Mini-Projet : Alignement de séquences
LU3IN003 - Sorbonne Université

Amann Emmanuelle & Malonda Clément

4 décembre 2019

Table des matières

1 Introduction **2**

2 Méthode naïve par énumération **2**

3 Programmation dynamique **3**

3.1 pour le calcul de la distance d'édition 3

3.2 pour le calcul d'un alignement optimal 5

4 Amélioration de la compléxité spatiale du calcul de la distance **8**

5 Amélioration de la compléxité spatiale du calcul d'un alignement optimal par la méthode "diviser pour régner" **9**

1 Introduction

Question 1 .
Soient $(\bar{x}, \bar{y})$ et $(\bar{u}, \bar{v})$ deux alignements respectivement de (x, y) et (u, v). $\bar{x}$ et $\bar{y}$ sont alignés et donc sont de même longueur ($|\bar{x}| = |\bar{y}|$). De même pour l'alignement $(\bar{u}, \bar{v})$, ($|\bar{x}| = |\bar{y}|$).
A partir de ces deux affirmations, nous pouvons dire que la concaténation de $\bar{x}$ et $\bar{u}$, $\bar{x}.\bar{u}$ ainsi que la concaténation de $\bar{y}$ et $\bar{v}$, $\bar{y}.\bar{v}$ sont de même longueur.

Question 2 .
Soient $x \in \sum^*$ un mot de longueur n et $y \in \sum^*$ un mot de longueur m, la longueur maximale d'un alignement de (x, y) est $n + m - 1$.
On prend par exemple $x = ATCG$ et $y = GCTGA$, l'aligement de longueur maximale est :

$$\bar{x} : ATCG\_\_\_\_$$
$$\bar{y} : \_\_\_GCTGA$$

2 Méthode naïve par énumération

Question 3 .
Soit $x \in \sum$ un mot de longueur n, on souhaite ajouter exactement k gaps dans ce mot pour obtenir le mot $\bar{x}$.
Il existe, sur un ensemble de $n + k$ valeurs, $n^k + 1$ combinaisons possibles.

Question 4 .
$|\bar{x}| = n \leq m = |y|$ où $\bar{x}$ est le mot x avec k gaps.
On a donc $|\bar{x}| - |y| = k_y$ le nombre de gaps que l'on doit insérer dans le mot y.

Question 5 .

Question 6 .
La complexité spatiale qu'aurait un algorithme naïf qui énumère tous les alignements de deux mots serait en O de la taille du plus grand alignement multiplié par la nombre d'alignements.

Tâche A Il est possible de résoudre les instances fournies en moins d'une minute pour celles qui sont de taille 10 et 12.
La consommation mémoire nécessaire au fonctionnement de cette méthode est d'environ 4900Ko. Nous utiliserons la commande suivant pour utiliser le programme :

```
time python3 TacheA.py un_instance.adn
```

3 Programmation dynamique

3.1 pour le calcul de la distance d'édition

On considère $(x, y) \in \sum^* \times \sum^*$ un couple de mots de longueurs respectives n et m.

Question 7 .
On distingue trois cas possible :
- si $\bar{u}_j =$ - alors $\bar{v}_l = y_j$ car par définition il n'est pas possible d'avoir deux gaps face à face dans un alignement.
- si $\bar{v}_l =$ - alors $\bar{u}_l = x_i$ car par définition il n'est pas possible d'avoir deux gaps face à face dans un alignement.
- si $\bar{u}_l \neq$ - et $\bar{v}_l \neq$ - alors $\bar{v}_l = \bar{u}_l$ car le seul moyen de ne pas avoir de gap en une position l de la liste est que les lettres des deux mots à cet endroit soient les mêmes.

