

Projet AAVP

Similarité de deux chaînes de caractères

LEFEVRE Clément

- IATIC4 2018-2019 -

Encadrant : M. SOHIER

Table des matières

<u>TABLE DES MATIERES</u>	2
<u>INTRODUCTION</u>	3
<u>RESUME DU TRAVAIL EFFECTUE</u>	4
<u>LA DISTANCE DE HAMMING</u>	5
1. L'ALGORITHME	5
2. IMPLEMENTATION EN C	7
3. VALIDITE – TESTS	7
4. COMPLEXITE – TEMPS DE CALCUL	12
<u>LA DISTANCE DE LEVENSHTEIN</u>	13
1. L'ALGORITHME	13
2. IMPLEMENTATION EN LANGAGE C	19
3. VALIDITE – TESTS	20
4. COMPLEXITE – TEMPS DE CALCUL	29
<u>LA DISTANCE DE DAMERAU-LEVENSHTEIN</u>	33
1. L'ALGORITHME	33
2. IMPLEMENTATION EN LANGAGE C	35
<u>LIMITES ET AXES D'AMELIORATIONS</u>	36
<u>CONCLUSION</u>	37
<u>WEBOGRAPHIE</u>	37

Introduction

Ce rapport traite du projet d'Analyse d'Algorithmes et de Validation de Programmes (AAVP), module enseigné par M. Sohier, dans le cadre de notre seconde année de cycle ingénieur informatique (IATIC 4) à l'ISTY (Institut des Sciences et Techniques des Yvelines).

Le sujet était le suivant :

Il vous est demandé d'implémenter un algorithme de votre choix et de lui appliquer des méthodes vues en cours visant à étudier sa validité ainsi que son temps de calcul. Ce travail sera fait en monômes.

Rapidement, je me suis tourné vers un problème particulier, auquel nous avons été confrontés l'an dernier, dans le cadre d'un mini-projet du module de Programmation Orientée Objet : **la comparaison de chaînes de caractères**.

En effet, au sein de ce projet, nous devions **développer un correcteur orthographique** en langage Python, en respectant le paradigme de programmation orienté-objet.

Au cours de la conception du projet, j'ai donc été amené à **étudier le fonctionnement de certains correcteurs orthographiques**, et de **leurs algorithmes**, ce qui m'a permis de me familiariser avec des notions, qui m'étaient jusqu'alors inconnues, telles que la **distance de Levenshtein** (aussi appelée **distance d'édition**).

Ainsi, afin de vérifier l'orthographe d'un mot saisi, nous **parcourions un dictionnaire** à sa recherche, et si ce **mot était introuvable**, nous **récupérions les mots proches**, à l'aide d'une **implémentation de la distance de Levenshtein**, et les proposions à l'utilisateur, via une interface graphique.

J'ai donc choisi de m'intéresser de nouveau à ce problème de comparaisons de chaînes de caractères, cette fois d'un point de vue plus axé sur l'algorithmique, en étudiant et en comparant différents algorithmes, et différentes implémentations de ceux-ci.

Au sein de ce rapport, nous allons donc étudier **trois manières** de mesurer la **similarité de deux chaînes de caractères**.

J'ai donc choisi d'introduire ce rapport avec la **distance de Hamming**, qui n'est pas un algorithme initialement prévu pour le problème de comparaison de chaînes de caractères, nous expliciterons donc ses limites face à ce problème, avant d'étudier deux autres algorithmes plus appropriés :

- La **distance de Levenshtein** (distance d'édition), sous deux formes, récursive et itérative.
- La **distance de Damerau-Levenshtein**, qui est une amélioration de la précédente.

Résumé du travail effectué

Pour chacun des trois algorithmes traités au sein de ce rapport, nous procéderons de la manière suivante :

- Nous introduirons l'algorithme, en détaillant son fonctionnement ainsi que son pseudocode.
- Nous étudierons une implémentation en langage C.
- Nous montrerons ensuite la validité de cet algorithme, notamment grâce à deux notions étudiées en cours : la spécification, l'invariant de boucle, et nous mettrons en place deux types de tests : les tests en boîte noire et les tests structurels, que nous établirons à l'aide de flowcharts.
- Enfin, nous étudierons la complexité de l'algorithme, et nous la comparerons avec le temps de calcul du programme implémenté.

Les différents tests ont été réalisés sur un Macbook Pro (mi-2012), possédant un processeur bi cœur Intel Core i5, cadencé à 2,5 GHz et 16 Go de mémoire vive (DDR3).

Les sources ont été compilées par *clang*, un compilateur pour les langages C, C++ et Objective-C.

De plus, il est important de préciser que les tests ont été réalisés avec l'ordinateur branché sur secteur, et avec le minimum de tâches en cours d'exécution.

Enfin, pour les mesures de temps d'exécutions des différents algorithmes, le résultat affiché correspond à une moyenne de 10 simulations.

Les mesures de temps d'exécutions ont d'ailleurs été réalisées à l'aide de la fonction *clock()* de la bibliothèque standard du C *time.h*, qui calcule le nombre de battements d'horloge écoulés depuis le lancement du programme.

On récupère donc cette valeur directement avant et après l'exécution de la fonction, puis on soustrait la première de la seconde, en divisant le résultat par la macro-constante *CLOCKS_PER_SEC*, afin de récupérer un temps (de type double) en secondes.

Grâce à ces précautions, les tests réalisés gagnent naturellement en fiabilité et en précision.

La distance de Hamming

1. L'algorithme

La **distance de Hamming**, qui tient son nom du mathématicien Richard Hamming, permet de quantifier la différence entre deux séquences de symboles, de même taille.

Elle a été introduite et décrite pour la première fois au sein d'un **article fondateur de la théorie des codes** : « *error-detecting and error-correcting-codes* ».

La distance de Hamming possède de **nombreuses applications**, elle est notamment utilisée pour le **traitement de signal** et les **télécommunications**.

Ici, nous l'utilisons simplement comme une **première approche de la comparaison de chaînes de caractères**.

Son algorithme est plutôt simple, il consiste à **parcourir les deux chaînes de caractères** par une boucle *Pour*, et à chaque itération :

1. On compare le caractère d'index *i* du premier mot, avec celui du second mot
2. S'ils sont différents, on incrémenté la distance de Hamming.

Algorithme : Hamming (**caractere** mot1[longueur], **caractere** mot2[longueur], **entier** longueur) : entier

Variables

dist : entier
i,j : entiers

Début

dist = 0

Pour i allant de 1 à longueur

Si mot1[i] != mot2[j] **alors**
 dist ← dist + 1

FinSi

FinPour

Renvoyer dist

Fin

Figure 1 : Algorithme de la distance de Hamming

Ainsi, pour deux mots de taille 10, A = *Algorithme* et B = *Allergique*, on a :

A	A	L	G	O	R	I	T	H	M	E
B	A	L	G	E	B	R	I	Q	U	E
Distance de Hamming	0	0	0	1	1	1	1	1	1	0

$$DistanceHamming(A, B) = 0 + 0 + 0 + 1 + 1 + 1 + 1 + 1 + 1 + 0 = 6$$

La différence de Hamming entre *Algorithme* et *Algébrique* est donc de 6, et les deux mots sont de taille 10, ils sont donc **relativement éloignés**.

La principale limite de cet algorithme est qu'il ne fonctionne réellement qu'avec des **chaînes de même taille**, ce qui le rend réellement inefficace pour des applications telles qu'un correcteur orthographique par exemple.

En effet, si on insère une lettre en plus dans le second mot, la **position de cette lettre influera très fortement sur la valeur de la distance**.

Prenons l'exemple d'une faute de frappe, pour A = *Algorithme* et B = *Alhgorithm*, on a :

A	A	L	G	O	R	I	T	H	M	E
B	A	L	G	H	O	R	I	T	H	M
Distance de Hamming	0	0	0	1	1	1	1	1	1	1

$$DistanceHamming(A, B) = 0 + 0 + 0 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 7$$

Prenons maintenant A = *Algorithme* et C = *Algorithm1*. Là encore, on se place dans l'hypothèse d'une faute de frappe, cette fois à la fin du mot.

A	A	L	G	O	R	I	T	H	M	E
B	A	L	G	O	R	I	T	H	M	L
Distance de Hamming	0	0	0	0	0	0	0	0	0	1

$$DistanceHamming(A, C) = 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 1 = 1$$

Pour les deux mots B et C, une seule lettre a été modifiée par rapport au mot A, pourtant, les distances de Hamming se révèlent fortement différentes.

En effet, la distance de Hamming se montre **peu robuste pour prendre en compte les décalages**.

2. Implémentation en C

```
int distance_hamming(char* mot1, char* mot2, unsigned int tailleMot1, unsigned int tailleMot2)
{
    unsigned int i,dist = 0;
    if(tailleMot1 != tailleMot2)
    {
        printf("Erreur, les deux tailles doivent être identiques.\n");
        exit(EXIT_FAILURE);
    }
    for(i = 0; i < tailleMot1; i++)
        if(mot1[i] != mot2[i])
            dist++;
    return dist;
}
```

Figure 2 : Implémentation de l'algorithme de la distance de Hamming en langage C

L'implémentation de cet algorithme se révèle assez simple et fidèle au pseudocode. Afin que l'algorithme soit fonctionnel, il est nécessaire d'avoir des mots de même longueur, c'est pourquoi, avant de l'exécuter, on émet cette condition. Si elle n'est pas respectée, alors on quitte le programme, sinon, on poursuit l'exécution.

On remarque aussi quelques précisions concernant les types utilisés. Puisque l'on manipule des longueurs, il peut être judicieux de manipuler des entiers non-signés.

3. Validité – Tests

A. Spécification et invariant de boucle

Spécification :

Si on considère deux mots $A = \{A_1, A_2, \dots, A_n\}$ et $B = \{B_1, B_2, \dots, B_n\}$ de taille n , la distance de Hamming entre A et B, peut être considérée comme étant **le nombre de positions auxquelles A et B sont différents**.

Cette distance **est supérieure ou égale à 0** (elle vaut 0 si les deux mots sont identiques) et est également **inférieure ou égale n** (elle vaut n si quelle que soit la position, les caractères de A et B sont différents).

