Projet Analyse Algorithme Validation de Programme

Similarité de chaines de caractères

Akli CHELLALA

Encadré par : M. SOHIER

Table des matières

Introduction :

En informatique théorique la mesure de distance entre mots est une manière de représenter la déférence entre deux mots ou deux chaines de caractères par un nombre.

Ce mécanisme est utilisé dans un correcteur des mots ou lors de la vérification de l'orthographe des mots. On prend l'exemple de la recherche d'un mot dans un dictionnaire, et la recherche approximative. Pour trouver un mot et son orthographe, la recherche consiste à parcourir un dictionnaire, si le mot est introuvable on va retourner les mots les plus proches, donc les mots qui ressemble au mot rechercher.

Dans ce rapport nous allons étudier et analyser deux algorithmes, la distance de Hamming, et la distance de Levenshtein pour mesurer la similarité de deux mots (chaines de caractères), puis on va présenter et donner l'algorithme de Jaro qui calcule le taux de similarité entre deux chaines de caractères.

Le but de notre travail étant d'introduire l'algorithme, en expliquant son fonctionnement et présentant son pseudocode. Ensuite donner une implémentation de l'algorithme utilisant le langage de programmation C. En fin, allons étudier la complexité de l'algorithmes et nous montrerons la validité de chaque algorithme.

I. La distance de Hamming :

1. Présentation de l'algorithme :

La distance de Hamming et une notion qui est défini par Richard Hamming, permet de quantifier la différence entre deux séquences de symboles de même taille.

Cette notion est utilisé dans plusieurs disciplines comme la théorie de l'information, la théorie des codes et la cryptographie, notamment le traitement de signal et dans les télécommunications.

Dans ce repport nous allons utiliser l'algorithme de la distance de Hamming pour comparer deux chaines de caractères, puis il renvoie un nombre qui représente la distance ou la différence entre les deux mots.

Algorithme distance de Hamming :

Entrée :
Mot1 de longueur (n)
Mot2 de longueur (n)
La longueur n
Sortie :
distance : distance entre le mot 1 et le mot 2

```
Début
    distance = 0
    Pour i = 1 jusqu'à longueur (n)
        Si Mot1[i] ≠ Mot2[i] alors
        distance := distance + 1
        FinSi
    FinPour
    Renvoyer distance
Fin
```

Figure 1 : Algorithme de la distance de Hamming

Le principe de cet l'algorithme consiste à parcourir les deux mots par une boucle. Pour chaque itération de boucle : l'algorithme compare le caractère de l'index **i** du Mot1 avec le caractère de l'index **i** du Mot2, si les deux caractères sont différents, on incrémente la variable distance.

Exemple de déroulement de l'algorithme :

On prend deux mots A = **allocation et B = allocution** de longueur **n =10 :**

A	**A**	**L**	**L**	**O**	**C**	**A**	**T**	**I**	**O**	**N**
B	A	L	L	O	C	U	T	I	O	N
Distance	0	0	0	0	0	1	0	0	0	0

Distance Hamming de (A,B) = 0+0+0+0+0+1+0+0+0+0 = **1**

La différence de Hamming entre le mot A et le mot B de taille 10 est **1,** on constate que les deux mots ne sont pas éloignés.

Cet algorithme fonctionne qu'avec des mots qui ont la même longueur, ce qui le rend inefficace.

Si on insère une lettre quelconque dans le mot A, la position de cette lettre influera très fortement sur la valeur de la distance. MotA = **allocation** MotB'= **alloocatio**

A	**A**	**L**	**L**	**O**	**C**	**A**	**T**	**I**	**O**	**N**
B'	A	L	L	O	O	C	A	T	I	O
Distance	0	0	0	0	1	1	1	1	1	1

Distance Hamming(A,B) = 0+0+0+0+1+1+1+1+1+1 = **6**

La différence de Hamming entre le mot A et le mot B' de taille 10 est **6,** on voit bien que l'ajout de la lettre 'O' influence fortement sur la distance de Hamming.

2. Implémentation de l'algorithme de la distance de Hamming :

L'implémentation algorithme de la distance de Hamming va se baser sur pseudocode qu'on a présenté dans la première section. On ajoute une condition au début pour vérifier que les deux mots ont une même taille. Si cette condition n'est pas vérifiée, alors on quitte le programme.

