Rapport de TP : Alignement optimal et détection de plagiat

Annie LIM, Quentin GARRIDO

6 janvier 2020

Table des matières

1 Introduction **2**

2 Exercice 1 **2**

3 Exercice 2 **4**

4 Exercice 3 **4**

5 Exercice 4 **5**
5.1 Théorie 5
5.1.1 Algorithme 5
5.1.2 Complexité 6
5.2 Implémentation 7

6 Annexe : Code source **8**

1 Introduction

Ce TP a pour but de concevoir un logiciel d'aide à la détection de plagiat à l'aide du calcul d'un alignement optimal entre deux textes.
Ce logiciel d'alignement de séquences affichera simultanément le texte que l'on pense être du plagiat avec le texte original, en mettant en avant les correspondances. Moins les textes diffèrent et plus les chances de détecter un plagiat sont grandes.
De la même manière moins nous devons faire de changement pour aligner les textes, plus les chances de plagiat sont grandes.

Tout le code est disponible à l'adresse suivante : github/garridoq/alignement-texte Le code est aussi fourni en annexe.

Pour le compiler il suffit d'effectuer la commande *gcc TD2.c -o td2* puis pour lancer l'éxécutable il faut faire la commande *./td2*. Une démonstration du fonctionnement pour les exercies 3 et 4 devrait alors se réaliser.

Le code a été testé uniquement sous Linux et compilé avec GCC 8.
Il ne devrait pas y avoir de problèmes de compatibilité, mais dans le cas où il y en aurait merci de nous le signaler pour que nous puissions le corriger/vous prouver le bon fonctionnement d'une autre manière.

2 Exercice 1

Considérons les chaînes x et y ainsi que un de leurs étirements respectifs x' et y'. Nous voulons en plus que $|x'| = |y'|$ afin que ce soient deux alignements.
Nous voulons trouver une méthode pour calculer le score d'un alignement optimal (un alignement de plus faible score) noté $d(x', y')$.

Un premier constat que nous pouvons faire est que nous avons besoin des deux opérations suivante pour réaliser un alignement :
— Insérer un caractère '␣' (non présent dans notre alphabet) dans x à un indice j
— Insérer un caractère '␣' (non présent dans notre alphabet) dans y à un indice i
Le caractère '␣'(que nous appelerons caractère blanc par la suite) n'étant pas présent dans notre alphabet A, l'insérer dans une chaîne ou l'autre va augmenter le coût de l'alignement de 1.

La troisième action que nous pouvons réaliser est tout simplement de ne pas modifier notre texte. Cela ajoutera ainsi un coût de 1 à notre alignement si les caractères de nos deux textes sont identiques, et un coût de 1 sinon.

Visuellement, si nous remplaçons tous les blancs de x' par le caractère au même endroit dans y', que nous enlevons dans les deux textes les caractères aux indices où y' contient un caractère blanc et que autres indices nous remplaçons $x'[i]$ par $y'[i]$ si ces caractères sont différents, nous avons trouvé un moyen de transformer x en y à partir d'un alignement de ces deux textes.
De plus le coût de cette transformation de x en y est le même que le score de l'alignement x', y'. Nous pouvons en déduire alors les relations suivantes pour la transformation :

Mettre un blanc dans x à l'indice $i \Rightarrow$ Insérer $y[i]$ à l'indice i dans x
Mettre un blanc dans y à l'indice $i \Rightarrow$ Supprimer le caractère à l'indice i dans x
Ne rien faire à l'indice $i \Rightarrow$ Substituer $x[i]$ par $y[i]$

Maintenant que nous avons vu que depuis un alignement entre x et y nous pouvons obtenir une façon de transformer x en y, prouvons aussi l'inverse.

Nous considérons que nous utilisons ici la distance de Levenshtein pour calculer la distance entre x et y.

Pour rappel la distance de Levenshtein entre x et y est défini comme suit :

$$Lev(x.a, y.b) = min \begin{cases} Lev(x.a, y) + Ins(b) \\ Lev(x, y.b) + Del(a) \\ Lev(x, y) + Sub(a, b) \end{cases}$$

Avec a et b deux caractères et '.' désignant la concaténation.

Si nous avons une suite d'opérations pour transformer x en y, il nous suffit de les changer de la manière suivante pour obtenir un alignement :

Insérer $y[i]$ à l'indice i dans x $\Rightarrow$ Mettre un blanc dans x à l'indice i
Supprimer le caractère à l'indice i dans x $\Rightarrow$ Mettre un blanc dans y à l'indice i
Substituer $x[i]$ par $y[i]$ $\Rightarrow$ Ne rien faire à l'indice i

Si le coût de l'insertion est de 1, de la suppression 1 et de la substitution 1 si les caractères sont différents et 0 sinon, alors le coût de la transformation de x en y ($Lev(x, y)$) est le même que celui de l'alignement correspondant.