On peut donc exprimer cela avec la spécification suivante :

$$(d \geq 0) \wedge (d \leq n) \wedge (d = |\{i \in \{1, \dots, n\}, A_i \neq B_i\}|)$$

En effet, cette spécification revient à dire que la distance est bornée par 0 et par n , et qu'elle est égale au cardinal de l'ensemble pour lequel $A_i \neq B_i, i \in \{1, \dots, n\}$

Invariant de boucle :

Un invariant de boucle est une propriété qui, si elle est vérifiée avant l'exécution d'une itération de boucle, le demeure après l'exécution de l'itération.

Ainsi, si on considère deux mots $A = \{A_1, A_2, \dots, A_n\}$ et $B = \{B_1, B_2, \dots, B_n\}$ de taille n , la propriété suivante est un invariant de boucle :

$$(d \geq 0) \wedge (d \leq i) \wedge (d = |\{j \in \{1, \dots, i\}, A_j \neq B_j\}|) \wedge (i \geq 1) \wedge (i < n)$$

Quelle que soit l'itération i de la boucle exécutée, la distance sera toujours bornée par 0 et par i .

En effet, si i caractères des mots ont déjà été traités, la distance de Hamming maximale à cet instant précis ne peut dépasser i (puisque dans le pire des cas, on a i caractères différents). De plus, cette distance est égale au cardinal de l'ensemble pour lequel $A_j \neq B_j, j \in \{1, \dots, i\}$

B. Tests

Tests fonctionnels (en boîte noire) :

- Paramètres incorrects :

Il est important de vérifier que notre programme traite correctement le cas de l'absence de paramètres saisi par l'utilisateur.

```
Clement:ProjetAAVP utilisateur$ ./distances
Erreur, trop peu d'arguments.
```

Dans le cas où aucun argument n'est saisi en entrée, le programme se termine bien correctement, en renvoyant un message d'erreur à l'utilisateur, tout en lui en expliquant la raison.

```
Clement:ProjetAAVP utilisateur$ ./distances ab ab ba
Hamming(ab,ab) = 0
```

Si cette fois on saisit trop d'arguments, ce n'est pas une source d'erreurs, puisque seuls les deux premiers seront traités, le troisième sera simplement ignoré par le programme.

- Le mot vide :

Lorsque l'on manipule des chaînes de caractères, il est intéressant de tester le programme avec la chaîne de caractère vide.

```
Clement:ProjetAAVP utilisateur$ ./distances "" ""
Hamming(,)= 0
```

Le programme renvoie bien le résultat attendu, 0, puisque les deux mots sont identiques.

- Mots de tailles différentes :

Cet algorithme ne fonctionnant que sur des chaînes de même taille, il est important de vérifier que le cas de chaînes de taille différentes en entrée est traité.

```
[Clement:ProjetAAVP utilisateur$ ./distances ab a
Erreur, les deux chaînes doivent être de taille identique.]
```

Ici encore, le résultat est bien celui attendu, un message d'erreur expliquant la raison de la non-exécution du programme.

- Mots d'un seul caractère :

Il peut être intéressant de voir si notre programme fonctionne correctement pour des mots d'un seul caractère. En effet, en se basant sur la spécification, on sait que notre programme doit renvoyer 0 si les deux caractères sont identiques, et 1 sinon.

```
[Clement:ProjetAAVP utilisateur$ ./distances ç ç
Hamming(ç,ç) = 0
[Clement:ProjetAAVP utilisateur$ ./distances ç é
Hamming(ç,é) = 1]
```

Le programme renvoie bien les résultats attendus.

- Mots composés :

Nous allons tester le programme pour des mots composés (d'un '-'). Le '-' étant considéré comme un caractère géré par l'ASCII, le programme devrait fonctionner normalement.

```
[Clement:ProjetAAVP utilisateur$ ./distances abat-jour bas-fonds
Hamming(abat-jour,bas-fonds) = 9]
```

Ceci correspond bien au résultat donné par la spécification.

- Mots au hasard :

Nous allons maintenant tester le programme sur des chaînes de caractères typiques, de même taille (de 2 à 10 caractères).

Nous comparerons les résultats obtenus avec les résultats issus de la spécification, à savoir :

```
Hamming("do","mi") = 2
Hamming("bal","dot") = 3
Hamming("fuis","fous") = 2
Hamming("casse","caste") = 1
Hamming("avocat","vertus") = 6
Hamming("volupte","accords") = 7
Hamming("unifiant","fasciner") = 7
Hamming("batiments","peintures") = 8
Hamming("contextuel","cultuelles") = 7
```

```
Clement:ProjetAAVP utilisateur$ ./distances do mi
Hamming(do,mi) = 2
Clement:ProjetAAVP utilisateur$ ./distances bal dot
Hamming(bal,dot) = 3
Clement:ProjetAAVP utilisateur$ ./distances fuis fous
Hamming(fuis,fous) = 2
Clement:ProjetAAVP utilisateur$ ./distances casse caste
Hamming(casse,caste) = 1
Clement:ProjetAAVP utilisateur$ ./distances avocat vertus
Hamming(avocat,vertus) = 6
Clement:ProjetAAVP utilisateur$ ./distances volupte accords
Hamming(volupte,accords) = 7
Clement:ProjetAAVP utilisateur$ ./distances unifiant fasciner
Hamming(unifiant,fasciner) = 7
Clement:ProjetAAVP utilisateur$ ./distances batiments peintures
Hamming(batiments,peintures) = 8
Clement:ProjetAAVP utilisateur$ ./distances contextuel cultuelles
Hamming(contextuel,cultuelles) = 7
```

- ▶ Omb
- ▶ Réfle
- ▶ Éclat
- ▶ Cont
- ▶ Mise

Le programme renvoie les mêmes valeurs que celles issues de la spécification.

Tests structurels :

Nous allons maintenant réaliser des tests structurels, qui contrairement aux tests fonctionnels effectués précédemment, s'intéressent à la structure du programme, aux instructions et à leur couverture.

Pour représenter la structure du programme, nous allons utiliser un flowchart.

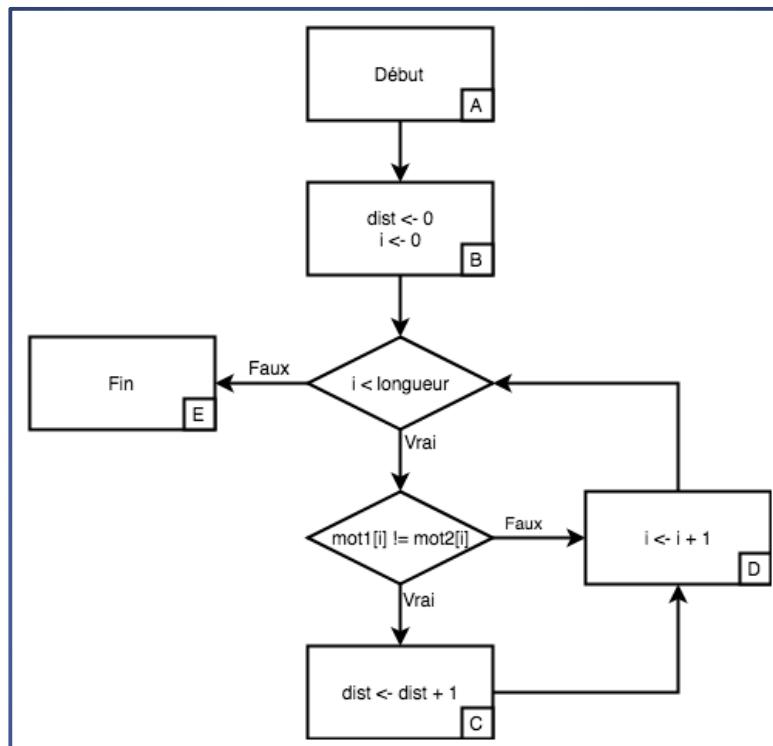


Figure 3 : Flowchart de l'algorithme de distance de Hamming

Source	Destination	Condition
A	B	Toutes chaînes de caractères
B	C	Chaînes d'au moins 1 caractère, commençant par un caractère différent
B	D	Chaînes d'au moins 1 caractère, commençant par le même caractère
B	E	Chaînes vides
C	D	Toutes chaînes permettant C
D	C	Chaînes d'au moins i caractères possédant un caractère d'index i différent
D	D	Chaînes d'au moins i caractères possédant un caractère d'index i identique
D	E	Chaînes de taille i

4. Complexité – Temps de calcul

Complexité :

Considérons deux mots A et B de longueur n .