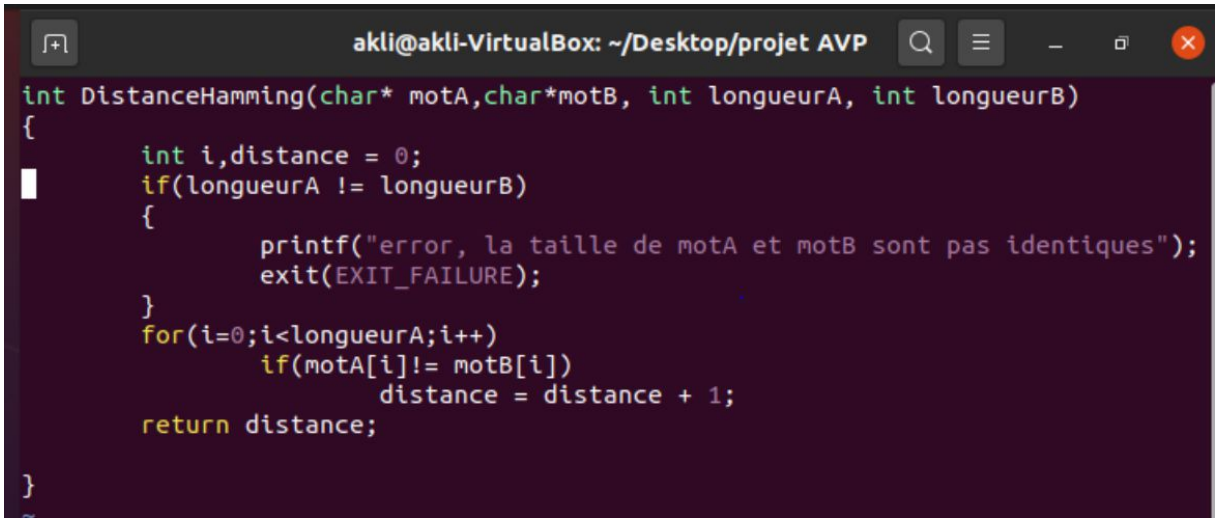


Figure 2 : Implémentation de l'algorithme de la distance de Hamming en langage C

3. Etude de validation de l'algorithme de Hamming :

a. Invariant de boucle

Un invariant de boucle est une propriété qui est vérifié avant et après chaque itération de boucle. L'algorithme de Hamming permet de calculer la distance entre deux mots de même longueur, autrement dit, il permet de calculer le nombre de caractères ou deux mots sont déférents. Ce nombre peut etre égale à 0 alors les deux mots sont identiques, ou supérieure à 0 ou inférieure ou égale à la longueur des deux mots, donc ce cas on peut dire que les deux mots ne sont pas identiques.

Algébriquement on peut dire que la distance de Hamming est bornée entre 0 et la longueur de deux mots n, et la distance est égale au nombre(cardinale) de caractères dans lesquelles A et B sont différents. On peut exprimer cela avec la spéciation suivante :

On considère A et B deux mots de même longueur tel que A_i et B_i (i de 1 à n) sont des caractères de mots A et mot B : A={A_1,A_2,…A_n} et B={B_1,B_2,…B_n} :

$$(0 \leq distance \leq n) \wedge (distance = Card(\{i \in \{1, \dots n\}\}) \text{tel que } A_i \neq B_i)$$

De plus, si i est le nombre de caractères dans les deux mots qui sont déjà été traités, la distance de Hamming sera bornée entre 0 et i(le nombre de caractères traitées), alors on peut dire que la distance est égale au cardinale card($A_k \neq B_k$)tel que k∈ {1, … i}, alors on peut avoir l'expression suivante à chaque itération de boucle :

$$(0 \leq distance \leq n) \wedge (distance = Card(\{k \in \{1, \dots i\}\}) \text{tel que } A_k \neq B_k) \wedge (1 \leq i \leq n)$$

- **Organigramme de l'algorithme de la distance de Hamming :**

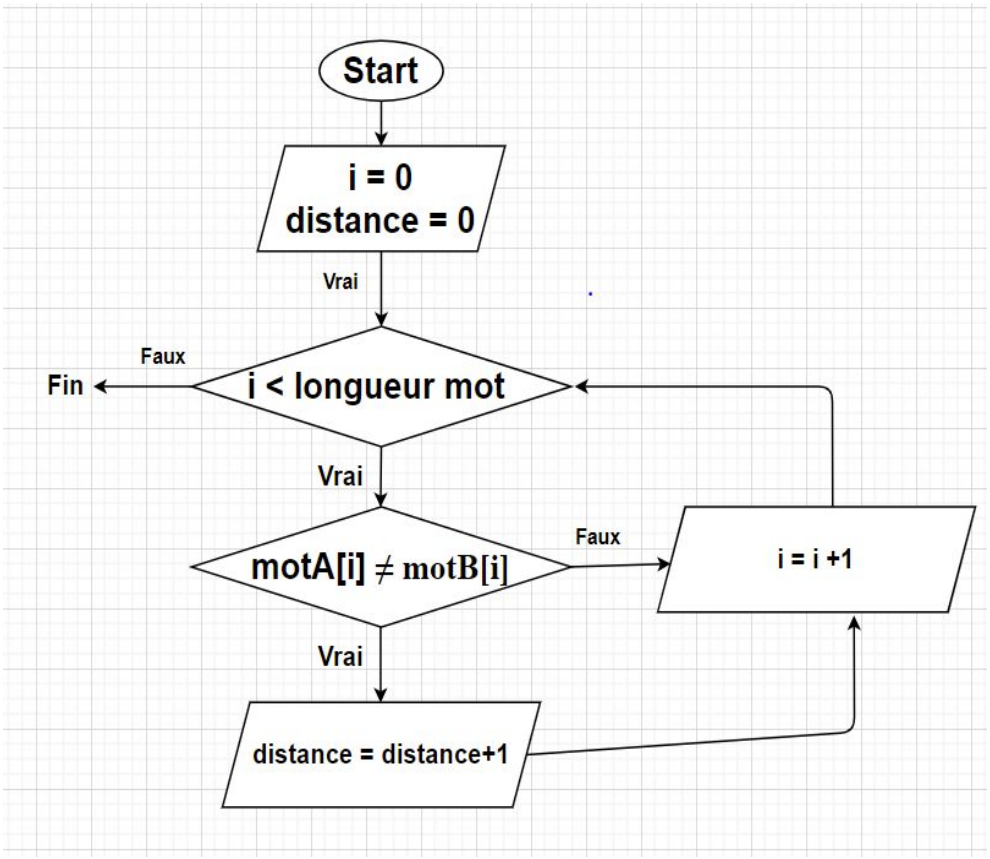


Figure 3 : **Organigramme** de l'algorithme Hamming

b. Test de l'implémentation :

1. Test avec deux mots similaires :

```
akli@akli:~/Bureau/projet AVP$ ./DisctanceHamming allocation allocation
La distance de Hamming de (allocation,allocation) = 0
```

Le programme nous retourne un **0** puisque on saisit deux mots identiques et de même longueur.