Nous avons ainsi une équivalence entre le problème de transformation d'une chaîne en une autre(de distance entre deux chaînes) et celui de trouver un alignement entre ces deux chaînes. Qui plus est, dans les deux cas nous avons trouvé une manière d'obtenir le même score/coût pour le même problème.
Ainsi nous avons démontré que :

Trouver un alignement optimal entre x et y $\Leftrightarrow$ Trouver la distance minimale entre x et y au sens de la distance de Levenshtein

Nous connaissons déja un algorithme pour calculer la distance de Levenshtein entre deux chaînes x et y qui est de complexité $O(|x| \times |y|)$.
Cela nous donne alors l'algorithme suivant pour calculer le côut d'un alignement optimal entre x et y.

Algorithm 1 Calcul du coût d'un alignement optimal

procedure COMPUTE_DISTANCE(x, y)
  $T[0][0] \leftarrow 0$
  for $i \leftarrow 1$ **to**$|y|$ **do**
    $T[i][0] \leftarrow T[i-1][0] + Ins(y[i-1])$
  end for
  for $j \leftarrow 1$ **to**$|x|$ **do**
    $T[0][j] \leftarrow T[0][j-1] + Del(x[j-1])$
  end for
  for $i \leftarrow 1$ **to**$|y|$ **do**
    for $j \leftarrow 1$ **to**$|x|$ **do**
      $T[i][j] \leftarrow min \begin{cases} T[i-1][j] + Ins(y[i-1]) \\ T[i][j-1] + Del(x[i-1]) \\ T[i-1][j-1] + Sub(x[j-1], y[i-1]) \end{cases}$
    end for
  end for
  return T
end procedure

Cet algorithme est bien de complexité $O(|x| \times |y|)$.

3 Exercice 2

À partir de la matric T calculée précédemment, nous allons chercher quelle opérations a été réalisée afin d'obtenir le coût minimum, puis nous allons utiliser la relation d'équivalence entre les opération d'édition du texte et celles utilisées pour réaliser un alignement.
Ces relations sont les suivantes :

Insérer $y[i]$ à l'indice i dans x $\Leftrightarrow$ Mettre un blanc dans x à l'indice i
Supprimer le caractère à l'indice i dans x $\Leftrightarrow$ Mettre un blanc dans y à l'indice i
Substituer $x[i]$ par $y[i]$ $\Leftrightarrow$ Ne rien faire à l'indice i

Nous allons alors pouvoir générer l'alignement entre nos chaînes comme suit :

Algorithm 2 Construction d'un alignement optimal

```
1:  procedure ALIGNEMENT(T, x, y)
2:      i ← |y|
3:      j ← |x|
4:      k ← 0
5:      while i > 0 or j > 0 do
6:          if i > 0 & T[i][j] = T[i − 1][j] + Ins(y[i − 1]) then
7:              x_aligned[k] ← " "
8:              y_aligned[k] ← y[i − 1]
9:              i ← i − 1
10:         else if j > 0 & T[i][j] = T[i][j − 1] + Del(x[j − 1]) then
11:             x_aligned[k] ← x[j − 1]
12:             y_aligned[k] ← " "
13:             j ← j − 1
14:         else if i > 0 & T[i][j] = T[i − 1][j − 1] + Sub(x[j − 1], y[i − 1]) then
15:             x_aligned[k] ← x[i − 1]
16:             y_aligned[k] ← y[i − 1]
17:             i ← i − 1
18:             j ← j − 1
19:         end if
20:     end while
21:     return x_aligned, y_aligned
22: end procedure
```

Les opérations Ins,Del,Sub étant en temps constant, cet algorithme est bien de complexité $O(|x|+|y|)$.

4 Exercice 3

L'implémentation est assez directe et est une simple traduction des algorithmes écrits aux exercices 1 et 2.
Le code pour le calcul du coût d'un alignement optimal se trouve des lignes 196 à 218 et celui pour la construction de cet alignement des lignes 220 à 270.

Comme nous pouvons le voir sur la figure 1 nous obtenons bien un alignement entre nos deux textes, qui est bien celui de coût minimum (d'après plusieurs tests et les résultats à l'exercice suivant).