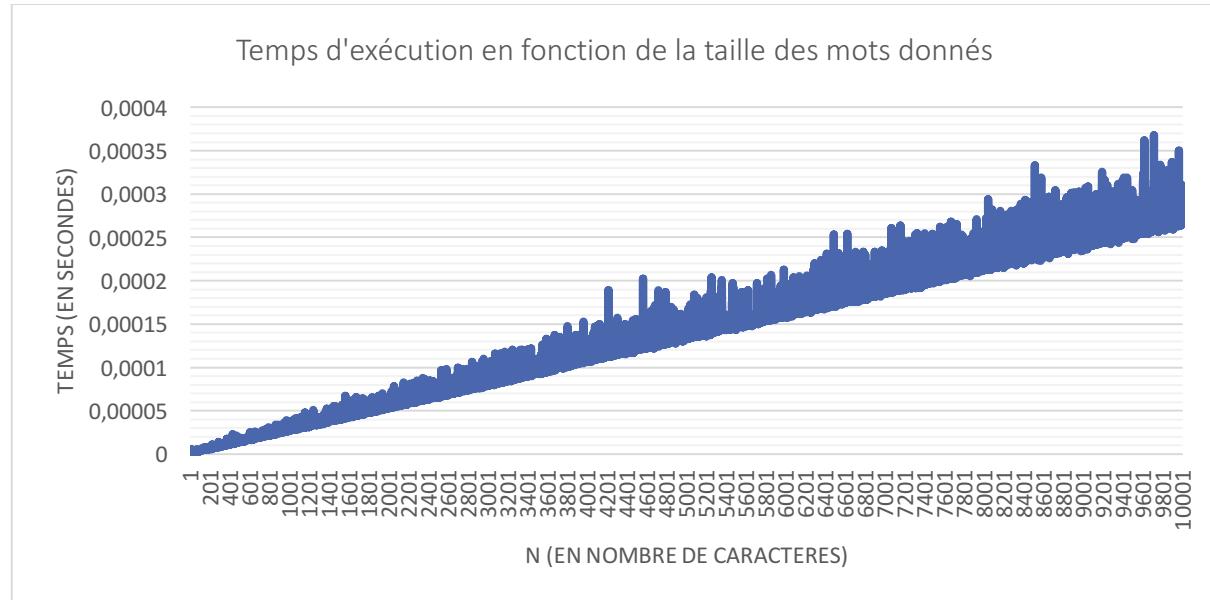
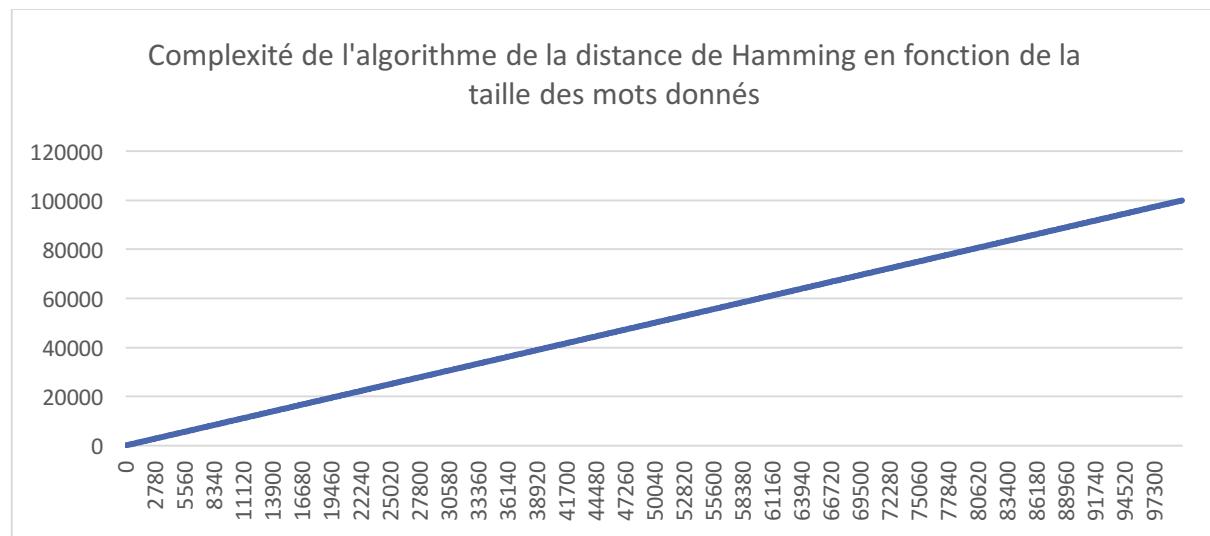
L'algorithme étant composé d'une seule boucle *Pour*, sa complexité en temps est donc de :

$$T = \theta(n)$$

Temps de calcul :

Afin d'évaluer la performance du programme, la taille des chaînes de caractères saisis en entrée du programme varie (de 1 à 10 001 caractères, avec un pas de 10)

Comme attendu, on obtient une droite qui correspond à la complexité linéaire en $\theta(n)$ calculée précédemment.



La distance de Levenshtein

1. L'algorithme

La distance de Levenshtein, aussi appelée distance d'édition, donne également une **mesure des différences entre deux chaînes de caractères**.

Elle tient son nom du mathématicien russe Vladimir Levenshtein, qui comme Richard Hamming, a consacré une grande partie de ses travaux à la théorie des codes.

Bien que conçue en 1965, cette distance est aujourd'hui encore **utilisée par de nombreux correcteurs orthographiques** (le programme va chercher dans un dictionnaire les mots présentant la distance de Levenshtein la plus faible avec le mot mal orthographié).

Elle pallie la principale limite de la distance de Hamming, puisqu'elle **fonctionne pour des chaînes de caractères de tailles différentes**.

Considérons deux mots A et B, la distance de Levenshtein entre A et B est le **coût minimal pour transformer A en B, en effectuant les trois opérations** que sont :

- L'**insertion** d'un caractère du mot A dans le mot B
- La **suppression** d'un caractère du mot A
- La **substitution** d'un caractère du mot A par un caractère (du mot B ou non)

Pour la suite, on supposera que le coût de chacune de ces opérations est unitaire.

Mathématiquement, on peut donc exprimer la distance de Levenshtein de deux mots de taille n et m comme étant $\text{levenshtein}_{A,B}(n,m)$, avec :

$$\text{levenshtein}_{A,B}(i,j) = \begin{cases} \max(i,j) \text{ si } \min(i,j) = 0 \\ \min \begin{cases} \text{levenshtein}_{A,B}(i-1,j) + 1 \\ \text{levenshtein}_{A,B}(i,j-1) + 1 \\ \text{levenshtein}_{A,B}(i-1,j-1) \text{ si } A[i] = B[j] \\ \text{levenshtein}_{A,B}(i-1,j-1) + 1 \text{ si } A[i] \neq B[j] \end{cases} \end{cases}$$

Où :

- $\text{levenshtein}_{A,B}(i-1,j) + 1$ correspond à la **suppression** d'un caractère du mot A
- $\text{levenshtein}_{A,B}(i,j-1) + 1$ correspond à l'**insertion** dans le mot A d'un caractère du mot B
- $\text{levenshtein}_{A,B}(i-1,j-1) + 1$ si $A[i] \neq B[j]$ correspond à la **substitution** d'un caractère de A par un autre caractère.
- $\text{levenshtein}_{A,B}(i-1,j-1)$ si $A[i] = B[j]$ correspond à l'**égalité** du caractère d'index i de A et d'index j de B. Dans ce cas, on recopie simplement la valeur précédente, sans ajouter de coût supplémentaire.

(i) Version Récursive

Grâce à cette formule, on peut écrire un premier algorithme récursif de cette distance de Levenshtein :

Algorithme : LevenshteinRE(**caractere** mot1[longueur1], **caractere** mot2[longueur2], **entier** longueur1, **entier** longueur2) : **entier**

Variables :

subst : **entier**

Début

Si longueur1 = 0 **alors**

Renvoyer longueur2

FinSi

Si longueur2 = 0 **alors**

Renvoyer longueur1

FinSi

Si mot1[longueur1 - 1] = mot2[longueur2 - 1] **alors**

 subst \leftarrow 0

Sinon

 subst \leftarrow 1

FinSi

Renvoyer min (

 LevenshteinRE(mot1,mot2,longueur1 - 1,longueur2) + 1,

 LevenshteinRE(mot1,mot2,longueur1,longueur2 - 1) + 1,

 LevenshteinRE(mot1,mot2,longueur1 - 1,longueur2 - 1) + subst)

Fin

Figure 4 : Algorithme récursif de la distance de Levenshtein

Cet algorithme prend donc deux tableaux de caractères ainsi que leurs tailles respectives en paramètre.

Si le premier mot est le mot vide, alors la distance de Levenshtein sera égale à la taille du second mot.

En effet, si on considère A le mot vide et B un mot quelconque, pour transformer le mot A et le mot B il va donc falloir insérer tous les caractères de B ce qui revient à un coût égal à la taille de B.

Inversement, si on considère A un mot quelconque et B le mot vide, pour transformer le mot A en B, il va falloir supprimer tous les caractères de A, ce qui revient à un coût égal à la taille de A.

On compare ensuite les derniers caractères des deux mots, s'ils sont identiques, le coût de substitution est nul, sinon, il est de 1.

On suit ensuite la définition, en effectuant trois appels récursifs, correspondant à :

- La suppression d'un caractère de A
- L'insertion d'un caractère dans le mot A
- La substitution d'un caractère de A par un caractère du mot B

On renvoie ensuite la valeur minimale de ces trois appels.

Rapidement, on remarque que cet algorithme ne sera pas performant pour des chaînes de caractères conséquentes, puisque les appels récursifs se révèleront très coûteux.

Pour pallier à cela, nous allons étudier une version itérative de cet algorithme.

(ii) Version Itérative

On peut modéliser ce problème sous forme de matrice, de $n+1$ lignes et $m+1$ colonnes :

DL	Mot B	B[0]	B[1]	...	B[j]	...	B[m-1]
Mot A	0	1	2	...	j+1	...	m
A[0]	1						
A[1]	2						
...	...						
A[i]	i+1						
...	...						
A[n-1]	n						DL[n][m]

Le principe reste issu de la formule étudiée précédemment, mais cette fois les résultats sont stockés dans cette matrice.

La première colonne correspond à la suppression des caractères du mot A, et la première ligne à l'insertion des caractères de B dans A, pour passer de A à B.

Une fois la matrice initialisée, pour chaque case $DL[i][j]$, il faut prendre la valeur minimale entre :

- $DL[i-1][j] + 1$ (ce qui correspond à la suppression d'un caractère de A)
- $DL[i][j-1] + 1$ (ce qui correspond à l'insertion d'un caractère de B dans A)
- $DL[i-1][j-1] + subst$ (subst valant 0 si $DL[i] = DL[j]$ ou 1 sinon).

Enfin, une fois la matrice remplie, chaque case $DL[i][j]$ contient la distance séparant les i-premières lettres du mot A des j-premières lettres du mot B.

Le résultat de la distance de Levenshtein sera stocké dans la dernière case de la matrice : $DL[n][m]$.

Remarque : Il peut y avoir plusieurs transformations différentes pour transformer un mot A en un mot B.

Dans la matrice, elles s'expriment à travers le chemin choisi, pour passer de la première case $DL[0][0]$ à la dernière case $DL[n][m]$.