Question 8

$$C(\bar{u}, \bar{v}) = C(\bar{u}_{[1...l-1]}, \bar{v}_{[1...l-1]}) + \begin{cases} c_{ins} \text{ si } \bar{u}_l = - \\ c_{del} \text{ si } \bar{v}_l = - \\ c_{sub} \text{ sinon} \end{cases}$$

Question 9 .
Soit $(\bar{u}, \bar{v})$ un alignement de $(x_{[1...i]}, y_{[1...j]})$
On sait que

$$\begin{array}{rcl} D(i,j) & = & d(x_{[1...i]}, y_{[1...j]}) \\ D(i,j) & = & min\{\bar{u}, \bar{v}\} \end{array}$$

Dans le cas n°1 : on a $\bar{v}_l = y_j$ et $\bar{u}_l = -$, ce qui une insertion. Puis, pour calculer la distance des éléments précédents, nous allons devoir prendre l'élément précédent dans y mais pas dans x car la place x_i est occupé par un gap.

Dans le cas n°2 : on a $\bar{u}_l = x_i$ et $\bar{v}_l = -$, ce qui une supression. Puis, pour calculer la distance des éléments précédents, nous allons devoir prendre l'élément précédent dans x mais pas dans y car la place y_j est occupé par un gap.

Dans le cas n°3 : il n'y a aucun gap donc on prendra dans les deux mots les lettres précédentes $x_{[1...i-1]}$ et $y_{[1...j-1]}$.

Question 10 .
$D(0, 0) = 0$ car les lettres d'un mot sont numérotées de 1 à n avec n la taille du mot, la position $(0, 0)$ correspond au mot vide.

Question 11 .
$D(0,j) = m \times c_{ins}$ l'alignement du mot vide w avec un mot de taille m se fait par l'ajout de m gaps dans w face à chaque lettre
$D(i,0) = n \times c_{dsl}$ l'alignement du mot vide w avec un mot de taille n se fait par l'ajout de n gaps dans w face à chaque lettre

Question 12 .

```
DIST_1 (x, y) :
    n <- x.taille + 1
    m <- y.taille + 1
    T <- entier[n][m]

    pour i allant de 0 a n faire :
        pour j allant de 0 a m faire  :
            si i==0 faire :
                T[i][j] <- j*2

            sinon si j==0 faire:
                T[i][j] <- i*2
            sinon :
                T[i][j] <- min (T[i-1][j]+2, T[i][j-1] + 2, T[i-1][j-1] + \\
                c_sub (x[i-1], y[j-1]))

    return (T[n-1][m-1], T)
```

Nous retournons le couple constitué de la distance d'édition et du tableau de toutes les valeurs de D car nous avons besoin de ce tableau dans SOL_1.

Question 13 .
L'algorithme DIST_1 utilise une matrice de taille $n \times m$, sa complexité spatiale est en $\Theta(n \times m)$.

Question 14 .
L'algorithme DIST_1 est constitué de deux boucles imbriquées. La boucle intérieure ne fait que des opérations élémentaires en $O(1)$ donc la complexité temporelle est en $O(n \times m)$.

3.2 pour le calcul d'un alignement optimal

Question 15 .
On a $i > 0$ et $D(i,j) = D(i-1,j) + c_{del}$
Si $\exists(\bar{s},\bar{t}) \in Al^*(i-1,j) + c_{del}$
Montrons alors que $(\bar{s}.x_i) \in Al^*(i,j)$
$Al^*(i-1,j)$ veut dire qu'il existe un alignement $(\bar{s},\bar{t})$ de $(x_[1...i-1], y_[1...j])$ tel que $C(\bar{s},\bar{t}) = D(i-1,j)$. Par définition c_{del} est le coût d'une supression qui consiste à encoder un gap dans $\bar{y}$ pour marquer la supression de la lettre de x qui est parallèle à ce gap dans $\bar{y}$. Par définition on va donc ajouter un élément dans $\bar{y}$ sans avancer dans le mot y et on parallèle on avance d'une lettre dans x.
On aura donc $D(i-1,j) + c_{del} = D(i,j)$
Donc il existera un alignement $(\bar{u},\bar{v})$ tel que $C(\bar{u},\bar{v}) = D(i,j)$ avec $\bar{u} = \bar{s}.x_i$ et $\bar{v} = \bar{t}.-$
Alors on a bien $(\bar{s}.x, \bar{t}.-) \in Al^*(i,j)$.