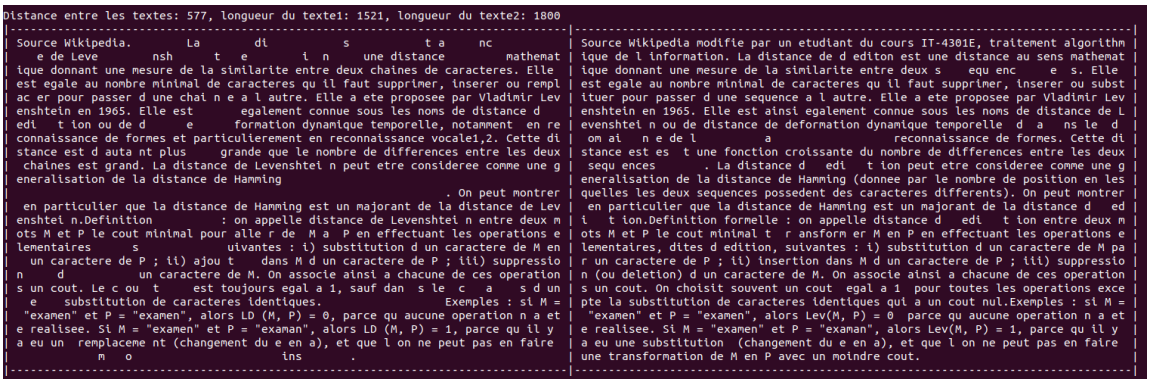

FIGURE 1 – Résultat de l'alignement des textes

5 Exercice 4

5.1 Théorie

5.1.1 Algorithme

Le principal changement ici est que nous voulons mettre en correspondance des lignes entre elles (séparées par des \n).
Précédemment nous alignions un texte composé de caractères, mais maintenant nous voulons aligner un texte composé de lignes/phrases/paragraphes qui seront nos éléments de "base".

Le problème étant très similaire au précédent, la méthode que nous utilisions devrait pouvoir être adaptée à ce nouveau problème.
Pour ce faire nous allons définir une nouvelle distance de Levenshtein agissant sur des lignes entières et plus uniquement des caractères. Nous allons définir la substitution, insertion, et suppression comme suit :

$$\text{Ins'}(y) = Lev(\epsilon, y) = |y|$$
$$\text{Del'}(x) = Lev(x, \epsilon) = |x|$$
$$\text{Sub'}(x, y) = Lev(x, y)$$

Ici Lev(x,y) est la distance de levenshtein définie précédemment, et x et y sont des lignes.

Il est assez facile de voir pourquoi nous avons choisi comme coût d'insertion et de suppression la longueur du paragraphe. En effet cela correspond à ajouter (resp. enlever) les caractères un par un, avec un coût de 1 à chaque fois.

Pour la substitution il est un peu moins clair au premier abord quelle valeur choisir. Le choix le plus simple est de supprimer puis d'insérer les paragraphes, cependant ce ne serait pas une distance car dans ce cas là $Sub(x, x) \neq 0$.
Nous avons étudié plusieurs distances entre les textes, chacunes avec leur défauts et avantages, mais celle qui paraît la meilleure est la distance de Levenshtein, qui nous donnera une meilleure indication de la différence entre nos paragraphes, et nous permettra ensuite facilement de créer un alignement ayant du sens. De plus nous avons prouvé sont équivalence avec le problème d'alignement qui nous intéresse, c'est donc la distance la plus appropriée.
Puisque nous avons considéré un coût d'ajout et de suppresion d'un caractère de 1 pour définir Del' et Ins' nous devons faire pareil dans la distance de Levenshtein Lev utilisée pour calculer Sub', et nous

considérerons un coût de substitution de 1 si les caractères dont différents et 0 sinon.

Nous pouvons alors définir notre nouvelle distance de Levenshtein comme suit :

$$Lev'(x.a, y.b) = min \begin{cases} Lev'(x.a, y) + Ins'(b) \\ Lev'(x, y.b) + Del'(a) \\ Lev'(x, y) + Sub'(a, b) \end{cases} = min \begin{cases} Lev'(x.a, y) + |b| \\ Lev'(x, y.b) + |a| \\ Lev'(x, y) + Lev(a, b) \end{cases}$$

Ici a et b ne sont plus des caractères mais sont désormais des paragraphes.
Nous sommes donc en mesure d'adapter le code précédemment écrit pour cette nouvelle version, sans faire beaucoup de changements.

Nous pouvons nous demander si Lev' est toujours une distance.
Étant donné que $Lev($ $\iff$ Sub' ici) est une distance et que $Del'(x) = Ins'(x)$ nous pouvons conclure que nous avons bien une distance, d'après une propriété vue en cours.