2. Test avec deux mots différents de même longueur :

```
akli@akli:~/Bureau/projet AVP$ ./DisctanceHamming allocation allocution
La distance de Hamming de (allocation,allocution) = 1
```

Comme dans notre exemple, le programme nous retourne un **1** puisque on saisit deux mots de même taille, mais qui sont différents avec une lettre « a » au lieu de « u ».

```
akli@akli:~/Bureau/projet AVP$ ./DisctanceHamming allocation alloocutio
La distance de Hamming de (allocation,alloocutio) = 6
```

Comme dans le deuxième exemple, Le programme nous retourne une distance de 6 puisque on saisit deux mots de même taille, avec deux mots différents en ajoutant O.

3. Test avec deux mots d'une longueur déférentes :

```
akli@akli:~/Bureau/projet AVP$ ./DisctanceHamming allocation alloocut
les deux mots ne sont pas de la meme longueur
```

Le programme nous retourne une erreur, les deux mots ne sont pas de la même longueur.

4. Test avec deux mots vides :

```
akli@akli:~/Bureau/projet AVP$ ./DisctanceHamming "" ""
La distance de Hamming de (,) = 0
```

Notre programme retourne 0 puisque on saisit deux mots vide qui sont identique.

Dans tous les tests effectués on constate que le programme retourne toujours le résultat attendu.

4. Etude de la complexité et Temps de calcule de l'algorithme de Hamming :

Algorithme distance de Hamming :

Entrée :
Mot1 de longueur (n)
Mot2 de longueur (n)
La longueur n
Sortie :
distance : distance entre le mot 1 et le mot 2

Début

Θ(1) { distance = 0

Θ(n) { **Pour** i = 1 jusqu'à longueur (n)
Si Mot1[i] ≠ Mot2[i] alors
distance := distance + 1
FinSi
FinPour

Θ(1) { **Renvoyer** distance

Fin

L'algorithme commence avec une initialisation de la variable distance qui est en **Θ(1)** , puis d'une seul boucle Pour qui est en **Θ(n),** enfin d'une instructions qui renvoie la distance qui est en **Θ(1),** alors la complexité est : **T = Θ(1) + Θ(n) + Θ(1) = Θ(n)** .

Pour évaluer la performance du programme, on va augmenter le nombre de caractère dans le motA et le motB à chaque test. Pour avoir des meilleurs résultats, les tests ont été réalisés avec une machine branchée sur secteur, et avec le minimum de tâches en cours d'exécution. Nous allons utiliser la fontion **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 le temps avant et après l'exécution de la fonction, puis on soustrait les deux valeurs, et on utilise CLOCKS_PER_SEC pour avoir une valeur en secondes.

Longueur mot	Temps d'exécution
10000	0,000029
20000	0,000061
40000	0,000119
80000	0,000213
160000	0,000397