Pour rappel :

- = insertion
- = substitution
- = suppression

Nous allons détailler le remplissage de la matrice à travers un exemple, prenons deux mots, A = Hexagone et B = Pentagone

DL	H	E	X	A	G	O	N	E	
	0	1	2	3	4	5	6	7	8
P	1								
E	2								
N	3								
T	4								
A	5								
G	6								
O	7								
N	8								
E	9								

DL	H	E	X	A	G	O	N	E	
	0	1	2	3	4	5	6	7	8
P	1	1	2						
E	2	2	1						
N	3								
T	4								
A	5								
G	6								
O	7								
N	8								
E	9								

$$DL[1][1] = \min(DL[0][1] + 1, DL[1][0], DL[0][0] + \text{subst})$$

$$DL[1][1] = \min(1 + 1, 1 + 1, 0 + 1) = \min(1, 2, 2) = 1$$

$$DL[1][2] = \min(DL[0][2] + 1, DL[1][1] + 1, DL[0][1] + \text{subst})$$

$$DL[1][2] = \min(2 + 1, 1 + 1, 1 + 1) = \min(3, 2, 2) = 2$$

$$DL[2][1] = \min(DL[1][1] + 1, DL[2][0] + 1, DL[1][0] + \text{subst})$$

$$DL[2][1] = \min(1 + 1, 2 + 1, 1 + 1) = \min(2, 3, 2) = 2$$

$$DL[2][2] = \min(DL[1][2] + 1, DL[2][1] + 1, DL[1][1] + \text{subst})$$

$$DL[2][2] = \min(2 + 1, 2 + 1, 1 + 0) = \min(3, 3, 1) = 1$$

Pour cette case, comme $A[1] = B[1] = 'E'$, le coût de substitution est nul.

On procède de la même manière pour chaque case de cette matrice, et on obtient le résultat ci-dessous :

DL	H	E	X	A	G	O	N	E	
	0	1	2	3	4	5	6	7	8
P	1	1	2	3	4	5	6	7	8
E	2	2	1	2	3	4	5	6	7
N	3	3	2	2	3	4	5	5	6
T	4	4	3	3	3	4	5	6	6
A	5	5	4	4	3	4	5	6	7
G	6	6	5	5	4	3	4	5	6
O	7	7	6	6	5	4	3	4	5
N	8	8	7	7	6	5	4	3	4
E	9	9	8	8	7	6	5	4	3

- [Yellow Box] = Substitution
- [Red Box] = Suppression
- [Green Box] = Identité

On obtient donc :

$$\text{DistanceLevenshtein}(\text{Pentagone}, \text{Hexagone}) = 3$$

On peut d'ailleurs vérifier ceci :

P	E	N	T	A	G	O	N	E
H	E		X	A	G	O	N	E

Pour transformer le mot *Pentagone* en *Hexagone*, on peut donc effectuer 2 substitutions de caractères ('P' par 'H' et 'T' par 'X') et une suppression ('N').

Comme précisé dans la remarque précédente, on aurait pu effectuer une autre séquence d'opérations pour arriver à ce résultat, par exemple :

DL	H	E	X	A	G	O	N	E	
	0	1	2	3	4	5	6	7	8
P	1	1	2	3	4	5	6	7	8
E	2	2	1	2	3	4	5	6	7
N	3	3	2	2	3	4	5	5	6
T	4	4	3	3	3	4	5	6	6
A	5	5	4	4	3	4	5	6	7
G	6	6	5	5	4	3	4	5	6
O	7	7	6	6	5	4	3	4	5
N	8	8	7	7	6	5	4	3	4
E	9	9	8	8	7	6	5	4	3

P	E	N	T	A	G	O	N	E
H	E	X		A	G	O	N	E

Ici on a simplement choisi d'effectuer la substitution avant la suppression, le coût pour transformer *Pentagone* en *Hexagone* demeure le même (3).

Voici donc la version itérative de l'algorithme de la distance de Levenshtein :

Algorithme : LevenshteinIT(**caractere** mot1[longueur1], **caractere** mot2[longueur2], **entier** longueur1, **entier** longueur2) : **entier**

Variables :

d[longueur1 + 1][longueur2 + 1] : **matrice (tableau 2D) d'entiers**

i,j : **entiers**

subst : **entier**

Début

Pour i allant de 0 à longueur1 **faire**

 d[i,0] \leftarrow i

FinPour

Pour j allant de 0 à longueur2 **faire**

 d[0,j] \leftarrow j

FinPour

Pour i allant de 1 à longueur1

Pour j allant de 1 à longueur2

Si mot1[i] = mot2[j] **alors**

 subst \leftarrow 0

Sinon

 subst \leftarrow 1

FinSi

 d[i,j] \leftarrow min(d[i-1,j] + 1, d[i,j-1] + 1, d[i-1,j-1] + subst)

FinPour

FinPour

Renvoyer d[longueur1, longueur2]

Fin

Figure 5 : Algorithme itératif (avec matrice) de la distance de Levenshtein

On procède de la même manière qu'étudiée précédemment, en initialisant la première ligne et la première colonne de la matrice.

Cette matrice est ensuite parcourue itérativement, et pour chaque case :

- On teste l'égalité du caractère d'indice i de mot1 avec celui d'indice j de mot2.
- On stocke à la case d'index i,j la valeur minimale entre l'insertion, la suppression ou la substitution d'un caractère.

On renvoie la dernière case de la matrice, qui contient la distance de Levenshtein entre les deux mots.

2. Implémentation en langage C

(i) Version Récursive

```
int distance_levenshtein_re(char* mot1, char* mot2, unsigned int tailleMot1, unsigned int tailleMot2)
{
    unsigned int eff,ins,sub,coutSub;

    if(tailleMot1 == 0)
        return tailleMot2;
    if(tailleMot2 == 0)
        return tailleMot1;

    if(mot1[tailleMot1 - 1] == mot2[tailleMot2 - 1])
        coutSub = 0;
    else
        coutSub = 1;

    eff = distance_levenshtein_re(mot1, mot2, tailleMot1 - 1, tailleMot2) + 1;
    ins = distance_levenshtein_re(mot1, mot2, tailleMot1, tailleMot2 - 1) + 1;
    sub = distance_levenshtein_re(mot1, mot2, tailleMot1 - 1, tailleMot2 - 1) + coutSub;
    return min(eff,ins,sub);
}
```

Figure 6 : Implémentation de l'algorithme récursif de la distance de Levenshtein

(ii) Version Itérative

```
int distance_levenshtein_it(char* mot1, char* mot2,unsigned int tailleMot1, unsigned int tailleMot2)
{
    unsigned int ** d = (unsigned int**) malloc (sizeof(unsigned int*) * (tailleMot1 + 1));
    unsigned int i,j,coutSub = 0,res;

    for(i = 0; i <= tailleMot1; i++)
    {
        d[i] = (unsigned int*) malloc(sizeof(unsigned int) * (tailleMot2 + 1));
        d[i][0] = i;
    }

    for(j = 0; j <= tailleMot2; j++)
        d[0][j] = j;

    for(i = 1; i <= tailleMot1; i++)
    {
        for(j = 1; j <= tailleMot2; j++)
        {
            if(mot1[i-1] == mot2[j-1])
                coutSub = 0;
            else
                coutSub = 1;
            d[i][j] = min(d[i-1][j] + 1, d[i][j-1] + 1, d[i-1][j-1] + coutSub);
        }
    }

    res = d[tailleMot1][tailleMot2];

    for(i = 0; i < tailleMot1; i++)
        free(d[i]);
    free(d);

    return res;
}
```

Figure 7 : Implémentation de l'algorithme itératif (avec matrice) de la distance de Levenshtein

3. Validité – Tests

A. Spécification et invariant de boucle

(i) Version Récursive

Spécification :

Si on considère deux mots A et B , de tailles respectives m et n , la distance de Levenshtein entre A et B doit :

- Etre bornée (par le bas) par la valeur absolue de la différence entre m et n :

$$|m - n| \leq d(m, n)$$
- Etre bornée (par le haut) par la plus grande taille :

$$d(m, n) \leq \max(m, n)$$

Il vient donc :

$$|m - n| \leq d(m, n) \leq \max(m, n)$$

On a également :

$$d(m, n) = \min(d(m - 1, n) + 1, d(m, n - 1) + 1, d(m - 1, n - 1) + 1_{A[m - 1] \neq B[n - 1]}$$

La spécification est donc l'union de ces deux propriétés :

$$\begin{cases} d(m, n) = \min(d(m - 1, n) + 1, d(m, n - 1) + 1, d(m - 1, n - 1) + 1_{A[m - 1] \neq B[n - 1]} \\ |m - n| \leq d(m, n) \leq \max(m, n) \end{cases}$$

(ii) Version Itérative

Spécification :

Considérons deux mots : A de taille n et B de taille m

À la fin de l'exécution de l'algorithme, et ce indépendamment des chaînes A et B saisies en paramètres, le résultat de la distance de Levenshtein se trouvera à la case $d[m, n]$ de la matrice.

La spécification de l'algorithme est donc :

$$d[m, n] = \text{levenshtein}(A, B)$$

$$d[m, n] = \begin{cases} \max(m, n) & \text{si } \min(m, n) = 0 \\ \min \begin{cases} d[m - 1, n] + 1 \\ d[m, n - 1] + 1 \\ d[m - 1, n - 1] + 1_{A[m - 1] \neq B[n - 1]} \end{cases} & \text{autre cas} \end{cases}$$

Invariant de boucle :

Considérons deux mots : A de taille n et B de taille m

L'algorithme étant composé de deux boucles imbriquées, nous allons donc déterminer deux invariants de boucle.

Commençons par l'invariant de la boucle située la plus à l'intérieur. Cette boucle effectue n itérations (pour j allant de 1 à n), et est exécutée m fois. Elle parcourt donc la totalité de la matrice, on a donc :

$$\left\{ \begin{array}{l} \forall x \in \{0..i-1\}, \forall y \in \{0..j-1\} : d[x,y] = \text{Levenshtein}(A[1..x], B[1..y]) \\ 1 \leq i \leq m+1 \\ 1 \leq j \leq n+1 \end{array} \right.$$

Cette propriété est vraie avant et après l'exécution de chaque itération. Elle illustre d'ailleurs la progression de notre boucle.

En effet, à la première itération, on a $d[x,y] = d[0][0] = 0$ (valeurs initiales de la matrice) et à la dernière itération, on aura $d[x,y] = d[m,n]$.

Concernant la première boucle, on peut trouver de la même manière l'invariant suivant :

$$\left\{ \begin{array}{l} \forall x \in \{0..i-1\}, \forall y \in \{0..n-1\} : d[x,y] = \text{Levenshtein}(A[1..x], B[1..y]) \\ 1 \leq i \leq m+1 \end{array} \right.$$

Ici encore, cette propriété est vérifiée avant et après chaque itération.

