le coût théorique constant ci-dessus peut s'avérer en pratique très élevé.

L'ALGORITHME PRINCIPAL INDICES

Input : les séquences g, t, et les nombres données th et k
Output : la liste des indices de t qui correspondent au début de correspondances parfaites entre un élément de Sg et un sous-mot de t

Pour cela il analyse toutes les sous-séquences tk de taille k de t et regarde si elles sont dans Sg (grâce à notre fonction estdansSg). Si tel est le cas, on ajoute l'indice à la liste chainée output.

- Il termine car t est de taille finie et l'algorithme parcourt tout t afin d'analyser toutes les sous-séquences tk.
- La complexité est un O(n=taille(t)) si k et la taille de l'alphabet sont considérés comme constants. En effet, on parcourt la séquence t et pour chaque itération on effectue estdansSg, on ajoute un indice à une liste et autres fonctionnalités de coût constant.

Task 8

Il s'agit d'effectuer la même recherche que dans la question 7, et d'effectuer successivement deux tâches supplémentaires. Premièrement, pour chaque matching entre deux sous-mots de taille k de t et g, étendre ces deux sous-mots à leurs deux extrémités tant que cela fait croître le score d'alignement. Deuxièmement, une fois ce nouveau matching obtenu, on le confronte à un score indicateur, thl*alignement(g,g).
Pour mener à bien ce travail, on modifie la structure de l'arbre stockant les sous-mots de g, en ajoutant à chacune des feuilles les indices où l'on peut trouver le début des mots correspondants à la branche terminant par cette feuille. Dès lors, on construit en parcourant les séquences de t de taille k un Hashmap dont les clés sont les indices de t et les valeurs les indices de matching associés dans g, sous forme de liste. Pour chacun de ces indices (clés), et les valeurs associés, on effectue l'extension des extrémités, puis si la condition de score est respectée, on modifie la clé et la valeur en conséquence, en faisant attention de ne pas rajouter à une clé une valeur sur laquelle elle pointe déjà.

Task 9

On s'intéresse à la prédiction du repliement 2D d'une protéine étant donné sa séquence 1D. Pour simplifier, la molécule est un enchaînement de lettres sur l'alphabet {P,H} où seules les liaisons H participent au score : chaque H qui se retrouve dans une case de la grille 2D adjacente à une case où se trouve un autre H augmente le score de 1, pourvu que les deux H ne soit pas dans l'enchaînement linéaire de la molécule.

La problématique est compliquée, et nous ne prétendons seuls pas avoir résolu le problème. Ci-dessous sont présentés des éléments de réponse synthétisés depuis la littérature à ce sujet Nous n'avons rien implémenté étant donné que nous n'avons pas de certitudes sur un algorithme complet répondant à la question.

La considération première est que deux éléments ne peuvent se retrouver en adjacence sur la grille que s'ils sont séparés d'un nombre pair de lettre dans l'enchaînement linéaire de la molécule. En effet, partant d'une case (i,j), avec un nombre pair d'opération, on aboutit à une case de la forme (i+k,j+k') avec k+k' pair. On ne peut donc pas aboutir aux cases (i+1,j),(i-1,j),(i,j-1),(i,j+1). Appuyons nous sur ce constat pour construire une structure pertinente sur nos séquences linéaires.

Si la séquence commence par un H par exemple, il est illusoire de tenter de coupler ces H avec tous les H suivants qui sont séparés d'un nombre impair de P. Regroupons donc tous ces premiers H. Par exemple, sur la séquence HPHPPPHPH-PP-HPH-H-PPPP-HPPPH, où les tirets sont là pour aider l'illustration, nous prendrons le groupe HPHPPPHPH. Appelons bloc, un tel groupe. Il est possible au sein d'une séquence de définir plusieurs blocs. Puisque la parité est le critère de possibilité d'appariement, nous sommes tenté de séparer les blocs par des groupes pairs de 0, que l'on pourra appeler inter-blocs. Ainsi dans l'exemple ci-dessus, HPHPPPHPH est le premier bloc, PP est le premier inter-bloc, HPH le second bloc, H le troisième bloc, PPPP le deuxième inter-bloc, HPPPH le dernier bloc. Par souci de séparer les blocs par des inter-blocs, nous dirons que si deux blocs sont séparés d'un élément vide, comme ci-dessus le 2ème et 3ème bloc, ils sont en réalité séparé d'un inter-bloc de taille 0.

