Franck Deturche – Dura (DSIA)
Guillaume Gay (DSIA)

Rapport de projet

Algorithme anti-plagiat – Applied Algorithms

Notes :

Nous avons choisi d'utiliser python (bien qu'interprété). L'énoncé demandant un rendu compilable et cela étant possible avec python, nous avons préféré nous concentrer sur la partie algorithmique en utilisant un langage que nous maîtrisons mieux que le C.

Pour suivre l'évolution de notre projet : git clone *https://github.com/franckdeturchedura/Detection_plagiat.git*

Question 1 :

Notre but est de trouver le score d'alignement optimal (c'est-à-dire un **score d'alignement minimisé)** de deux séquences données x et y (suites de caractères). Le tout doit être fait par un algorithme en **O(|x| * |y|).**

Pour cela nous allons reprendre le principe d'un algorithme connu du cours : **Edit String**. Celui-ci a pour but, initialement, de donner le nombre d'opérations nécessaires pour obtenir, à partir de deux séquences différentes, des séquences identiques.

Les dites opérations autorisées sont Ins(), Del() et Sub(x,y) qui permettent respectivement d'obtenir le coût de l'insertion d'un caractère, le coût de la suppression d'un caractère et le coût de l'échange de caractères.

Pour calculer le score optimal d'alignement, on affecte à Del() et Ins() un coût de +1 (car plus le score est haut, pire est l'alignement). Pour Sub(), nous n'affectons aucun coût si les lettres des deux séquences sont les mêmes. Nous affectons un coût de +1 dans le cas contraire.

Ainsi, nous obtenons une matrice telle que pour tout i et tout j, t[i][j] donne la distance entre les séquences x_i, y_j, préfixes de x et y de longueurs i et j.

Pseudo Code :

Données :

- Séquence A : A
- Séquence B : B
- Del() : retourne 1
- Ins() : retourne 1
- Sub(x,y) : retourne 0 si x = y et 1 sinon

```
distance(A,B)  {
        lenA = longueur(A) #Taille de la séquence A
        lenB = longueur(B) #Taille de la séquence B
        T[0][0] = 0
        Pour i allant de 1 à lenA :
                T[i][0] = T[i-1][0] + Del()
        Pour j allant de 1 à lenB :
                T[0][j] = T[0][j-1] + Ins()
        Pour i allant de 1 à lenA :
                Pour j allant de 1 à lenB :
                        T[i][j] = min(
                                        T[i][j-1] + Ins(),
                                        T[i-1][j] + Del(),
                                        T[i-1][j-1] + Sub(A[i-1], B[j-1])
                                )
        retourner T
}
```

Comme pour Edit String, on a une complexité en lenA * lenB (soit |x|*|y|).

Question 2 :

Via l'algorithme précédent, nous obtenons pour deux séquences données x et y, la matrice t de taille (|x|+1)*(|y|+1), telle que t[i][j] est le score d'alignement optimal entre x_i et y_j.

Nous cherchons maintenant à obtenir, à partir de deux séquences données et de l'algorithme précédent, l'alignement optimal x',y' des deux séquences x et y.

L'algorithme doit être de **complexité linéaire**.

Tout d'abord, nous avons implémenté une fonction retournant, à partir de deux indices donnés et la matrice T renvoyée par l'algorithme ci-dessus, les indices suivants à utiliser pour « remonter » la matrice T et reconstituer x' et y'.

Soient x et y les deux séquences initiales.

Soit m = |x| et n = |y| et T la matrice associée aux séquences x et y.

En effet, pour trouver l'alignement optimal x',y', nous allons partir du score d'alignement optimal de X_m, Y_n (soit les deux séquences entières), à T[m][n].

Nous allons ensuite remonter la matrice en prenant les scores d'alignements minimaux (et donc optimaux) et en déduire les caractères de x' et y'.

L'algorithme retourne basiquement les indices des valeurs minimales de la matrice à partir d'un point donné. Pour cela il vérifie pour chaque cas les valeurs des 3 cases accolées.

Données :

- indice i de départ
- indice j de départ
- Matrice des distances T

```
retourne_indices(i, j, T)  {
        Si T[i-1][j] < T[i][j-1] et T[i-1][j] < T[i-1][j-1] :
                i = i-1
                retourner i, j

        Sinon Si T[i][j-1] < T[i-1][j] et T[i][j-1] < T[i-1][j-1] :
                j = j-1
                retourner i, j

        Sinon Si T[i-1][j-1] < T[i][j-1] et T[i-1][j-1] < T[i-1][j] :
                i = i-1
                j = j-1
                retourner i, j

        Sinon :
                i = i-1
                j = j-1
                retourner i, j
}
```

Une fois cela fait, nous pouvons implémenter l'algorithme d'alignement.