Lors de la première itération, les valeurs de la première colonne sont déjà calculées (initialisation de la matrice), puis ensuite, les valeurs de chaque colonne d'indice x sont préalablement calculées par la boucle intérieure lors de la précédente itération.

B. Tests

(i) Version Récursive

Tests fonctionnels (en boîte noire)

- Mauvais paramètres :

```
[pc5:ProjetAAVP utilisateur$ ./distances
Erreur, trop peu d'arguments.
```

- Gestion du mot vide :

Pour ce test, nous allons vérifier le retour du programme lorsque l'on exécute le programme avec deux chaînes vides, puis avec une chaîne vide et un mot.

D'après la spécification, la distance de Levenshtein entre deux mots vides est de 0, et celle entre un mot vide et un mot quelconque de taille t est de t.

```
[pc5:ProjetAAVP utilisateur$ ./distances "" ""
LevenshteinRE(,) = 0
[pc5:ProjetAAVP utilisateur$ ./distances "" mot
LevenshteinRE(,mot) = 3
```

Les retours du programme correspondent bien aux résultats attendus.

- Mots d'un seul caractère :

```
[pc5:ProjetAAVP utilisateur$ ./distances a b
LevenshteinRE(a,b) = 1
[pc5:ProjetAAVP utilisateur$ ./distances a a
LevenshteinRE(a,a) = 0
```

Ici encore, le programme renvoie bien les résultats attendus.

- Vérification de la symétrie :

La distance de Levenshtein est une distance au sens mathématique du terme, ce qui signifie qu'elle doit être symétrique.

$$\text{Levenshtein}(A, B) = \text{Levenshtein}(B, A)$$

Si on prend A = soir et B = noir, on montre aisément que :

$$\text{Levenshtein}(\text{noir}, \text{soir}) = \text{Levenshtein}(\text{soir}, \text{noir}) = 1$$

```
[pc5:ProjetAAVP utilisateur$ ./distances noir soir
LevenshteinRE(noir,soir) = 1
[pc5:ProjetAAVP utilisateur$ ./distances soir noir
LevenshteinRE(soir,noir) = 1
```

Le résultat du programme est conforme à nos attentes.

Prenons maintenant A = voir et B = recevoir, deux mots de taille différente. En se basant sur la spécification on trouve

$$\text{Levenshtein}(\text{recevoir}, \text{voir}) = \text{Levenshtein}(\text{voir}, \text{recevoir}) = 4$$

```
[pc5:ProjetAAVP utilisateur$ ./distances voir recevoir
LevenshteinRE(voir,recevoir) = 4
[pc5:ProjetAAVP utilisateur$ ./distances recevoir voir
LevenshteinRE(recevoir,voir) = 4
```

Ici encore, le résultat retourné par le programme est conforme à nos attentes

- Mots au hasard :

On va reprendre le même jeu de tests que pour la distance de Hamming.
En se basant sur la spécification, il vient :

*LevenshteinRE(do, mi) = 2
 LevenshteinRE(bal, dot) = 3
 LevenshteinRE(fuis, fous) = 2
 LevenshteinRE(casse, caste) = 1
 LevenshteinRE(avocat, vertus) = 6
 LevenshteinRE(unifiant, fasciner) = 7
 LevenshteinRE(batiments, peintures) = 8
 LevenshteinRE(contextuel, cultuelles) = 7*

```
pc5:ProjetAAVP utilisateur$ ./distances bal dot
LevenshteinRE(bal, dot) = 3
pc5:ProjetAAVP utilisateur$ ./distances fuis fous
LevenshteinRE(fuis, fous) = 2
pc5:ProjetAAVP utilisateur$ ./distances casse caste
LevenshteinRE(casse, caste) = 1
pc5:ProjetAAVP utilisateur$ ./distances avocat vertus
LevenshteinRE(avocat, vertus) = 6
pc5:ProjetAAVP utilisateur$ ./distances unifiant fasciner
LevenshteinRE(unifiant, fasciner) = 7
pc5:ProjetAAVP utilisateur$ ./distances batiments peintures
LevenshteinRE(batiments, peintures) = 8
pc5:ProjetAAVP utilisateur$ ./distances contextuel cultuelles
LevenshteinRE(contextuel, cultuelles) = 7
```

Tests structurels

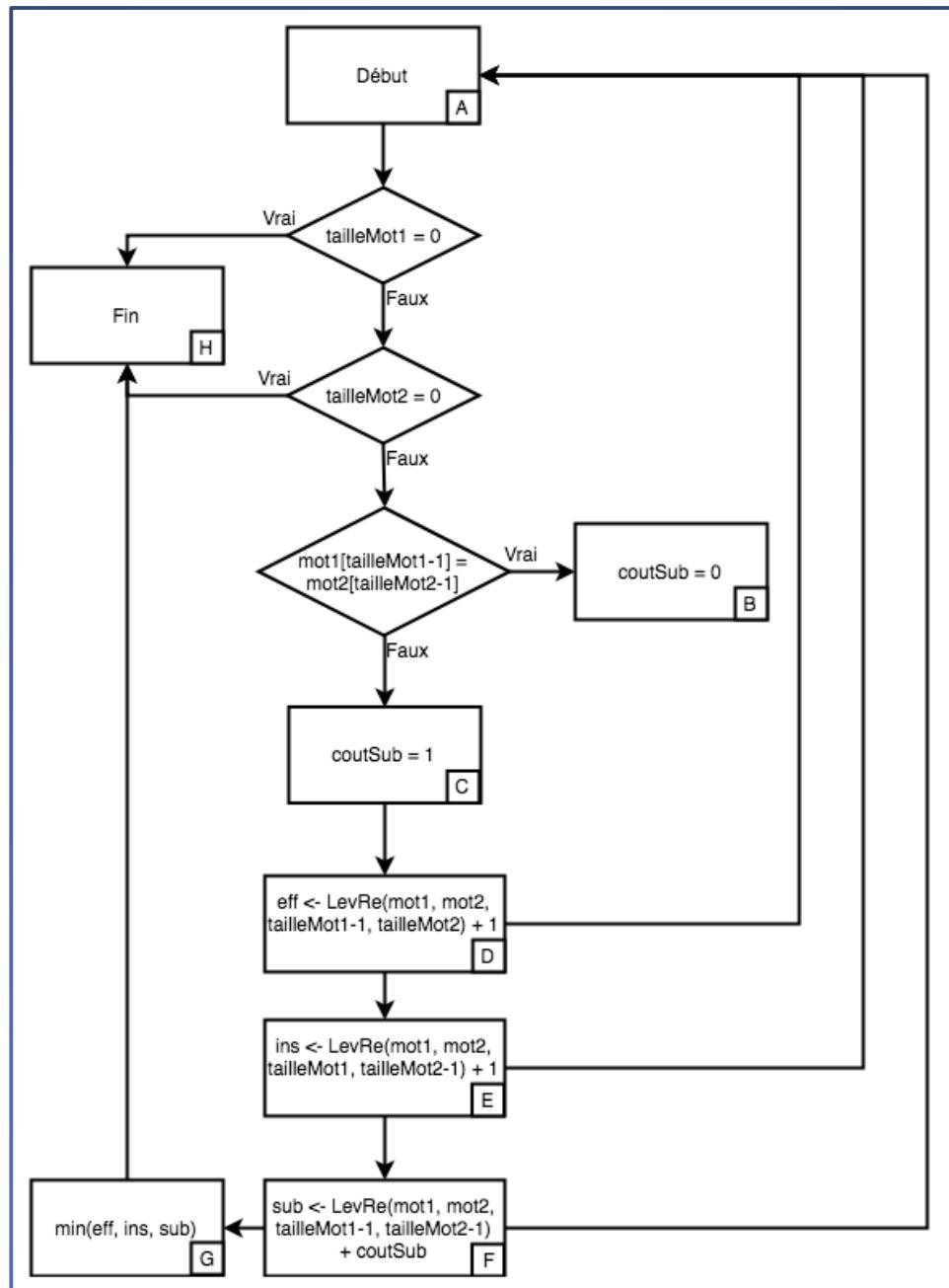


Figure 8 : Flowchart de l'algorithme de la distance de Levenshtein (version récursive)

Source	Destination	Condition
A	B	Deux chaînes d'au moins un caractère, et dont le dernier caractère est différent
A	C	Deux chaînes d'au moins un caractère, et dont le dernier caractère est identique
A	H	Au moins une des deux chaînes est vide
C	D	Toute chaîne de caractères satisfaisant C
D	E	Toutes chaînes de caractères satisfaisant D
D	A	Toutes chaînes de caractères satisfaisant D
E	F	Toutes chaînes de caractères satisfaisant E
E	A	Toutes chaînes de caractères satisfaisant E
F	G	Toutes chaînes de caractères satisfaisant F
F	A	Toutes chaînes de caractères satisfaisant F
G	H	Toutes chaînes de caractères satisfaisant G

(ii) Version Itérative

Tests fonctionnels (en boîte noire)

Le jeu de test choisi est le même que pour la version récursive.

On obtient les résultats suivants :

```
[pc5:ProjetAAVP utilisateur$ ./distances
Erreur, trop peu d'arguments.
[pc5:ProjetAAVP utilisateur$ ./distances "" ""
LevenshteinIT(,) = 0
[pc5:ProjetAAVP utilisateur$ ./distances "" mot
LevenshteinIT(,mot) = 3
[pc5:ProjetAAVP utilisateur$ ./distances a b
LevenshteinIT(a,b) = 1
[pc5:ProjetAAVP utilisateur$ ./distances a a
LevenshteinIT(a,a) = 0
[pc5:ProjetAAVP utilisateur$ ./distances noir soir
LevenshteinIT(noir,soir) = 1
[pc5:ProjetAAVP utilisateur$ ./distances soir noir
LevenshteinIT(soir,noir) = 1
[pc5:ProjetAAVP utilisateur$ ./distances voir recevoir
LevenshteinIT(voir,recevoir) = 4
[pc5:ProjetAAVP utilisateur$ ./distances recevoir voir
LevenshteinIT(recevoir,voir) = 4
[pc5:ProjetAAVP utilisateur$ ./distances do mi
LevenshteinIT(do,mi) = 2
[pc5:ProjetAAVP utilisateur$ ./distances bal dot
LevenshteinIT(bal,dot) = 3
[pc5:ProjetAAVP utilisateur$ ./distances fuis fous
LevenshteinIT(fuis,fous) = 2
[pc5:ProjetAAVP utilisateur$ ./distances casse caste
LevenshteinIT(casse,caste) = 1
[pc5:ProjetAAVP utilisateur$ ./distances avocat vertus
LevenshteinIT(avocat,vertus) = 6
[pc5:ProjetAAVP utilisateur$ ./distances unifiant fasciner
LevenshteinIT(unifiant,fasciner) = 7
[pc5:ProjetAAVP utilisateur$ ./distances batiments peintures
LevenshteinIT(batiments,peintures) = 8
[pc5:ProjetAAVP utilisateur$ ./distances contextuel cultuelles
LevenshteinIT(contextuel,cultuelles) = 7
```

On constate que le programme de l'algorithme de distance de Levenshtein itératif produit les mêmes résultats que le récursif, mais surtout que ceux obtenus à partir de la spécification de l'algorithme.