Nous sommes toujours une distance de Levenshtein/une edit distance, donc nous sommes toujours en train de résoudre un problème équivalent à celui de l'alignement optimal.

Pour l'algorithme de construction de l'alignement optimal, il est identique à celui vu précédemment, en adaptant juste la reconstruction des paragraphes.
Si nous avons choisi Ins' ou Del' nous construisons une chaine de caractère vide.
Si nous avons choisi Sub' nous construisons l'alignement optimal entre nos deux paragraphes comme nous l'avons fait précédemment.

5.1.2 Complexité

L'algorithme va remplir un tableau de taille $(n+1) \times (m+1)$ (n et m étant le nombre de paragraphes des textes que nous souhaitons aligner).
Il faut donc trouver la complexité de Lev' afin de conclure sur la complexité totale.
Les complexités des différentes opérations sont :

- $Lev'(x.a, y) + |b|$ a une complexité en $O(|b|)$ dans le pire des cas (comme *strlen* en C)
- $Lev'(x, y.b) + |a|$ a une complexité en $O(|a|)$ dans le pire des cas (comme *strlen* en C)
- $Lev'(x, y) + Lev(a, b)$ a une complexité en $O(|a| \times |b|)$

Si nous notons l la longueur du plus long paragraphe, la complexité de calcul de Lev' est alors $O(l^2 + l + l) = O(l^2)$.
La complexité totale du calcul de la table est donc $O(m \times n \times l^2)$.

Nous avons vu la complexité pour calculer la table des ditances, regardons maintenant pour la construction de l' alignement optimal.
Pour aligner deux paragraphes, si nous avons gardé en mémoire les tables des distances de Levenshtein utilisées pour le calcul de Sub' lors de la construction de la table de Lev' la complexité sera de $O(l)$.
Si en revanche nous ne les avons pas gardé en mémoire (comme dans notre implémentation) la complexité sera alors de $O(l^2)$ pour reconstruire la table puis $O(l)$ pour construire l' alignement dans le cas où l' opération choisie est la substitution. Dans ce cas, la complexité de la construction de l'alignement de deux paragraphes est $O(l^2)$ si nous avons choisi la substitution et $O(l)$ sinon. Il serait alors intéressant en cas d'égalité de choisir l'opération la moins coûteuse. Cela correspond à tester en premier l'égalité avec les Insertion et Suppresion, comme dans notre implémentation.

Nous obtenons alors une complexité totale pour la construction de l'alignement des textes de $O((m+n) \times l^2)$ si nous n'avons pas gardé les tables en mémoire et et de $O((m+n) \times l)$ si nous l'avons fait.

5.2 Implémentation

L'implémentation pour le calcul de la table est presque identique à celle de l'exercice 3, nous avons juste changé le calcul de sub, ins et del.
Le code est disponible en annexe des lignes 324 à 372.
Afin de séparer les différents paragraphes des textes, nous avons implémenté une fonction utilisant *strtok_r* qui devrait donc être compatible sur tous les OS, qu'ils respectent les normes POSIX ou non.
Son implémentation est fournie en annexe aux lignes 307 à 322.

Pour le backtracking aussi le principe reste le même.
Pour l'alignement si nous avons fait une insertion ou suppression nous allons implement insérer une chaîne remplie du caractère vide de bonne longueur.
Si nous avons fait une substitution nous allons alors insérer l'alignement entre les deux textes substitués.
Le code est disponible en annexe des lignes 372 à 438.