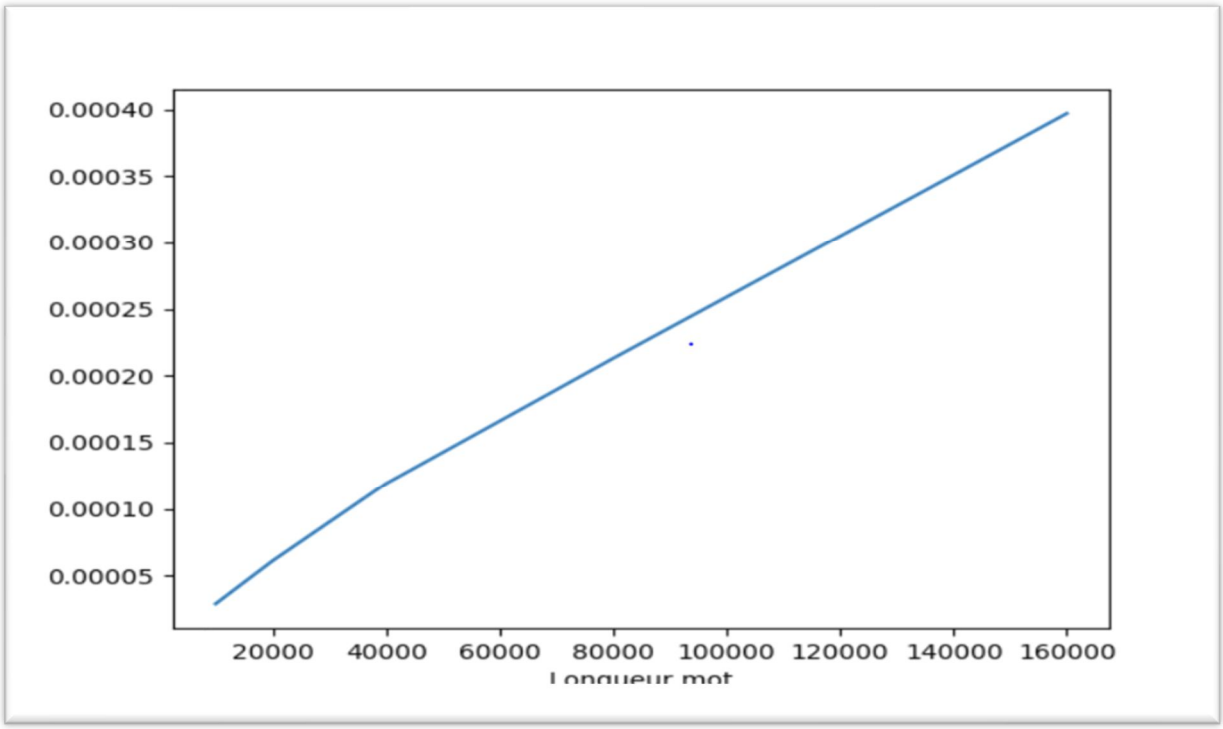


Figure 4 : Temps d'exécution en fonction de la longueur de la chaine de caractères

Après plusieurs testes et en augmentant la longueur des mots, on obtient une courbe (presque une droite). D'après le graphe ci-dessous on constate que la complexité de cet algorithme est linéaire, ce qui confirme le résultat calculé précédemment qui est **Θ(n).**

II. Distance de Levenshtein

1. Présentation de l'algorithme :

L'algorithme de la distance de Levenshtein permet de mesurer la similarité et déférence entre deux mots. Cet algorithme est utilisé par de nombreux correcteurs orthographiques. Il fonctionne avec des mots de longueur différentes, donc il résout la limite de l'algorithme de Hamming qui fonctionne qu'avec des mots de même longueur.

La distance de Levenshtein est le nombre minimal de caractères pour transformer un mot A de longueur **n** à un mot B de longueur **m** en passant par trois opérations : insertion d'un caractère du mot A dans le mot B, et une suppression d'un caractère, et remplacement d'un caractère d'un mot par un autre caractère.

L'algorithme de la distance de Levenshtein retourne un nombre positif ou nul. On dit que deux mots sont identiques si leur distance est nulle. Nous allons présenter deux versions de cet algorithme une version récursive et itérative.

a. Version itérative :

Algorithme itérative distance de Levenshtein:

Entrée :
MotA de longueur (n)
MotB de longueur (m)
La longueur **n**
La longueur **m**
Sortie :
distance [n,m] : matrice entiers
Var :
distance[n+1,m+1] : matrice d'entiers
i,j, remp : entier

```
Début
Pour i = 0 à n faire
        distance[i,0] = i
Fin pour
Pour j = 0 à m faire
        distance[0,j] = j
Fin pour
Pour i = 1 à n faire
        Pour j = 1 à m faire
                Si MotA[i] = MotB[j] alors
                        remp =  0
                Sinon
                        remp = 1
                Finsi
                distance[i,j] = minimum(distance [i-1,j] + 1, distance [i,j-1] + 1, distance [i-1,j-1] + remp)
        FinPour
FinPour
Fin
```

Exemple de déroulement de l'algorithme :

Considérons un mot A de longueur n et un mot B de longueur m

On peut présenter cet algorithme sous forme d'un tableau (matrice) de n+1 lignes et de m+1 colonnes comme suit :

			0	1	2	…	j	…	m-1
		----	B[0]	B[1]	B[2]	….	B[j]	…	B[m]
	----	0	1	2	3	…	J+1	…	m
0	A[0]	1							
1	A[1]	2							
2	A[2]	3							
…	…	…							
i	A[i]	i+1							
…	…	…							
n-1	A[n]	n							Distance[n,m]

On note que la 1ère colonne va correspondre à la suppression des caractères dans le mot A et la 1ère ligne correspond à l'insertion des caractères du mot B dans A. Par exemple la 1ère case, pour passer d'un mot A vide à un mot B vide on insert **0** caractère, pour passer d'un mot A vide à un mot B qui est constitué par un seul caractère, on insert un seul caractère au mot A… etc.

Maintenant que la matrice est initialisée, on va procéder au remplissage des autres cases, pour chaque case **[i,j]** de la matrice, on prend la valeur minimale entre :

- **Distance[i-1][j] + 1**
- **Distance[i][j-1] + 1**
- **Distance[i-1][j-1] +remp** (est égale à 0 si Distance[i] = Distance[j] ou 1 sinon)

En terminant le remplissage de la matrice, la distance entre le mot A et le mot B est stockée dans la **case [n,m].**

On prend le **motA = Maçon** de longueur 5 et on prend le **motB = Maison** de longueur 6 et on va essayer de calculer la distance avec l'algorithme de Levenshtein.