Question 16 .

```
SOL_1 (x, y, T) :
    u <- ""
    v <- ""
    n <- x.taille
    m <- y.taille
    i <- n
    j <- m

    si (n==0) alors :
        u <- m*"-"
        v <- y
        return (u,v)
    sinon si (m==0) alors :
        u <- x
        v <- n*"-"
        return (u,v)

    tant que (i >= 1) or (j >= 1) faire :

        si (i == 1) alors :
            u <- x[i-1] + u
            si (j==1) faire:
                v <- y[j-1] + v
                return (u,v)
            sinon :
                tant que (j>=1) faire :
```

```
                    u <- "-" + u
                    v <- y[j-1] + v
                    j <- j -1
                v <- y[j-1] + v
                retourne (u, v)

        sinon si j == 1 alors :
            v <- y[j-1] + v
            tant que (i>1) alors :
                v <- "-" + v
                u <- x[i-1] + u
                i <- i -1
            u <- u[i-1] + u
            retourne (u, v)

        sinon si (T[i][j]) == (T[i-1][j] + 2) alors :
            u <- x[i-1] + u
            v <- "-" + v
            i <- i - 1

        sinon si (T[i][j]) == (T[i][j-1] + 2) alors :
            u <- "-" + u
            v <- y[j-1] + v
            j <- j - 1

        sinon si T[i][j] == (T[i-1][j-1] + cout.c_sub (x[i-1], yfaire [j-1]))
            u <- x[i-1] + u
            v <- y[j-1] + v
            i <- i - 1
            j <- j - 1
    retourne (u, v)
```

Question 17 .
La résolution du problème ALI, par combinaison des algorithmes DIST_1 et SOL_1, se fait avec un complexité en $O(n \times m)$ car chacun des deux algorithmes et que SOL_1 fait appelle une fois à DIST_1.

Question 18 .
La complexité spatiale de DIST_1 est en $\Theta(n \times m)$ et celle de SOL_1 est en $O(n + m)$.

Tâche B .

Nous utiliserons la commande suivant pour utiliser le programme :

```
time python3 sol_1.py un_instance.adn
```

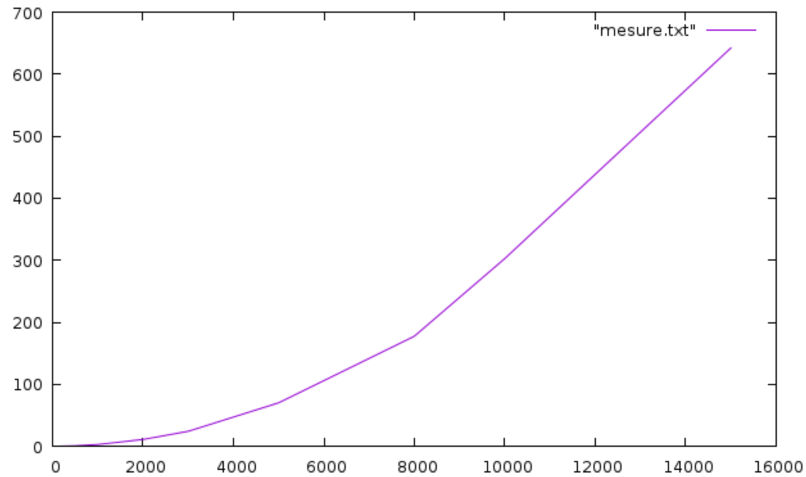

Figure 1 – Courbe de consommation de temps (en s) CPU en fonction de la taille |x|

4 Amélioration de la compléxité spatiale du calcul de la distance

Question 19

Question 20 .

```
DIST_2 (x, y):
    n <- x.taille
    m <- y.taille
    T[2][m]
    for i in range (n) :
        for j in range (m) :
            if i== 0:
                T[i][j] <- 2*j
            if j==0 :
                T[i][j] <- 2*i
            T[i-1][j] <- T[i][j]
            T[i][j] <- min (T[i-1][j] + 2, T[i][j-1]+2, T[i-1][j-1] \\
            + cout.c_sub(x(i-1), y(j-1))
```

Tâche C Nous utiliserons la commande suivant pour utiliser le programme :

```
time python3 sol_1_2.py un_instance.adn
```