Nous obtenons ainsi une description unique et complète de la séquence 1d. En effet, tous les 0 consécutifs sont dans des blocs ou inter-blocs : blocs s'ils sont en nombre impair, inter-blocs sinon. Tous les H sont dans des blocs.

Voyons en quoi cette modélisation est utile. Prenons deux blocs consécutifs d'une séquence donnée, séparés d'un inter-bloc. Prenons un H dans chacun de ces blocs. Il est très visible que ces deux H sont séparés d'un nombre pair d'éléments. Néanmoins, si l'on avait pris un H

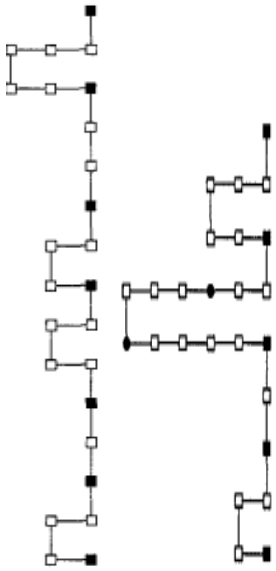
dans le 1er bloc et un H dans le 3ème bloc, ils auraient été séparés d'un nombre impair d'éléments. De manière générale, montrons que si l'on prend un H dans le i-ème bloc et un H dans le j-ème bloc, où i<j, ces H sont séparés d'un nombre d'éléments dont la parité et celle de j-i+1. En effet, comptons les éléments depuis le H du bloc i jusqu'à l'élément précédent le H choisi du bloc j. Tous les éléments comptés dans les inter-blocs ne modifient pas la parité (ils contiennent un nombre pair de P). Tous les blocs strictement croisés contiennent un nombre impair d'éléments. Cela fait déjà une parité de équivalente à celle de j-i-1. De l'élément succédant le H du bloc i jusque la fin du bloc i, il y a un nombre pair d'éléments. De même pour les éléments du début du bloc j jusqu'à l'élément précédant le H choisi dans le bloc j.
Ainsi, un H du bloc i ne pourra potentiellement s'affilier qu'avec les H des blocs j tels que j-i est impair. Cela nous incite fortement à poser de classe d'équivalence de blocs. Appelons 1-blocs et 2-blocs les deux classes obtenus. Les H des 1-blocs ne peuvent que s'affilier avec les H des 2-blocs (potentiellement). L'exemple ci-dessus présente les 1-bloc HPHPPPHPH et H ; les 2-blocs HPH et HPPPH, et les inter blocs PP, _, et PPPP.

Notons N le nombre de H dans les 1-blocs et M celui dans les 2-blocs. Quitte à inverser les classes d'équivalence, supposons que N>M, ou que si N=M, Les 1-blocs contiennent plus de H en début ou fin de séquence que les 2-blocs. On peut dès lors majorer l'énergie optimale du repliement 2D d'une séquence s :

$$OPT(repliement(s)) <= 2N(s) + 3F(N(s))$$

Où F(X(S)) est le nombre d'éléments H terminaux appartenant à des 1-blocs. F(N(s)) vaut 0, 1, ou 2. En effet, il, y a plus de H dans la classe des 1-blocs, et ceux-ci, s'ils ne sont pas terminaux, ne peuvent s'affilier qu'avec au plus deux H des 2-blocs (car deux cases au maximum disponibles), tandis que les éléments terminaux ont au plus trois affiliations possibles.

