Rapport de TP : Alignement optimal et détection de plagiat

Annie LIM, Quentin GARRIDO

6 janvier 2020

Table des matières

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

Pour calculer le score d'un alignement optimal entre x et y, nous pouvons utiliser l'algorithme de distance de Levenshtein, appelé distance d'édition (edit distance).
Nous voulons observer les différences entre deux textes. Cela revient à calculer leur score d'alignement, le coût des opérations nécessaires (deletion, insertion, substitution) pour obtenir le même texte. Plus ce score est faible et plus les textes sont similaires, et donc sujet d'être un plagiat.
Le score optimal correspond au minimum entre les trois valeurs données par les opérations deletion, insertion et substitution.
Cet algorithme est bien de complexité $O(|x| \times |y|)$.

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

3 Exercice 2

Soit une matrice T telle que $T[i][j]$ est le score d'un alignement optimal entre x_i et y_j avec x_i ry y_j les préfixes de x et y de longueur i et j.
A partir de cette matrice, nous pourrons retrouver les opérations nécessaires à la solution optimale pour aligner les deux textes, afin de construire les textes 1 et 2 modifiés alignés.
Le backtracking consiste à suivre le chemin minimum de la matrice T de $T[i][j]$ jusqu'à $T[0][0]$.

Algorithm 2 Construction d'un alignement optimal

procedure ALIGNEMENT(T, x, y)
  $i \leftarrow |y|$
  $j \leftarrow |x|$
  $k \leftarrow 0$
  while $i > 0$ **or** $j > 0$ **do**
    if $i > 0$ & $T[i][j] = T[i-1][j] + Ins(y[i-1])$ **then**
      $x_{aligned}[k] \leftarrow$ " "
      $y_{aligned}[k] \leftarrow y[i-1]$
      $i \leftarrow i - 1$
    else if $j > 0$ & $T[i][j] = T[i][j-1] + Del(x[i-1])$ **then**
      $x_{aligned}[k] \leftarrow x[i-1]$
      $y_{aligned}[k] \leftarrow$ " "
      $j \leftarrow j - 1$
    else if $i > 0$ & $T[i][j] = T[i-1][j-1] + Sub(x[j-1], y[i-1])$ **then**
      $x_{aligned}[k] \leftarrow x[i-1]$
      $y_{aligned}[k] \leftarrow y[i-1]$
      $i \leftarrow i - 1$
      $j \leftarrow j - 1$
    end if
  end while
  return $x_{aligned}, y_{aligned}$
end procedure

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

4 Exercice 3

```
Distance entre les textes: 577, longueur du texte1: 1521, longueur du texte2: 1800
|-------------------------------------------------------------------------------|-------------------------------------------------------------------------------|
| Source Wikipedia.       La       di          s          t a     nc           | Source Wikipedia modifie par un etudiant du cours IT-4301E, traitement algorithm |
|    e de Leve        nsh      t   e        i   n     une distance         mathemat | ique de l information. La distance de d editon est une distance au sens mathemat |
| ique donnant une mesure de la similarite entre deux chaines de caracteres. Elle  | ique donnant une mesure de la similarite entre deux s    equ enc     e  s. Elle  |
| est egale au nombre minimal de caracteres qu il faut supprimer, inserer ou rempl | est egale au nombre minimal de caracteres qu il faut supprimer, inserer ou subst |
| ac er pour passer d une chai n e a l autre. Elle a ete proposee par Vladimir Lev | ituer pour passer d une sequence a l autre. Elle a ete proposee par Vladimir Lev |
| enshtein en 1965. Elle est       egalement connue sous les noms de distance d    | enshtein en 1965. Elle est ainsi egalement connue sous les noms de distance de L |
| edi   t ion ou de d      e      formation dynamique temporelle, notamment  en re | evenshtei n ou de distance de deformation dynamique temporelle  d  a   ns le  d  |
| connaissance de formes et particulierement en reconnaissance vocale1,2. Cette di | om ai    n e de l          a                  reconnaissance de formes. Cette di |
| stance est d auta nt plus     grande que le nombre de differences entre les deux | stance est es   t une fonction croissante du nombre de differences entre les deux |
|  chaines est grand. La distance de Levenshtei n peut etre consideree comme une g |  sequ ences       . La distance d   edi   t ion peut etre consideree comme une g |
| eneralisation de la distance de Hamming                                       | eneralisation de la distance de Hamming (donnee par le nombre de position en les |
|                                                        . On peut montrer      | quelles les deux sequences possedent des caracteres differents). On peut montrer |
|  en particulier que la distance de Hamming est un majorant de la distance de Lev |  en particulier que la distance de Hamming est un majorant de la distance d    ed |
| enshtei n.Definition          : on appelle distance de Levenshtei n entre deux m | i   t ion.Definition formelle : on appelle distance d   edi   t ion entre deux m |
| ots M et P le cout minimal pour alle r de  M a   P en effectuant les operations e | ots M et P le cout minimal t   r ansform er M en P en effectuant les operations e |
| lementaires      s          uivantes : i) substitution d un caractere de M en | lementaires, dites d edition, suivantes : i) substitution d un caractere de M pa |
|   un caractere de P ; ii) ajou t    dans M d un caractere de P ; iii) suppressio | r un caractere de P ; ii) insertion dans M d un caractere de P ; iii) suppressio |
| n     d          un caractere de M. On associe ainsi a chacune de ces operation | n (ou deletion) d un caractere de M. On associe ainsi a chacune de ces operation |
| s un cout. Le c ou   t     est toujours egal a 1, sauf dan  s le   c    a    s d un | s un cout. On choisit souvent un cout  egal a 1   pour toutes les operations exce |
|   e    substitution de caracteres identiques.                 Exemples : si M = | pte la substitution de caracteres identiques qui a un cout nul.Exemples : si M = |
| "examen" et P = "examen", alors LD (M, P) = 0, parce qu aucune operation n a et | "examen" et P = "examen", alors Lev(M, P) = 0  parce qu aucune operation n a et |
| e realisee. Si M = "examen" et P = "examan", alors LD (M, P) = 1, parce qu il y | e realisee. Si M = "examen" et P = "examan", alors Lev(M, P) = 1, parce qu il y |
| a eu un  remplaceme nt (changement du e en a), et que l on ne peut pas en faire | a eu une substitution  (changement du e en a), et que l on ne peut pas en faire |
|          m   o                  ins       .                                  | une transformation de M en P avec un moindre cout.                            |
|-------------------------------------------------------------------------------|-------------------------------------------------------------------------------|
```