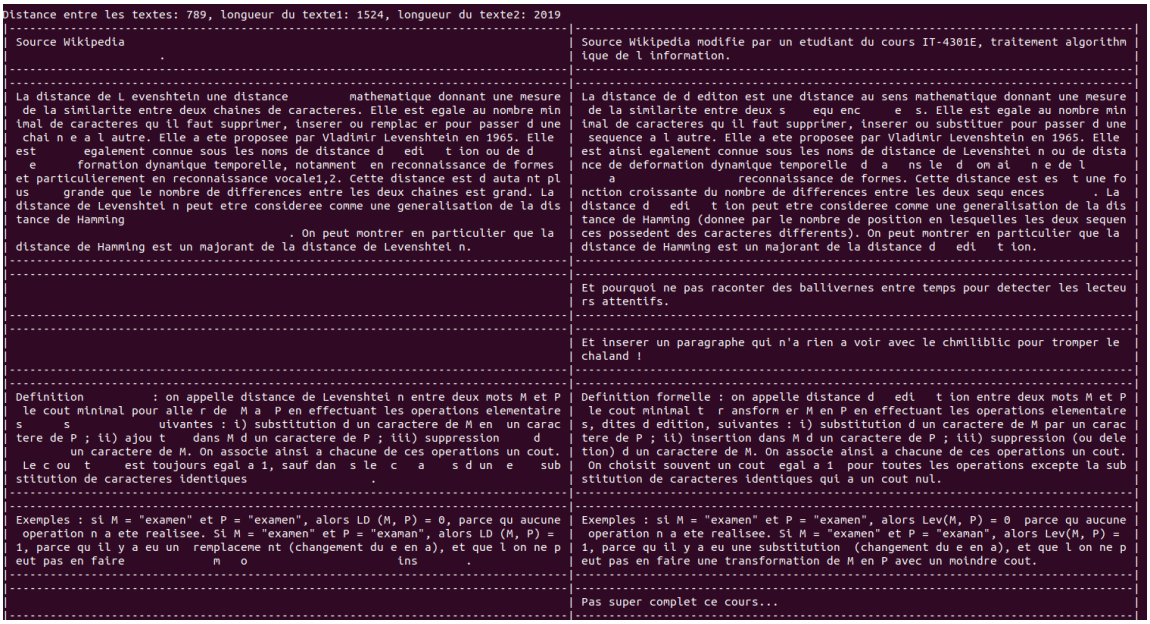

FIGURE 2 – Résultat de l'alignement des textes paragraphe par paragraphe

Comme nous pouvons le voir sur la figure 2, nous obtenons bien un bon alignement des deux textes, identique au résultat attendu, à quelques détails d'affichage près.

D'après les tests et vérifications que nous avons effectuées avec Valgrind, le programme ne comporte aucune fuite mémoire.