Pour cela, nous faisons, tant que nous ne sommes pas revenus au point initial (T[0][0]), des conditions sur la différence entre les indices actuels et les indices suivants calculés via l'algorithme ***retourne_indices()***. Une fois cela fait, nous avons la « direction » à prendre et n'aurons plus qu'à inverser les tableaux obtenus à la fin.

Ainsi, selon le résultat obtenu avec ces différences, on sait si, pour x' et y', on doit ajouter une lettre de la séquence de base ou étirer le texte avec un espace.

Les détails sont dans les commentaires de l'algorithme (position suivante horizontale et verticale -> tiret pour x' ou y' et lettre pour l'autre, position suivante diagonale -> lettres pour x' et y').

L'algo permettant de déterminer les indices suivants ayant une complexité constante, l'algo principal est bien de complexité linéaire.

Après tests sur différents textes et séquences de notre invention, l'algo retourne bien x' et y', cohérents et chacun pouvant être étirés.

Franck Deturche – Dura (DSIA)
Guillaume Gay (DSIA)

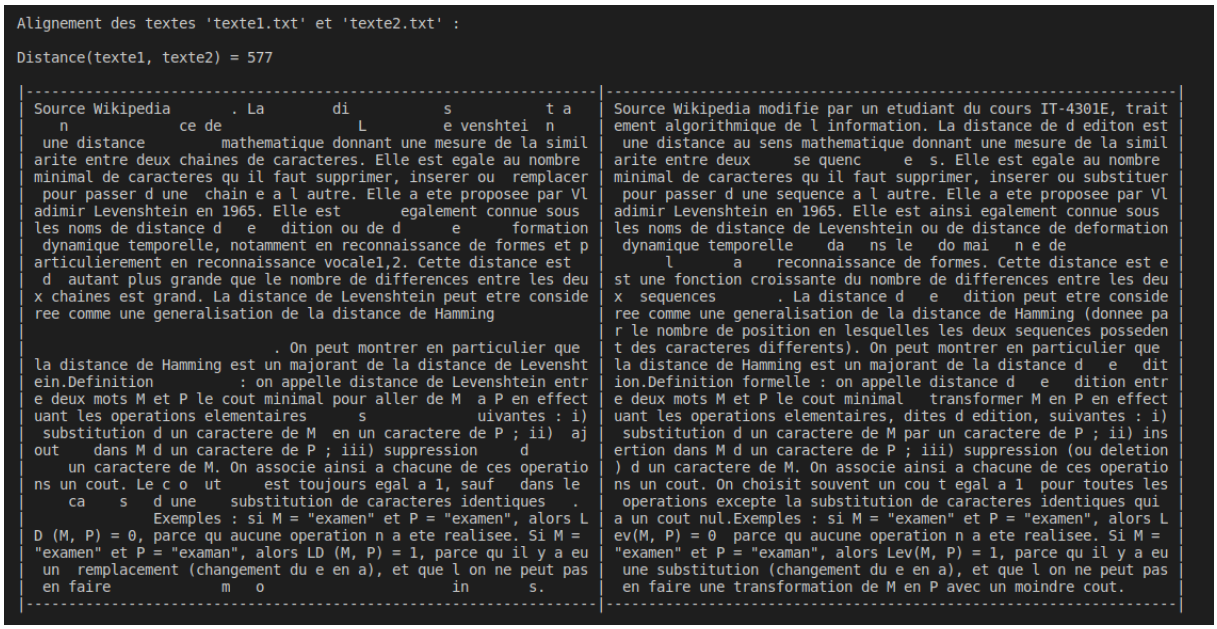
Données :