Tests structurels

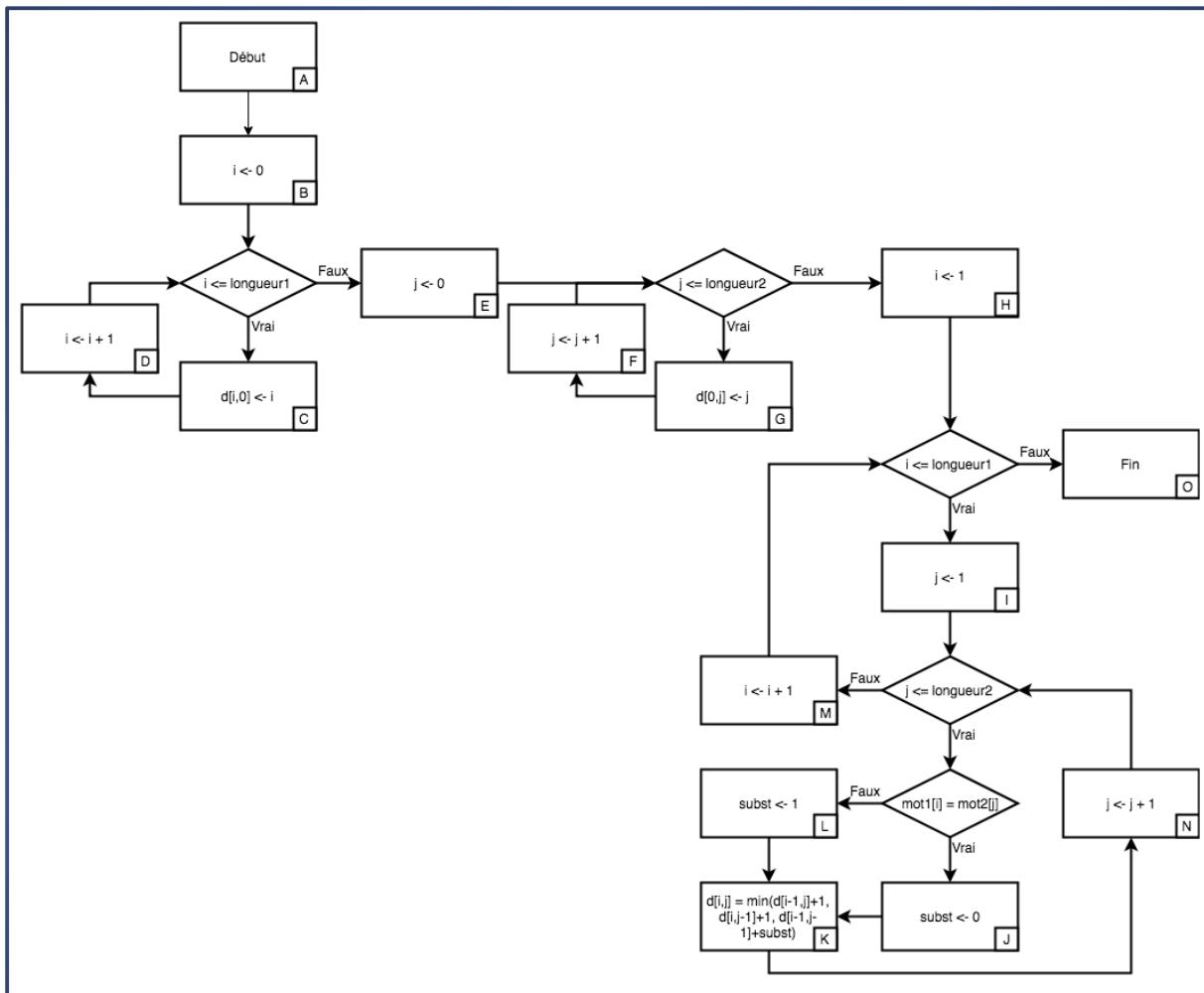


Figure 9 : Flowchart de l'algorithme de distance de Levenshtein (version itérative)

Source	Destination	Condition
A	B	Toutes chaînes de caractères
B	C	Toutes chaînes de caractères
B	E	Impossible (on passe au moins une fois dans C)
C	D	Toutes chaînes de caractères
D	C	Mot1 non vide, Mot1 de taille > i
D	E	Mot1 vide, Mot1 de taille <= i
E	G	Toutes chaînes de caractères

E	H	Impossible (on passe au moins une fois dans G)
G	F	Toutes chaînes de caractères
F	G	Mot2 non vide, Mot2 de taille $> j$
F	H	Mot2 vide, Mot2 de taille $\leq j$
H	I	Mot1 non vide
H	O	Mot1 vide
I	M	Mot2 vide
I	J	Mot1 non-vide et Mot2 non-vide et premier caractère de Mot1 identique au premier caractère de Mot2
I	L	Mot1 non-vide et Mot2 non-vide et premier caractère de Mot1 différent du premier caractère de Mot2
J	K	Toutes chaînes de caractères satisfaisant J
L	K	Toutes chaînes de caractères satisfaisant L
K	N	Toutes chaînes de caractères satisfaisant K
N	J	Mot1 et Mot2 non-vides et Mot1 de taille $> i$ et Mot2 de taille $> j$ et caractère d'indice i de Mot1 identique au caractère d'indice j de Mot2
N	L	Mot1 et Mot2 non-vides et Mot1 de taille $> i$ et Mot2 de taille $> j$ et caractère d'indice i de Mot1 différent du caractère d'indice j de Mot2
N	M	Mot2 de taille $< j$
M	I	Mot1 de taille $\geq i$
M	O	Mot1 de taille $< i$

4. Complexité – Temps de calcul

(i) Version Récursive

Complexité :

Dans un premier temps, dressons l'arbre d'exécution pour deux chaînes de deux caractères : A = ab et B = cd.

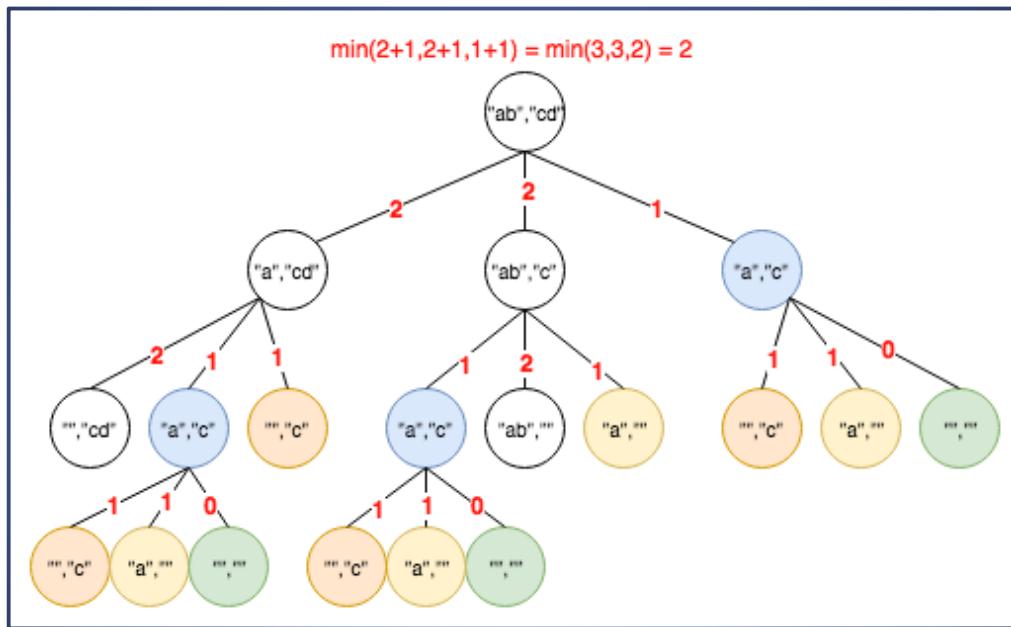


Figure 10 : Arbre d'exécution de l'algorithme récursif de distance de Levenshtein pour A = ab et B = cd

Pour m = 2 et n = 2, on remarque qu'il y a eu 18 appels récursifs à la fonction.

La coloration des nœuds permet également de mettre en exergue la principale limite de cette implémentation « naïve » de la distance de Levenshtein : la redondance des calculs. En effet, on remarque que les distances de Levenshtein des mêmes sous-chaînes ont été calculées plusieurs fois.

Revenons à la complexité de cet algorithme.

Si on considère m et n les deux tailles respectives des deux mots (mot1 et mot2), la complexité en temps de cet algorithme est :

$$\begin{cases} T(m, n) = T(m - 1, n) + T(m, n - 1) + T(m - 1, n - 1) + 1 \\ T(0, n) = m \\ T(m, 0) = n \end{cases}$$

Comme on peut le constater sur notre arbre, le chemin le plus court est de longueur $\min(m, n)$ (ici 2), donc l'arbre d'exécution contiendra forcément au moins $3^{\min(m,n)}$ nœuds.