		0	1	2	3	4	5	6
		----	m	a	i	s	o	n
0	---	0	1	2	3	4	5	6
1	m	1						
2	a	2						
3	ç	3						
4	o	4						
5	n	5						

		0	1	2	3	4	5	6
		---	m	a	i	s	o	n
0	---	0	1	2	3	4	5	6
1	m	1	0	1	2	3	4	5
2	a	2	1	0				
3	ç	3						
4	o	4						
5	n	5						

Distance [1,1] : on voit bien que motA[1] = motB[1] = m alors on ne peut pas remplacer ce caractère, donc la valeur remp vaut 0 :

Distance [1,1] = min(Distance[0,1]+1 , Distance[1,0]+1, Distance[0,0]+remp) , remp = 0
Distance [1,1] = min(1+1,1+1,0+0)=0

Distance [1,2] : on voit bien que motA[1] != motB[2] alors on peut remplacer motA[1] par motB[1] caractère, donc la valeur remp vaut 1 :

Distance [1,2] = min(Distance[0,2]+1 , Distance[1,1]+1, Distance[0,1]+remp) , remp = 1
Distance [1,1] = min(1+1,1+1,0+1)=1

On effectue les calculs de la même manière et on obtient le résultat suivant :

		0	1	2	3	4	5	6
		---	m	a	i	s	o	n
0	---	0	1	2	3	4	5	6
1	m	1	0	1	2	3	4	5
2	a	2	1	0	1	2	3	4
3	ç	3	2	1	1	2	3	4
4	o	4	3	2	2	2	2	3
5	n	5	4	3	3	3	3	2

Alors la distance de Levenshtein entre le mot maison et maçon est **2**

On peut vérifier ça :

		Insertion	Substitution		
M	A		Ç	O	N
M	A	I	S	O	N

On constate que pour passer du mot maçon au mot maison on peut faire **une insertion de caractère i** et **une substitution de ç par s,** alors on effectue deux opérations qui égale à la distance de **Levenshtein**

b. **Version Récursive :**

Algorithme Récursive distance de Levenshtein:

Entrée :
MotA de longueur (n)
MotB de longueur (m)
La longueurMotA **n**
La longueurMotB **m**
Sortie :
Distance : entier
Var :
remp: entier

Début
Si longueurMotA = 0 alors
Retourner longueurMotB
Finsi
Si longueurMotB = 0 alors
Retourner longueurMotA
Finsi
Si MotA[longueurMotA - 1**] = MotB[**longueurMotB - 1**] alors**
remp :=0
Sinon
remp :=1
Finsi
retrourner (Algorithme_Récurive_distance_Levenshtein(motA,motB, longueurMotA -1, longueurMotB)+1,
Algorithme_Récurive_distance_Levenshtein(motA,motB, longueurMotA , longueurMotB - 1)+1,
Algorithme_Récurive_distance_Levenshtein(motA,motB, longueurMotA -1, longueurMotB - 1)+1**)**

Fin

L'algorithme récursive de la distance de Levenshtein, prend en entré deux mots, et leurs longueurs n et m respectivement. La première condition vérifier si le mot A est vide alors l'algorithme retourne la longueur de mot B, puisque pour transformer une chaine de caractère vide à un mot B, il faut insérer tous les caractères de mot B, alors la distance est égale à la longueur de mot B. La seconde condition vérifier si le mot B est vide (longueur est égale à 0) alors l'algorithme retourne une distance qui est égale à la longueur de mot A. Ensuite la troisième condition qui vérifier l'équivalence de dernier caractère de deux mots, alors s'il est identique la valeur de la variable remp est **0** si non **1**. Enfin, l'algorithme effectue 3 appels récursifs : la suppression, l'insertion d'un caractère, et le remplacement d'un caractère de mot A par un autre caractère du mot B.

2. Etude de la validité de l'algorithme de la distance de Levenshtein :

a. Une Spécification de la version itérative :

L'algorithme prend en entré deux mots A de longueur n et B de longueur A.

A la fin de l'algorithme itérative de Levenshtein, la valeur de la distance se trouve dans la dernière case du tableau **Distance[n,m],** d'après l'algorithme on peut trouver une spécification :

Distance[n,m] = minimum {
Distance[i-1,j] + 1,
Distance[i,j-1] + 1,
Distance[i-1,j-1] + remp. (remp=1 si motA[n-1] != motB[m-1])

b. Une Spécification de la version récursive :

On sait bien que :

Distance[n,m] = minimum(d[i-1,j] + 1,d[i,j-1] + 1, d[i-1,j-1] + remp) (remp = 1 si motA[n-1] != motB[m-1]

On sait également, que l'algorithme prend en paramètre deux mots A de longueur n et B de longueur m. Dans le cas où les deux mots sont complétement différent alors la distance de Levenshtein retourne le maximum entre les deux longueur de mot A et mot B, on constate alors que la distance est borné par le **maximum(n,m)**

Alors on a : **Distance[n,m] ≤ max(n,m)**

Dans un autre cas, l'algorithme retourne un **0** si les deux mots sont identiques, ou un nombre qui est inférieur **à n et à m** si on les deux mots ont des caractères en commun, alors on peut exprimer la distance de Levenshtein comme suit : **|n – m| ≤ Distance[n,m]**

Alors on obtient : **|n – m| ≤ Distance[n,m] ≤ max(n,m)**

Alors on obtient la spécification suivante :

{
Distance[n,m] = minimum (d[i-1,j] + 1,d[i,j-1] + 1, d[i-1,j-1] + remp)
|n – m| ≤ Distance[n,m] ≤ max(n,m)

c. Invariant de Boucle :

L'algorithme prend deux chaines de caractères **motA** de longueur **n** et **motB** de longueur **m**.