6 Annexe : Code source

```c
#include<stdio.h>
#include<stdlib.h>
#include <sys/stat.h>
#include <string.h>

struct alignement
{
  char * x;
  char * y;
};


/* ================================================================== */
char * readtextfile(char * filename)
  /* Retourne le contenu du fichier texte filename */
  /* ================================================================== */
{
  struct stat monstat;
  int N;
  char * text = NULL;
  FILE *fd = NULL;

  N = stat(filename, &monstat);
  if (N == -1)
  {
    fprintf(stderr, "error : bad file %s\n", filename);
    exit(0);
  }
  N = monstat.st_size;
  text = (char *)malloc(N+1);
  if (text == NULL)
  {   fprintf(stderr,"readtextfile() : malloc failed for text\n");
    exit(0);
  }
  fd = fopen(filename,"r");
  if (!fd)
  {
    fprintf(stderr, "readtextfile: can't open file %s\n", filename);
    exit(0);
  }

  fread(text, sizeof(char), N, fd);
  if((N>0) && (text[N-1] == '\n') ) text[N-1] = '\0';
  else text[N-1] = '\0';
  fclose(fd);
  return text;
}

/* ================================================================== */
int Imax(int a, int b)
  /* Retourne  le maximum de a et b                                   */
  /* ================================================================== */
{
  if (a < b) return b;
  else return a;
}

/* ================================================================== */
int Imin2(int a, int b)
  /* Retourne  le minimum de a et b                                   */
  /* ================================================================== */
{
  if (a < b) return a;
  else return b;
```

```
65 }
66
67 /* ================================================================== */
68 int Imin3(int a, int b, int c)
69   /* Retourne  le minimum de a, b et c                                */
70   /* ================================================================ */
71 {
72   return Imin2(Imin2(a,b),c);
73 }
74
75 /* ================================================================== */
76 void retourne(char *c)
77   /* Retourner la chaine de caractere c                               */
78   /* ================================================================ */
79 {
80   char tmp;
81   int m, j, i;
82   m = strlen(c);
83   j = m/2;
84   for(i = 0; i < j; i++ ){
85     tmp = c[i];
86     c[i] = c[m-i-1];
87     c[m-i-1] = tmp;
88   }
89 }
90 /* ================================================================== */
91 void afficheSeparateurHorizontal(int nbcar)
92   /* ================================================================ */
93 {
94   int i;
95   printf("|-");
96   for(i=0; i < nbcar; i++)
97     printf("-");
98   printf("-|-");
99   for(i=0; i < nbcar; i++)
100    printf("-");
101  printf("-|\n");
102 }
103
104
105 /* ================================================================== */
106 void affiche(char* texte1, char* texte2, int nbcar)
107   /* Affiche simultanement texte1 et texte 2 en positionnnant nbcar
108      caracteres sur chaque ligne. */
109   /* ================================================================ */
110 {
111   int i, l1, l2, l;
112
113   char *t1,*t2;
114
115   char out[512];
116
117   l1 = strlen(texte1);
118   l2 = strlen(texte2);
119
120   t1 = (char*) malloc(sizeof(char) * (nbcar + 1));
121   t2 = (char*)malloc(sizeof(char) * (nbcar + 1));
122
123   l = Imax(l1, l2);
124   afficheSeparateurHorizontal(nbcar);
125   for(i = 0; i < l; i+= nbcar){
126     if (i < l1) {
127       strncpy(t1, &(texte1[i]), nbcar);
128       t1[nbcar] = '\0';
129     } else t1[0] = '\0';
130     if (i < l2) {
```

```
131       strncpy(t2, &(texte2[i]),nbcar);
132       t2[nbcar] = '\0';
133     } else t2[0] = '\0';
134
135     sprintf(out, "| %c-%ds | %c-%ds |\n",'%', nbcar, '%', nbcar);
136     printf(out, t1,t2);
137   }
138   afficheSeparateurHorizontal(nbcar);
139   free(t1);
140   free(t2);
141 }
142
143
144
145 /* ================================================================== */
146 /* idem affiche, mais avec un formattage different*/
147 /* ================================================================== */
148 void affiche2(char* texte1, char* texte2, int nbcar)
149 {
150
151   int i, l1, l2, l;
152
153   char *t1,*t2;
154
155   char out[512];
156
157   l1 = strlen(texte1);
158   l2 = strlen(texte2);
159
160   t1 = (char*) malloc(sizeof(char) * (nbcar + 1));
161   t2 = (char*)malloc(sizeof(char) * (nbcar + 1));
162
163   l = Imax(l1, l2);
164
165   for(i = 0; i < l; i+= nbcar){
166     if (i < l1) {
167       strncpy(t1, &(texte1[i]), nbcar);
168       t1[nbcar] = '\0';
169     } else t1[0] = '\0';
170     if (i < l2) {
171       strncpy(t2, &(texte2[i]),nbcar);
172       t2[nbcar] = '\0';
173     } else t2[0] = '\0';
174
175     sprintf(out, "x: %c-%ds \ny: %c-%ds\n",'%', nbcar, '%', nbcar);
176     printf(out, t1,t2);
177
178   }
179   free(t1);
180   free(t2);
181 }
182
183
184 /*================================================================== */