5 Amélioration de la compléxité spatiale du calcul d'un alignement optimal par la méthode "diviser pour régner"

Question 21 Pseudo-code de la fonction mot_gaps :

```
mot_gaps(k):
    res <- ""
    pour i allant de 0 a k-1 faire:
        res <- res + "-"
    retourne res
```

Question 22 Pseudo-code de la fonction align_lettre_mot :

```
align_lettre_mot(lettre, mot):
    i <- 0
    tant que lettre != mot[i] et i < mot.taille:
        i <- i + 1
    si i < mot.taille alors:
        x <- mot_gaps(i) + lettre + mot_gaps(mot.taille - i - 1)
    sinon:
        mot <- mot + "-"
        lettre <- mot_gaps(mot.taille) + lettre
    retourne(lettre,mot)
```

Question 23 .
On a les coûts suivants :

$$c_{ins} = c_{del} = 3 \text{ et } c_{sub} = \begin{cases} 0 \text{ si } a = b \\ 5 \text{ si } a \text{ et } b \text{ sont deux consonnes/voyelles distinctes} \\ 7 \text{ sinon} \end{cases}$$

On a : x = BALLON et y = ROND donc x^1 = BAL, $x^2 = LON$, y^1 = RO et y^2 = ND.

$$\bar{s} : B\_A\_L$$

$$\bar{t} : \_R\_O\_$$

$$\bar{u} : LON\_$$

$$\bar{v} : \_\_ND$$

Les coûts de ces deux alignements sont respectivement de 13 et 9, ce qui fait un coût total de 22.
Or il est possible de faire un autre alignement :

$$\bar{s} : \_BALLON\_$$

$$\bar{t} : R\_\_\_OND$$

Le coût de l'alignement si dessus est de $5 \times 3 + 5 = 20$ soit un coût inférieur à celui réalisé avec la coupure.

Question 24 Pseudo-code de la fonction SOL_2 en considérant que l'on possède une fonction coupure

```
SOL_2(x, y):
    si y.taille == 0 alors:
        retourne (x,mot_gaps(x.taille))
    si x.taille == 1 et si y.taille == 1 alors:
        si x == y alors :
            retourne (x,y)
        sinon retourne (x+"-", "-"+y)
    sinon:
        res1 <- SOL_2(x[:x.taille/2],y[:coupure(x,y)])
        res2 <- SOL_2(x[x.taille/2:],y[coupure(x,y:)])
        retourne (res1[0]+res2[0], res1[1]+res2[1])
```

Question 25 Pseudo-code de la fonction coupure :

```
coupure (x, y, T) :
    n <- len(x)+1
    m <- len(y)+1
    p <- (n-1)//2
    I <- np.zeros((n, m))
    i <- 1
    j <- 1

    pour i allant de 1 a n faire:
        pour j allant de 1 a  m faire :

            si (i<p) faire :
                I[i][j] <- 0

            sinon si (i==p) faire :
                I[i][j] <- j

            sinon :
                s <- min (T[i-1][j], T[i][j-1], T[i-1][j-1])

                si (s == T[i-1][j]) faire :
                    I[i][j] <- I[i-1][j]
```

```
            sinon si (s == T[i][j-1]) faire :
                I[i][j] <- I[i][j-1]

            sinon si (s == T[i-1][j-1]) faire :
                I[i][j] <- I[i-1][j-1]
    return (I[n-1][m-1])
```

Question 26 .
La complexité spatiale de la fonction coupure est $\Theta(m)$ car elle utilise un tableau de taille $2 \times m$ pour stocker les valeurs $D(i, j)$ données par l'équation de réccurrence et que l'on utilise uniquement un tableau à deux lignes car se sont les seuls utiles au moment du calcul.

Question 27 .
La complexité spatiale de la fonction SOL_2 est en $O((n+m)^2)$ car elle utilise deux chaînes de caractères pour stocker solution est utilise des appels récursifs.

Question 28 .
La complexité temporelle de la fonction coupure est en $O(n \times m)$ car elle parcourt un tableau de taille $2 \times m$ mais celui-ci doit être mis-à-jour et donc parcouru à nouveau jusqu'à un maximum de n fois.

Tâche D .
Nous utiliserons la commande suivant pour utiliser le programme :

```
time python3 sol_2.py un_instance.adn
```

Question 29