L'algorithme itérative est composé de deux boucles **pour** imbriquées. La 1ère boucle effectue n itérations, et la 2ème boucle effectue m itérations et exécutée **n** fois (donc on a en tout m*n itérations). Pour montrer la validité de cet algorithme, on va trouver un invariant de boucle qui est vrai avant et après une boucle.

Pour la boucle interne, elle effectue **m** itérations (**j de 1 m ➔ 1 ≤ j≤ m+1**)pour chaque incrémentation de la valeur de **i(i de 1 n ➔ 1 ≤ i≤ n+1**), donc **n * m,** on obtient l'équation suivante :

∀x ∈{0. . i− 1} , ∀y ∈ {0. .j – 1} on a distance[x, y] = AlgoLevenshtein(motA[1... x], motB [1… y])

1 ≤ i≤ n+1

1 ≤ j≤ m+1

La première itération : i=1,j=1 alors ∀x ∈, {0. . 1− 1} et ∀y ∈, {0. . 1− 1}, donc **x= 0 et y = 0** (valeurs initiales de la matrice est 0). On a donc **Distance[x, y] = Distance[0,0] = 0**.

La dernière itération : : **i=n+1,j=m+1** alors ∀x ∈{0…n+1− 1} et ∀y ∈, {0…m+1− 1}, donc **x= n et y = m** (la valeur de la distance). On a donc **Distance[x, y] = Distance[n,m]**= résultat de la distance**.**

Concernant la boucle externe, on peut trouver le i est incrémenté chaque fois que la boucle interne est arrivée à m (j=m) donc on obtient :

1 ≤ i≤ n+1

∀x ∈{0. . i− 1},∀y ∈ {0. .m– 1} on a distance[x, y] = AlgoLevenshtein(motA[1... x], motB [1… y])

d. **Organigramme de l'algorithme de la distance de Levenshtein :**

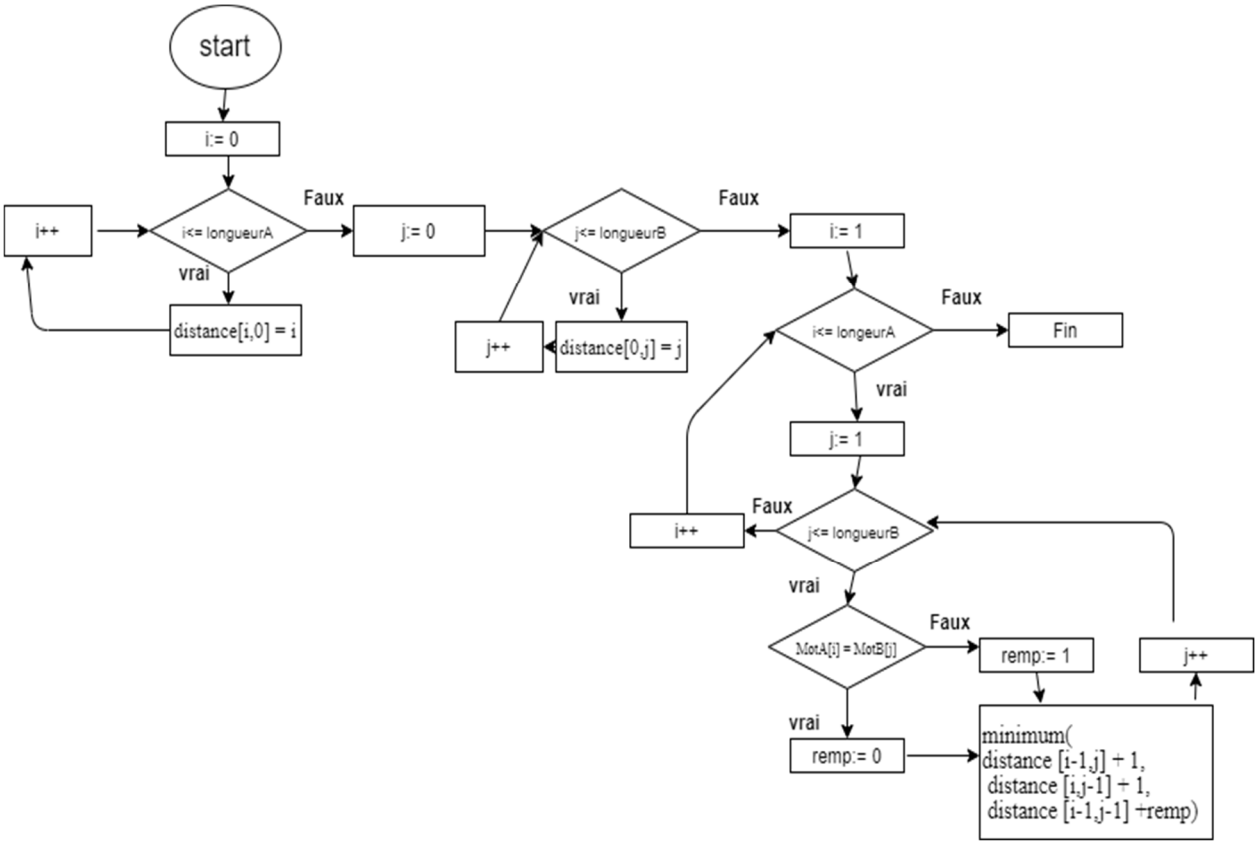


Figure : Organigramme de l'algorithme itérative de Levenshtein

3. Implémentation de l'algorithme de la distance de Levenshtein :

- **Version itérative :**

On va commencer par une fonction qui retourne le minimum de trois valeurs :

```
int minimum( int a,int b, int c)
{
        if(a <= b && a <= c)
        {
                return a;
        }
        else if(b <= a  && b <= c)
        {
                return b;
        }
        else
        {
                return c;
        }
```