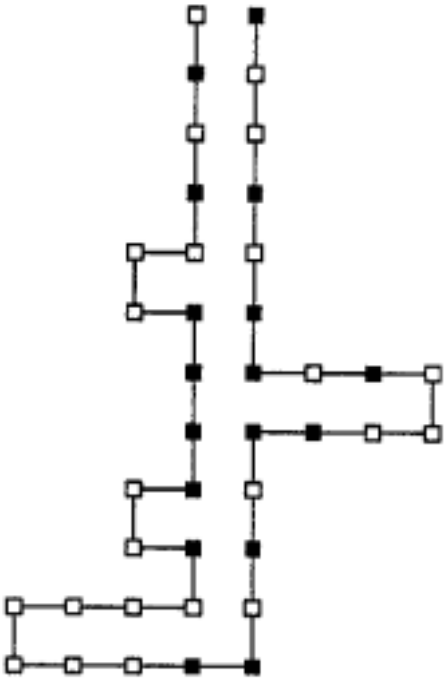
Tentons d'utiliser la modélisation de blocs, pour tracer un chemin en 2D. Définissons une structure dite de serpent. Il s'agit de se donner une verticale comme axe majeur et de placer de bas en haut nos éléments comme suit. Pour un bloc donné, on place un H sur la verticale, puis on place les P précédents le prochain H sur la gauche de H, de sorte à ce que sur la verticale, on lise de bas en haut H, P, H. Lorsqu'on croise un inter bloc, on effectue un aller retour sur la gauche.



Pour y voir plus clair, voici un exemple sur la séquence HPPPHPHPPPPHPPPHPPHPPPPPH où les P sont les carrés blancs et les H les noirs. Le premier bloc est HPPPHPH et on le lit jusqu'au 3ème carré noir. On voit ensuite une boucle de carrés blancs qui correspondent à un inter-bloc, etc.

Dans le 2ème exemple à droite, on pousse plus loin la structure. On divise la séquence 1D en 1-blocs et 2-blocs. On modifie la structure précédente pour obtenir la structure en serpent où les 2-blocs sont traités comme des inter-blocs, c'est-à-dire qu'ils forment entièrement une boucle avec les inter-blocs. On voit ici le premier 1-bloc HPPPHPH déroulée comme dans le premier exemple, puis l'inter-bloc PPPP, le 2-bloc HPPPH, et l'inter-bloc PP qui forme une boucle. On parle de serpent 1-dominants puisque les 2 blocs sont traités comme des inter-blocs. On construit de même les serpents 2-domintants.

À quoi bon cette structure en serpent qui produit un score nulle ? L'idée, c'est de faire un appariement, à partir d'une séquence s, entre deux de ces serpents. Pour cela, on choisit un élément très précis de la séquence, une ancre, et l'on obtient les serpents des parties gauches et droites (respectivement 1-dominants ou 2-dominants), délimitées par cette ancre, que l'on replie sous forme de fer à cheval.



Tout est dans le choix de l'ancre qui va délimiter les séquences gauche g et droite d de la séquence s.
Pour cela, on implémente un algorithme (en complexité linéaire), qui part de la première lettre, et parcourt la chaîne en comptant les H des 1-blocs déjà rencontrés et les H des 2-blocs déjà rencontrés. Notons N1(a), M1(a) ces quantités, où a est la position courante de l'ancre. En incrémentant peu à peu, il est facile de voir que l'on arrive à un point critique où
N2(a) <= M-E(M+1/2) et N1(a)>=E(X/2)
Ou N2(a) <= M-E(M/2) et N1(a)>=E(X+1/2)
Où E désigne la partie entière supérieure.
On choisit cette ancre critique en définitive.

Puis on replie les deux séquences et on les place en serpents 1-dominants et 2-dominants.

Toutes les opérations envisagées sont successives et linéaires, donc l'algorithme théorique envisagé est également linéaire en la longueur de la séquence

Il reste à voir que l'on obtient ainsi un score supérieur à E(N/2). Rappelons que l'on avait N>=M. Asymptotiquement, cela nous donne donc une 1/4-approximation de la conformation optimale !

En, réalité, cette structure en fer à cheval offre la possibilité à de nombreuses modifications qui peuvent encore améliorer le score. En effet localement, on peut déplacer les boucles sur les verticales à peu près librement et jouer sur le score de cette façon. Sans rentrer dans le détail, nous proposons simplement une illustration de ces dernier propos.