- Séquence A : A
- Séquence B : B
- retourne_indices(i,j,T) : fonction prenant i et j les indices actuels ainsi que T la matrice des distances. Retourne les indices suivants.
- longueur(a) : retourne la longueur de la string a
- distance(A,B) : retourne la matrice T

```
alignements_optimaux(A,B) {
    A_p = []  # là où nous stockerons A'
    B_p =[]   # là où nous stockerons B'

    i = longueur(A) # pour partir de la dernière ligne de la matrice T
    j = longueur(B) # pour partir de la dernière colonne de la matrice T

    i_p = 0    # contiendra la ligne suivante
    j_p = 0    # contiendra la colonne suivante

    T =  distance(A,B) # On instancie la matrice des scores optimaux

    Tant que i différent de 0 et j différent de 0 :  #tant qu'on n'est pas revenu
                                                        au point de départ
        i_p, j_p = retourne_indices(i,j,T)     #stocke dans i_p et j_p les indices
                                                  suivants

        Si i_p – i =-1 ET  j_p -j =-1 Alors {     # Cas où le score optimal
                                                     suivant est sur la diagonale
            A_p.append(A[i_p])  # On ajoute à A' la lettre correspondante
                                   de la séquence A
            B_p.append(B[j_p])  # On ajoute à B' la lettre correspondante
                                   à la séquence B
            i = i-1                # on met à jour l'indice des lignes
            j = j-1                # on met à jour l'indice des colonnes
        }
        Sinon si i_p-i =-1 ET j_p-j=0 Alors {   # Cas où le score optimal
                                                    suivant est sur la ligne
            A_p.append(A[i_p])
            B_p.append(«  »)        # on ajoute un espace (étirement) à B'
            i = i-1                 # on met à jour l'indice des lignes
        }
        Sinon Si i_p-i =0 ET j_p-j=-1 Alors { # Cas où le score optimal
                                                 suivant est sur la colonne
            A_p.append(«  »)
            B_p.append(B[j_p])
            j = j-1                      # on met à jour l'indice des colonnes
        }

    inverser(A_p)    # On remet les séquences à l'endroit car on a parcouru les
                        textes à l'envers
    inverser(B_p)
    retourner A_p, B_p
}
```

Question 3 :

On applique la fonction ***alignements_optimaux()*** aux fichiers ‘texte1.txt’ et ‘texte2.txt’ :



```
Alignement des textes 'texte1.txt' et 'texte2.txt' :

Distance(texte1, texte2) = 577

|-----------------------------------------------------------------|-----------------------------------------------------------------|
| Source Wikipedia       . La        di          s           t a  | Source Wikipedia modifie par un etudiant du cours IT-4301E, trait |
|    n            ce de              L         e venshtei   n     | ement algorithmique de l information. La distance de d editon est |
|  une distance          mathematique donnant une mesure de la simil | une distance au sens mathematique donnant une mesure de la simil |
| arite entre deux chaines de caracteres. Elle est egale au nombre  | arite entre deux     se quenc     e  s. Elle est egale au nombre  |
| minimal de caracteres qu il faut supprimer, inserer ou  remplacer | pour passer d une sequence a l autre. Elle a ete proposee par Vl |
|  pour passer d une  chain e a l autre. Elle a ete proposee par Vl | adimir Levenshtein en 1965. Elle est ainsi egalement connue sous  |
| adimir Levenshtein en 1965. Elle est       egalement connue sous | les noms de distance de Levenshtein ou de distance de deformation |
| les noms de distance d   e   dition ou de d      e      formation | dynamique temporelle    da   ns le   do mai   n e de             |
|  dynamique temporelle, notamment en reconnaissance de formes et p |      l       a    reconnaissance de formes. Cette distance est e |
| articulierement en reconnaissance vocale1,2. Cette distance est   | st une fonction croissante du nombre de differences entre les deu |
|  d  autant plus grande que le nombre de differences entre les deu | x  sequences        . La distance d   e   dition peut etre conside |
| x chaines est grand. La distance de Levenshtein peut etre conside | ree comme une generalisation de la distance de Hamming (donnee pa |
| ree comme une generalisation de la distance de Hamming            | r le nombre de position en lesquelles les deux sequences posseden |
|                                                                   | t des caracteres differents). On peut montrer en particulier que  |
|                          . On peut montrer en particulier que     | la distance de Hamming est un majorant de la distance d   e   dit |
| la distance de Hamming est un majorant de la distance de Levensht | ion.Definition formelle : on appelle distance d   e   dition entr |
| ein.Definition         : on appelle distance de Levenshtein entr | e deux mots M et P le cout minimal   transformer M en P en effect |
| e deux mots M et P le cout minimal pour aller de M  a P en effect | uant les operations elementaires, dites d edition, suivantes : i) |
| uant les operations elementaires      s            uivantes : i)  |  substitution d un caractere de M par un caractere de P ; ii) ins |
|  substitution d un caractere de M  en un caractere de P ; ii)  aj | ertion dans M d un caractere de P ; iii) suppression (ou deletion |
| out    dans M d un caractere de P ; iii) suppression     d      | ) d un caractere de M. On associe ainsi a chacune de ces operatio |
|    un caractere de M. On associe ainsi a chacune de ces operatio  | ns un cout. On choisit souvent un cou t egal a 1  pour toutes les |
| ns un cout. Le c o  ut     est toujours egal a 1, sauf   dans le  |  operations excepte la substitution de caracteres identiques qui  |
|     ca    s   d une    substitution de caracteres identiques   .  | a un cout nul.Exemples : si M = "examen" et P = "examen", alors L |
|             Exemples : si M = "examen" et P = "examen", alors L   | ev(M, P) = 0  parce qu aucune operation n a ete realisee. Si M =  |
| D (M, P) = 0, parce qu aucune operation n a ete realisee. Si M =  | "examen" et P = "examan", alors Lev(M, P) = 1, parce qu il y a eu |
| "examen" et P = "examan", alors LD (M, P) = 1, parce qu il y a eu |  une substitution (changement du e en a), et que l on ne peut pas |
|  un  remplacement (changement du e en a), et que l on ne peut pas  |  en faire une transformation de M en P avec un moindre cout.      |
|  en faire            m   o                    in       s.       |                                                                 |
|-----------------------------------------------------------------|-----------------------------------------------------------------|
```