Ensuite la version itérative de l'algorithme de Levenshtein :

```
int DistanceLevenshteinIterative(char* motA, char* motB, int longueurA, int longueurB){
	int ** Distance = (int **) malloc (sizeof(int *) * (longueurA +1));
	int i,j,remp,dist;
	// initalisation de la premiere ligne
	for(i = 0; i <= longueurA; i++)
		{
			Distance[i] = (int *) malloc (sizeof(int ) * (longueurB+1));
			Distance[i][0] = i;

		}
	// initalisation de la premiere colonne
	for(j = 0; j <= longueurB; j++)
		{
			Distance[0][j] = j;

		}
	// remplissage de tableau
	for(i = 1; i <= longueurA; i++)
		{
			for(j = 1; j <= longueurB; j++)
				{
					if(motA[i-1] == motB[j-1])
						{
							remp = 0;
						}
					else
						{
							remp =1;
						}
					Distance[i][j] = minimum(Distance[i-1][j]+1,Distance[i][j-1]+1,Distance[i-1][j-1]+remp);
				}
		}
		// on stock la distance de levenshtein (la dernière case du tableau) dans dist
		dist = Distance[longueurA][longueurB];
		// fre
		for(i= 0; i<= longueurA;i++)
			free(Distance[i]);
		free(Distance);
		return dist;

}
```

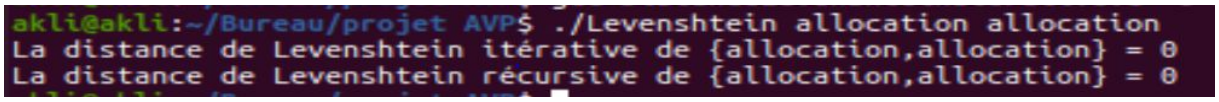
- **Version récursive :**

```
int DistanceLevenshteinRecursive(char* motA, char* motB,int longueurA,int longueurB){
	int a,b,c,remp;
	// si le motA est vide la distance est la taille de motB
	if(longueurA == 0){
		return longueurB;
	}
	// si le motB est vide la distance est la taille de motA
	if(longueurB == 0){
		return longueurA;
	}
	// on vérifier si le dernier caractère des deux mots est identiques
	if(motA[longueurA-1] == motB[longueurB-1])
		{
			remp = 0;
		}
	else
		{
			remp =1;
		}
a =DistanceLevenshteinRecursive(motA,motB,longueurA-1,longueurB)+1; // supression
b =DistanceLevenshteinRecursive(motA,motB,longueurA,longueurB-1)+1; // insertion
c =DistanceLevenshteinRecursive(motA,motB,longueurA-1,longueurB-1)+remp; // remplacement
// on return le min entre a,b,c
return minimum(a,b,c);
}
```

a. Test de l'implémentation :

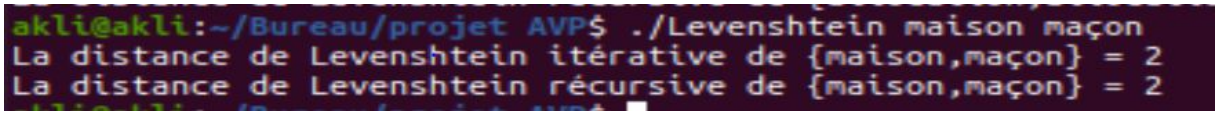
Dans cette partie nous allons faire des tests sur les deux versions de l'algorithme de Levenshtein au même temps**.**

1. Test avec deux mots similaires de même longueur :



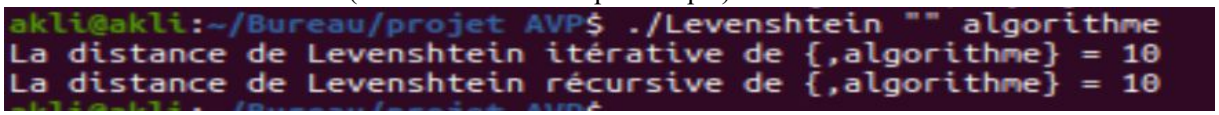
Les deux implémentations nous retournent un **0** puisque on saisit deux mots identiques de même taille.

2. Test avec deux mots d'une longueur déférentes :



Comme dans notre exemple présenté ci-dessus, Les deux implémentations nous retournent une distance de **2.**

3. Test avec deux mots (mot vide et un mot quelconque) :



D'après la spécification, la distance de Levenshtein entre un mot vide et un mot quelconque est égale à la longueur de ce mot qui n'est pas vide. Alors dans notre exemple la longueur de mot algorithme est égale à 10, donc la distance de Levenshtein vaut 10 aussi.

Les deux versions de l'algorithme de Levenshtein produisent les mêmes résultats.
Nous allons procéder dans la partie suivante, l'études de la complexité et les performances des deux versions.