185 /*              Exercice 3                                          */
186 /*================================================================== */
187
188
189 int sub(char a, char b){
190   if(a == b)
191     return 0;
192   return 1;
193 }
194
195
196 int** compute_distance(char* texte1, char* texte2){
```

```
197   int n = strlen(texte1);
198   int m = strlen(texte2);
199
200   int** T= (int**) malloc((m+1)*sizeof(int*));
201   for(int i=0; i<=m; i++)
202     T[i]= (int*) malloc((n+1)*sizeof(int));
203
204   //T[m+1][n+1]
205   T[0][0] = 0;
206   for(int i=1; i<n+1;++i)
207     T[0][i] = T[0][i-1] + 1; //Cout del
208   for(int j=1; j<m+1;++j)
209     T[j][0] = T[j-1][0] + 1; //Cout ins
210   for(int i=1; i<n+1;++i)
211     for(int j=1; j<m+1;++j){
212       T[j][i] = Imin3(T[j-1][i]+1,\
213           T[j][i-1]+1 ,\
214           T[j-1][i-1]+sub(texte1[i-1],texte2[j-1]));
215     }
216
217   return T;
218 }
219
220 char** get_alignement(int** T, char* texte1, char* texte2, int verbose){
221   char blank = ' ';
222   int n = strlen(texte1);
223   int m = strlen(texte2);
224   int i = m;
225   int j = n;
226
227   int len = m+n;
228   char *texte1_aligned = (char *)malloc((len+1)*sizeof(char));
229   char *texte2_aligned = (char *)malloc((len+1)*sizeof(char));
230
231   int k = 0;
232   while(i !=0 || j != 0){
233     //Si on a choisi Ins
234     if( i>0 && (T[i][j] == T[i-1][j] + 1)){
235       if(verbose) printf("Ins %c dans texte 1\n",texte2[i-1]);
236       texte1_aligned[k] = blank;
237       texte2_aligned[k] = texte2[i-1];
238       i--;
239     }
240     //Si on a choisi Del
241     else if( j>0 && (T[i][j] == T[i][j-1] + 1)){
242       if(verbose) printf("Del %c dans texte 1\n",texte1[j-1]);
243       texte1_aligned[k] = texte1[j-1];
244       texte2_aligned[k] = blank;
245       j--;
246     }
247     //Si on a choisi Sub
248     else if((T[i][j] == T[i-1][j-1] + sub(texte1[j-1],texte2[i-1]))){
249       if(verbose) printf("Remplacer %c par %c dans le texte 1\n",texte1[j-1],texte2[j
      -1]);
250       texte1_aligned[k] = texte1[j-1];
251       texte2_aligned[k] = texte2[i-1];
252       j--;
253       i--;
254     }
255     else{
256       printf("Erreur dans le calcul de la table\n");
257       break;
258     }
259     k++;
260   }
261   texte1_aligned[k] ='\0';
```

```
262   texte2_aligned[k] ='\0';
263
264   char **textes = (char**)malloc(2*sizeof(char*));
265   retourne(texte1_aligned);
266   retourne(texte2_aligned);
267   textes[0] = texte1_aligned;
268   textes[1] = texte2_aligned;
269   return textes;
270 }
271
272 void align_sentence(char* texte1, char*texte2){
273
274   int** T = compute_distance(texte1, texte2);
275   printf("Distance entre les textes: %d, longueur du texte1: %ld, longueur du texte2: %
        ld\n",\
276       T[strlen(texte2)][strlen(texte1)], strlen(texte1),strlen(texte2));
277
278   char** result = get_alignement(T, texte1, texte2, 0);
279   affiche(result[0],result[1],80);
280
281   //Free results
282   free(result[0]);
283   free(result[1]);
284   free(result);
285
286   //Free T
287   for(int i=0;i<=strlen(texte2);++i)
288     free(T[i]);
289   free(T);
290 }
291
292
293 /*============================================================== */
294 /*                Exercice 4                                    */
295 /*============================================================== */
296
297
298 int count_occurences(char* texte, const char sep){
299   int count = 0;
300   for(int i = 0; i < strlen(texte);++i){
301     if(texte[i] == sep)
302       count++;
303   }
304   return count;
305 }
306
307 char** split(char* texte,int count, const char* sep){
308   if(count == 0)
309     return NULL;
310
311   char* texte_cop = strdup(texte);
312   char** liste = (char**)malloc((count+1)*sizeof(char*));
313   char* reste = NULL;
314   char* token;
315   int i = 0;
316   for(token = strtok_r(texte_cop,sep,&reste); token !=NULL; token=strtok_r(NULL,sep,&
        reste))
317   {
318     liste[i] = strdup(token);
319     i++;
320   }
321   return liste;
322 }
323
324 int sub_strings(char* texte1, char* texte2){
325   int** T = compute_distance(texte1,texte2);
```

```
326   int val = T[strlen(texte2)][strlen(texte1)];
327
328   for(int i = 0; i <= strlen(texte2);++i)
329     free(T[i]);
330   free(T);
331   return val;
332 }
333
334 int ins_strings(char* texte2){
335   return strlen(texte2);
336 }
337 int del_strings(char* texte1){
338   return strlen(texte1);
339 }
340
341 char* blank_string(char blank, int n){
342   char* string = (char*)malloc((n+1)*sizeof(char));
343   memset(string,blank,n);
344   string[n] = '\0';
345   return string;
346 }
347
348
349 int** compute_distance_strings(char** texte1,int n1, char** texte2,int n2){
350   int n = n1+1;
351   int m = n2+1;
352
353   int** T= (int**) malloc((m+1)*sizeof(int*));
354   for(int i=0; i<=m; i++)
355     T[i]= (int*) malloc((n+1)*sizeof(int));
356
357   //T[m+1][n+1]