La borne inférieure est donc :

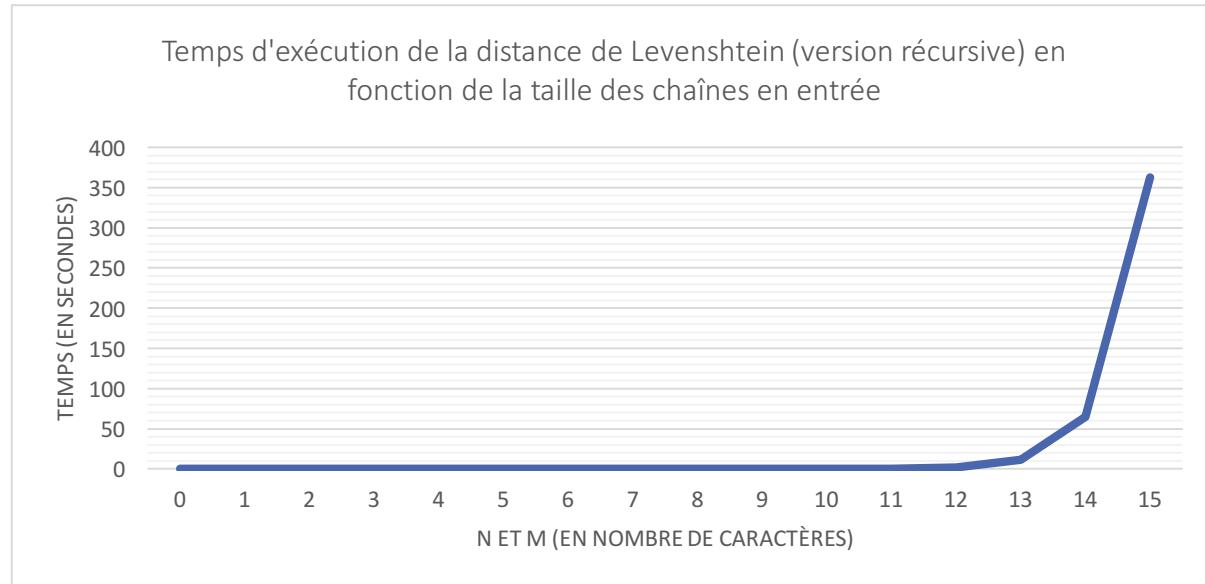
$$T(m, n) = \Omega(3^{\min(m,n)})$$

A contrario, le chemin le plus long est obtenu en réduisant les deux chaînes de caractères l'une après l'autre (par exemple le chemin $(ab, cd) \rightarrow (a, cd) \rightarrow (a, c) \rightarrow (, c)$). Ce chemin est de longueur 3, ce qui correspond à une longueur de $m+n-1$.

La borne supérieure est donc :

$$T(m, n) = O(3^{m+n-1})$$

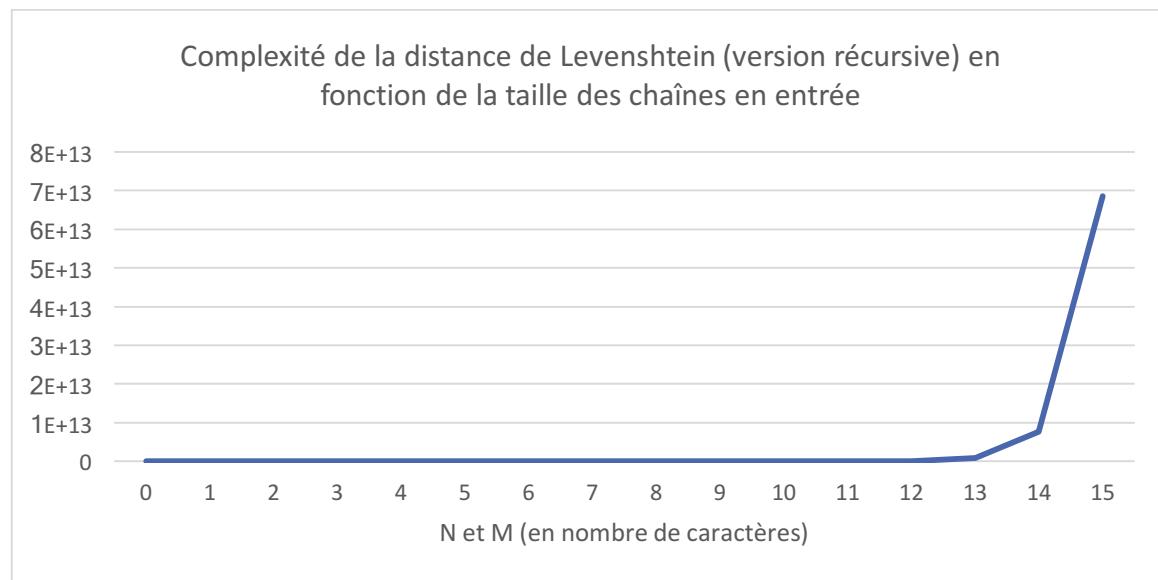
Temps de calcul :



Comme on pouvait s'y attendre, au vu de la complexité, cet algorithme n'est pas du tout performant.

En effet, pour calculer la distance de Levenshtein pour des mots de 12 caractères, cet algorithme prendra plus de 2 secondes.

On atteint plus de 360 secondes d'exécution pour des mots de 15 caractères.



(ii) Version Itérative

Complexité :

Si on considère m et n les deux tailles respectives des deux mots (mot1 et mot2), la complexité en temps de cet algorithme est :

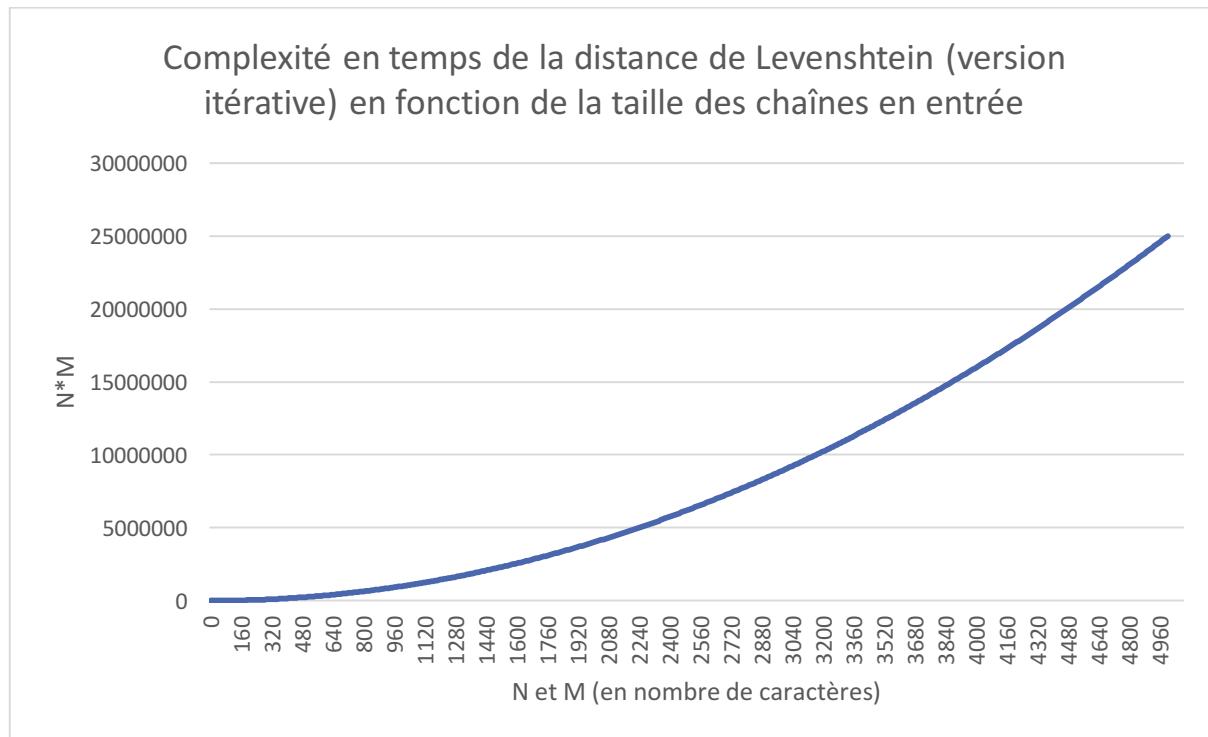
$$T = m + 1 + \underbrace{n + 1}_{\text{1ère ligne}} + \underbrace{m * n}_{\text{Remplissage de la matrice}}$$

1^{ère} colonne

En factorisant, on obtient :

$$\begin{aligned} T &= (m + 1) * (n + 1) + 1 \\ T &= O(m * n) \end{aligned}$$

Ce qui nous donne la courbe suivante :



Temps de calcul :

Durant l'exécution du script, j'ai pu constater une limite de l'algorithme initial, dans lequel je manipulais la matrice de taille $(m + 1) * (n + 1)$ de manière statique.

En effet, passé un certain seuil (mots de 1450 caractères environs), des erreurs de segmentations sont apparues.

Les variables statiques étant allouées sur la pile (*stack*), et une limite d'utilisation de cette dernière est certainement la source de ces erreurs.