Question 4 :

L’inconvénient de cette implémentation dans le cadre de la détection de plagiat dans un texte est qu’elle ne peut aligner correctement qu’un seul paragraphe ou ligne là où la plupart des textes sont organisés en plusieurs paragraphes.

On cherche donc à améliorer l’algorithme pour pouvoir détecter un plagiat entre plusieurs paragraphes d’un texte et pour cela on va utiliser le fait que l’algorithme de Levensthein peut être utiliser pour calculer la distance qui sépare autre chose que des caractères comme par exemple des images ou bien des paragraphes entiers dans notre cas.

On va donc modifier la fonction ***distance()*** en définissant une nouvelle distance, la distance entre les paragraphes de deux textes. L’algorithme ***distance()*** reste inchangé, ce sont les coûts des différentes opérations qui vont changer.

Del(seq) *et* ***Ins(seq)*** prennent maintenant une paragraphe en paramètre et le coût de suppression ou d’insertion de ce paragraphe correspond à la longueur de ce paragraphe.

Le coût d’une permutation donné par ***Sub(seq1, seq2)*** correspond à la distance entre les paragraphes seq1 et seq2.

Ainsi la fonction ***distance()*** est modifiée comme suit :

Données :

- Texte 1 T1
- Texte 2 T2
- Del2(seq) : retourne len(seq)
- Ins2(seq) : retourne len(seq)
- Sub2(x,y) : retourne distance(x, y)

```
distance2(T1, T2)  {
        lenT1 = longueur(T1)
        lenT2 = longueur(T2)
        T[0][0] = 0
        Pour i allant de 1 à lenT1 :
                T[i][0] = T[i-1][0] + Del2(T1[i])
        Pour j allant de 1 à lenT2 :
                T[0][j] = T[0][j-1] + Ins2(T2[j])
        Pour i allant de 1 à lenT1 :
                Pour j allant de 1 à lenT2 :
                        T[i][j] = min(
                                        T[i][j-1] + Ins2(T2[j-1]),
                                        T[i-1][j] + Del2(T1[i-1]),
                                        T[i-1][j-1] + Sub2(T1[i-1], T2[j-1])
                                )
        retourner T
}
```

La fonction ***alignement_paragraphes(T1, T2)*** permettant de faire correspondre les paragraphes de T1 et T2 est identique à ***alignements_optimaux(A, B)*** à ceci près qu'on utilise cette fois ***distance2()***.

Cette algorithme permet cette fois d'apparier les paragraphes plagiés entre deux textes.

Exemple sur les fichiers ‘t1.txt’ et ‘t2.txt’ :

Le score de similarité correspond au nombre de caractères en commun entre les deux textes, ramené au nombre total de caractères entre les deux textes.

Fonctions d’affichage

Les fonctions utiliser pour afficher les textes sont écrites en C dans le fichier *affichage.c* et sont importées par le fichier *affichage.so* dans Python à l’aide du packages *ctypes.*

Pour compiler le fichier *affichage.so*, il faut lancer la commande bash :

gcc -shared -Wl,-soname,affichage -o affichage.so -fPIC affichage.c

Instructions d’exécution

La marche à suivre pour reproduire le fichier TD3.c ainsi que l’exécutable, voir le fichier README.md ou le projet Github donné en début de rapport.

Pour exécuter le fichier compilé, lancer la commande bash :

./TD3