358   T[0][0] = 0;
359   for(int i=1; i<n+1;++i)
360     T[0][i] = T[0][i-1] + del_strings(texte1[i-1]); //Cout del
361   for(int j=1; j<m+1;++j)
362     T[j][0] = T[j-1][0] + ins_strings(texte2[j-1]); //Cout ins
363   for(int i=1; i<n+1;++i)
364     for(int j=1; j<m+1;++j){
365       T[j][i] = Imin3(T[j-1][i]+ins_strings(texte2[j-1]),\
366           T[j][i-1]+del_strings(texte1[i-1])   ,\
367           T[j-1][i-1]+sub_strings(texte1[i-1],texte2[j-1]));
368     }
369
370
371   return T;
372 }
373
374 char*** get_alignement_texts(int** T, char** texte1,int n1, char** texte2,int n2, int
      verbose, int* count){
375   char blank = ' ';
376   int n = n1+1;
377   int m = n2+1;
378   int i = m;
379   int j = n;
380
381   int len = m+n;
382   //int len = Imax(m,n);
383   char **texte1_aligned = (char **)malloc((len)*sizeof(char*));
384   char **texte2_aligned = (char **)malloc((len)*sizeof(char*));
385
386   int k = 0;
387   while(i !=0 || j != 0){
388     //Si on a choisi Ins
389     if( i>0 && (T[i][j] == T[i-1][j] + ins_strings(texte2[i-1]))){
390       if(verbose) printf("Ins\n");
```

```
391         texte1_aligned[k] = blank_string(blank,strlen(texte2[i-1]));
392         texte2_aligned[k] = strdup(texte2[i-1]);
393         i--;
394       }
395       //Si on a choisi Del
396       else if( j>0 && (T[i][j] == T[i][j-1] + del_strings(texte1[j-1]))){
397         if(verbose) printf("Del\n");
398         texte1_aligned[k] = strdup(texte1[j-1]);
399         texte2_aligned[k] = blank_string(blank, strlen(texte1[j-1]));
400         j--;
401       }
402       //Si on a choisi Sub
403       else if((T[i][j] == T[i-1][j-1] + sub_strings(texte1[j-1],texte2[i-1]))){
404         if(verbose) printf("Sub\n");
405
406         int** T_temp = compute_distance(texte1[j-1],texte2[i-1]);
407
408         char** alignes = get_alignement(T_temp, texte1[j-1], texte2[i-1],0);
409
410         texte1_aligned[k] = alignes[0];
411         texte2_aligned[k] = alignes[1];
412
413         //Free T
414         int size =strlen(texte2[i-1]);
415         for(int i=0; i<=size;++i){
416           free(T_temp[i]);
417         }
418         free(T_temp);
419         //Free alignes
420         free(alignes);
421         j--;
422         i--;
423       }
424       else{
425         printf("Erreur dans le calcul de la table\n");
426         break;
427       }
428       k++;
429     }
430
431     *count = k;
432
433
434     char ***textes = (char***)malloc(2*sizeof(char**));
435     textes[0] = texte1_aligned;
436     textes[1] = texte2_aligned;
437     return textes;
438 }
439
440 void align_texts(char* texte1, char* texte2){
441     int n1 = count_occurences(texte1,'\n');
442     char** liste1 = split(texte1,n1,"\n");
443     int n2 = count_occurences(texte2,'\n');
444     char** liste2 = split(texte2,n2,"\n");
445
446     int** T = compute_distance_strings(liste1, n1, liste2, n2);
447     printf("Distance entre les textes: %d, longueur du texte1: %ld, longueur du texte2: %
        ld\n",\
448         T[n2+1][n1+1], strlen(texte1),strlen(texte2));
449     int count;
450     char*** result = get_alignement_texts(T, liste1, n1, liste2, n2, 0, &count);
451     for(int i = count-1; i>=0;--i){
452       affiche(result[0][i],result[1][i],80);
453     }
454
455     //Free results
```

```
456   for(int i=0;i<count;++i){
457     free(result[0][i]);
458     free(result[1][i]);
459   }
460   free(result[0]);
461   free(result[1]);
462   free(result);
463 
464   //Free T
465   for(int i=0;i<=n2+1;++i)
466     free(T[i]);
467   free(T);
468 
469   //Free liste1 et liste2
470   for(int i=0;i<=n1;++i){
471     free(liste1[i]);
472   }
473   free(liste1);
474 
475   for(int i=0;i<=n2;++i){
476     free(liste2[i]);
477   }
478   free(liste2);
479 
480 }
481 
482 /* ============================================================== */
483 int main(int argc, char **argv)
484   /* ============================================================== */
485 {
486   char *texte1, *texte2;
487 
488   printf("================================================================================\n");
489   printf("                              Exercice 3                                        \n");
490   printf("================================================================================\n");
491 
492   texte1 = readtextfile("texte1.txt");
493   texte2 = readtextfile("texte2.txt");
494 
495   align_sentence(texte1, texte2);
496 
497   free(texte1);
498   free(texte2);
499 
500   printf("================================================================================\n");
501   printf("                              Exercice 4                                        \n");
502   printf("================================================================================\n");
503 
504   texte1 = readtextfile("t1.txt");
505   texte2 = readtextfile("t2.txt");
506 
507   align_texts(texte1, texte2);
508 
509   free(texte1);
510   free(texte2);
511 
512 }
```