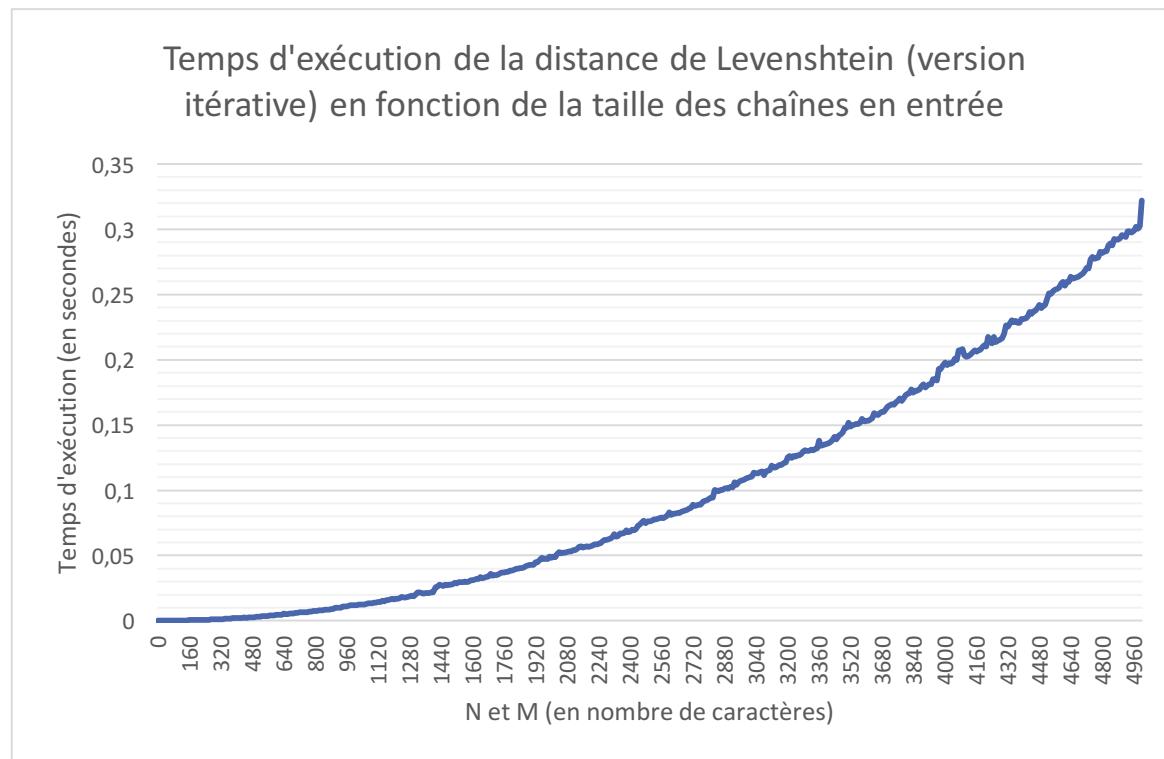
J'ai donc choisi ensuite de modifier le programme, de façon à gérer la matrice dynamiquement, à l'aide de la fonction *malloc*.

Les variables dynamiques sont allouées sur le tas (*heap*).

L'inconvénient de cette solution est une augmentation du temps d'exécution.

Pour tester la performance de notre programme, le même script que celui de la distance de Hamming a été utilisé.

Le programme a donc été exécuté avec des chaînes d'une taille variant de 0 à 500 caractères.



La courbe de temps d'exécution obtenue est de la même forme que celle de complexité.

La distance de Damerau-Levenshtein

1. L'algorithme

La distance de Damerau-Levenshtein est une amélioration de la distance de Levenshtein traitée dans la partie précédente.

Cette amélioration a été apportée par le chercheur en traitement du langage naturel et exploration de données *Frederick J. Damerau*.

Elle consiste en l'ajout d'une opération à la distance de Levenshtein.

Pour rappel, la distance de Levenshtein considérait trois opérations : la **suppression**, l'**ajout** et la **substitution** d'un caractère.

Damerau en a donc ajouté une quatrième : **la transposition de deux caractères adjacents**, en déclarant que l'ensemble de ces quatre opérations représenteraient environ plus de **80% des fautes d'orthographe humaines**.

Concernant l'algorithme, cela correspond donc à l'ajout de la condition suivante à l'algorithme de la distance de Levenshtein :

Si $i > 1$ **et** $j > 1$ **et** $\text{mot1}[i] = \text{mot2}[j-1]$ **et** $\text{mot1}[i-1] = \text{mot2}[j]$ **alors**
 $d[i,j] \leftarrow \min(d[i,j], d[i-2,j-2] + \text{subst})$

FinSi

Figure 11 : Condition de transposition de deux caractères

On obtient donc l'algorithme suivant :

```

Algorithme : DamerauLevenshtein(caractere mot1[longueur1], caractere mot2[longueur2],
entier longueur1, entier longueur2) : entier

Variables :
    d[longueur1 + 1][longueur2 + 1] : matrice (tableau 2D) d'entiers
    i,j : entiers
    subst : entier

Début
    Pour i allant de 0 à longueur1 faire
        d[i,0]  $\leftarrow$  i
    FinPour

    Pour j allant de 0 à longueur2 faire
        d[0,j]  $\leftarrow$  j
    FinPour

    Pour i allant de 1 à longueur1
        Pour j allant de 1 à longueur2
            Si mot1[i] = mot2[j] alors
                subst  $\leftarrow$  0
            Sinon
                subst  $\leftarrow$  1
            FinSi

            d[i,j]  $\leftarrow$  min(d[i-1,j] + 1, d[i,j-1] + 1, d[i-1,j-1] + subst)

            Si i > 1 et j > 1 et mot1[i] = mot2[j-1] et mot1[i-1] = mot2[j] alors
                d[i,j]  $\leftarrow$  min(d[i,j], d[i-2,j-2] + subst)
            FinSi

        FinPour
    FinPour

    Renvoyer d[longueur1, longueur2]
Fin
```

Figure 12 : Algorithme de Damerau-Levenshtein

2. Implémentation en langage C

```

int distance_damerau_levenshtein(char* mot1, char* mot2, unsigned int tailleMot1, unsigned int tailleMot2)
{
    unsigned int ** d = (unsigned int**) malloc (sizeof(unsigned int*) * (tailleMot1 + 1));
    unsigned int i,j,coutSub = 0, res;

    for(i = 0; i <= tailleMot1; i++)
    {
        d[i] = (unsigned int*) malloc(sizeof(unsigned int) * (tailleMot2 + 1));
        d[i][0] = i;
    }
    for(j = 0; j <= tailleMot2; j++)
        d[0][j] = j;

    for(i = 1; i <= tailleMot1; i++)
    {
        for(j = 1; j <= tailleMot2; j++)
        {
            if(mot1[i-1] == mot2[j-1])
                coutSub = 0;
            else
                coutSub = 1;

            d[i][j] = min(d[i-1][j] + 1, d[i][j-1] + 1, d[i-1][j-1] + coutSub);

            if((i > 1) && (j > 1) && (mot1[i-1] == mot2[j-2]) && (mot1[i-2] == mot2[j-1])) // Transposition de deux caractères adjacents
                if(d[i-2][j-2] + coutSub < d[i][j])
                    d[i][j] = d[i-2][j-2] + coutSub;
        }
    }

    res = d[tailleMot1][tailleMot2];
    for(i = 0; i < tailleMot1; i++)
        free(d[i]);
    free(d);
}

return res;
}

```

Figure 13 : Implémentation en langage C de la distance de Damerau-Levenshtein

Cet algorithme étant quasiment identique à la version itérative de l'algorithme de distance de Levenshtein, nous ne détaillerons donc pas sa validité et sa complexité, qui sont sensiblement les mêmes.

Limites et axes d'améliorations

Cette partie vise à dresser une liste non-exhaustive des limites rencontrées dans le cadre de ce projet, ainsi que différentes améliorations que l'on pourrait y apporter.

- La version récursive de la distance de Levenshtein :

Comme nous avons pu le constater au cours de son étude, cette version de l'algorithme de Levenshtein est très inefficace et coûteuse, et ce même pour une utilisation « normale ». En effet, les distances de Levenshtein des mêmes sous-chaînes sont recalculées plusieurs fois.

Pour pallier à cela, une solution pourrait être de mettre en place un tableau à deux dimensions, dans lequel, à la manière de la version itérative, chaque case $d[i][j]$ de la matrice contiendrait la distance de Levenshtein entre les i-premiers caractères du mot A et les j-premiers caractères du mot B.

- La version itérative de la distance de Levenshtein (et Damerau-Levenshtein) :

Ces algorithmes possèdent une complexité en temps et en mémoire en $O(m * n)$.

Il est possible d'optimiser cet algorithme en mémoire, en ne conservant que deux lignes de la matrice : celles d'indices i et i-1.

- La distinction d'accents :

Dans la langue française, les erreurs d'accents assez courantes, il est donc important de constater la manière dont le programme gère ceci.

En effet, l'ASCII (American Standard Code for Information Interchange) ne prend pas en charge les caractères accentués présents dans la langue française.

Pour pallier à cela, il est possible de changer l'encodage, ou bien le type de caractère utilisé (`wchar_t` par exemple).

- Autres distances :

Il existe d'autres distances permettant de mesurer la similarité de chaînes (de caractères ou de tout autre type), on pourrait citer par exemple :

- La distance de Jaro-Winkler, qui renvoie un « coefficient de similarité » entre deux chaînes. Si ce coefficient vaut 0, cela représente l'absence de similarité, et au contraire, s'il vaut 1, les chaînes comparées sont égales.
- La distance de Jaccard, qui permet d'évaluer la similarité entre deux ensembles, avec, si on considère A, B deux ensembles :

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Conclusion

Comme nous avons pu le constater, les distances de Levenshtein et de Damerau-Levenshtein sont des outils mathématiques très efficaces pour la correction orthographique.

Elles possèdent également de nombreuses autres applications, parmi lesquelles le séquençage d'ADN, la reconnaissance de formes, la reconnaissance vocale, etc.

Ce projet d'Analyse d'Algorithmes et de Validation de Programmes (AAVP) m'a permis de mettre en pratique les notions étudiées en cours, et ce de manière plus concrète que durant les Travaux Dirigés de ce module.

En effet, de la conception et l'analyse de l'algorithme (spécification, invariant de boucle, pseudocode) jusqu'à son implémentation, sa validation et ses tests de performance, j'ai pu mettre en pratique toutes ces notions pour répondre à un seul et même problème : la similarité des chaînes de caractères.

Ce projet s'est révélé très instructif, et l'expérience qu'il nous a apporté nous sera forcément utile au cours de notre vie professionnelle, au cours de laquelle nous serons certainement amenés à réaliser des études et des tests similaires, sur des algorithmes et des programmes potentiellement plus conséquents.

A tort, durant nos précédents projets (liés au développement), nous n'utilisions que partiellement les notions d'analyse d'algorithmes et de validations de programmes (principalement des tests unitaires).

Toutefois, ce projet m'a permis de comprendre leur importance, et, bien que parfois difficiles à mettre en place, je suis persuadé que d'appliquer ces notions nous aurait été d'une grande aide, et nous aurait permis de soulever un certain nombre de problèmes au sein de nos codes.

Webographie

https://fr.wikipedia.org/wiki/Distance_de_Levenshtein

<http://www-igm.univ-mlv.fr/~lecrocq/cours/distances.pdf>