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 délétion comme suit :

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

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 délétion 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 sur 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.
Puisque nous avons considéré un coût d'ajout et de suppresion d'un caractère de 1 pour définir Sub' et Ins' nous devons faire pareil dans la distance de Levenshtein, et nous considérerons un coût de substitution de 1 si les caractères dont différents et 0 sinon.

Nous pouvons ensuite 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 est une distance et aue $Sub'(x) = Ins'(x)$ nous pouvons conclure que nous avons bien une distance.

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 construsions l'alignement optimal entre nos deux paragraphes comme nous l'avons fait précédemment.

5.1.2 Complexité

L'algorithme va remplir $n \times m$ cases (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.

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 373.
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 délétion 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 375 à 439.

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.

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

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

```
262   texte1_aligned[k] ='\0';
263   texte2_aligned[k] ='\0';
264
265   char **textes = (char**)malloc(2*sizeof(char*));
266   retourne(texte1_aligned);
267   retourne(texte2_aligned);
268   textes[0] = texte1_aligned;
269   textes[1] = texte2_aligned;
270   return textes;
271 }
272
273 void align_sentence(char* texte1, char*texte2){
274
275   int** T = compute_distance(texte1, texte2);
276   printf("Distance entre les textes: %d, longueur du texte1: %ld, longueur du texte2: %
        ld\n",\
277       T[strlen(texte2)][strlen(texte1)], strlen(texte1),strlen(texte2));
278
279   char** result = get_alignement(T, texte1, texte2, 0);
280   affiche(result[0],result[1],80);
281
282   //Free results
283   free(result[0]);
284   free(result[1]);
285   free(result);
286
287   //Free T
288   for(int i=0;i<=strlen(texte2);++i)
289     free(T[i]);
290   free(T);
291 }
292
293
294 /*============================================================== */
295 /*              Exercice 4                                      */
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   T[1][1] = Imin3(T[0][1]+1,T[1][0]+1, T[0][0]);
364   for(int i=1; i<n+1;++i)
365     for(int j=1; j<m+1;++j){
366       T[j][i] = Imin3(T[j-1][i]+ins_strings(texte2[j-1]),\
367           T[j][i-1]+del_strings(texte1[i-1])   ,\
368           T[j-1][i-1]+sub_strings(texte1[i-1],texte2[j-1]));
369     }
370
371
372   return T;
373 }
374
375 char*** get_alignement_texts(int** T, char** texte1,int n1, char** texte2,int n2, int
      verbose, int* count){
376   char blank = ' ';
377   int n = n1+1;
378   int m = n2+1;
379   int i = m;
380   int j = n;
381
382   int len = m+n;
383   //int len = Imax(m,n);
384   char **texte1_aligned = (char **)malloc((len)*sizeof(char*));
385   char **texte2_aligned = (char **)malloc((len)*sizeof(char*));
386
387   int k = 0;
388   while(i !=0 || j != 0){
389     //Si on a choisi Ins
390     if( i>0 && (T[i][j] == T[i-1][j] + ins_strings(texte2[i-1]))){
```

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

```
456   //Free results
457   for(int i=0;i<count;++i){
458     free(result[0][i]);
459     free(result[1][i]);
460   }
461   free(result[0]);
462   free(result[1]);
463   free(result);
464
465   //Free T
466   for(int i=0;i<=n2+1;++i)
467     free(T[i]);
468   free(T);
469
470   //Free liste1 et liste2
471   for(int i=0;i<=n1;++i){
472     free(liste1[i]);
473   }
474   free(liste1);
475
476   for(int i=0;i<=n2;++i){
477     free(liste2[i]);
478   }
479   free(liste2);
480
481 }
482
483 /* ============================================================================ */
484 int main(int argc, char **argv)
485   /* ============================================================================ */
486 {
487   char *texte1, *texte2;
488
489   printf("================================================================================\n");
490   printf("                              Exercice 3                                        \n");
491   printf("================================================================================\n");
492
493   texte1 = readtextfile("texte1.txt");
494   texte2 = readtextfile("texte2.txt");
495
496   align_sentence(texte1, texte2);
497
498   free(texte1);
499   free(texte2);
500
501   printf("================================================================================\n");
502   printf("                              Exercice 4                                        \n");
503   printf("================================================================================\n");
504
505   texte1 = readtextfile("t1.txt");
506   texte2 = readtextfile("t2.txt");
507
508   align_texts(texte1, texte2);
509
510   free(texte1);
511   free(texte2);
512
513 }
```