4. Etude de la complexité de l'algorithme de Levenshtein :

a. Version itérative :

Algorithme itérative distance de Levenshtein:

```
Début
Θ(n)   Pour i = 0 à n faire
           distance[i,0] = i                    } Θ(1)
       Fin pour
Θ(m)   Pour j = 0 à m faire
           distance[0,j] = j                    } Θ(1)
       Fin pour
Θ(n)   Pour i = 1 à n faire
Θ(m)       Pour j = 1 à m faire
               Si MotA[i] = MotB[j] alors
                   remp = 0
               Sinon
                   remp = 1
               Finsi
               distance[i,j] = minimum(distance[i-1,j] + 1, distance [i,j-1] + 1, distance [i-1,j-1] + remp)
           FinPour
       FinPour
Fin
```

On considère mot A de longueur n et mot B de longueur m.

L'algorithme commence avec une initialisation de la 1er ligne de la matrice avec n itérations de boucle donc de l'ordre **Θ(n)** et une affection de **Θ(1),** puis une autre boucle pour l'initialisation de la 1er colonne du la matrice avec m itérations de boucle donc de l'ordre **Θ(m)** et une affection de **Θ(1).** Enfin le remplissage de la matrice avec deux boucles imbriquées, la première boucle de 0 à n puis la deuxième boucle de 0 à m, on obtient alors m*n.
Alors la complexité de cet version itérative est :

T = n+1+m+1+m*n ➔ T = (m+1)*(n+1)+1 ➔ T = (m*n)

La complexité de la version itérative de l'algorithme de Levenshtein est de l'ordre de **Θ(m*n)**

Longueur mot	Temps d'exécution
10	0,000016
100	0,000201
500	0,00321
1000	0,0106
2000	0,0412
4000	0,212
8000	1,116

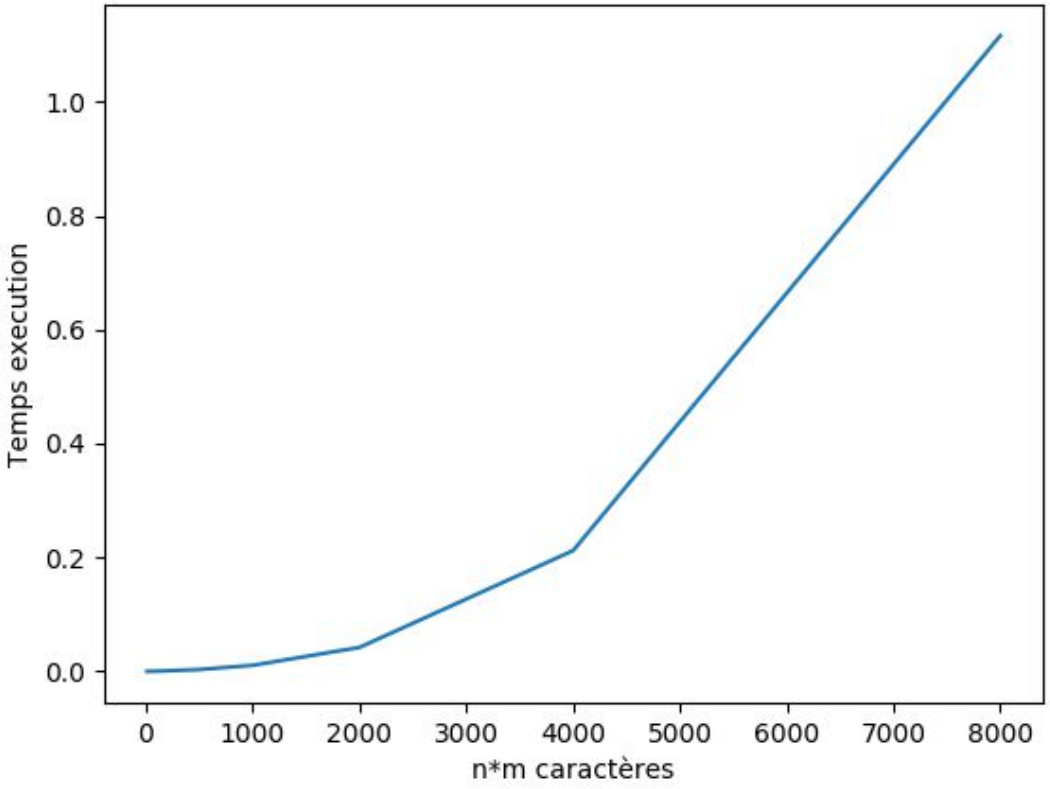


Figure 6 : Temps d'exécution en fonction de la longueur n * m

b. Version récursive :

Algorithme Récursive distance de Levenshtein:

Début

Si longueurMotA = 0 alors
    Retourner longueurMotB (**T = m)**
Finsi
Si longueurMotB = 0 alors
    Retourner longueurMotA (**T = n)**
Finsi
Si MotA[longueurMotA - 1**] = MotB[**longueurMotB - 1**] alors**
    remp :=0
Sinon
    remp :=1
Finsi
retrourner (Algorithme_Récurive_distance_Levenshtein(motA,motB, longueurMotA -1, longueurMotB)+1, T(n-1,m)
    Algorithme_Récurive_distance_Levenshtein(motA,motB, longueurMotA , longueurMotB - 1)+1, , T(n,m-1)
    Algorithme_Récurive_distance_Levenshtein(motA,motB, longueurMotA -1, longueurMotB - 1)+1**)** T(n-1,m-1)

Fin

On considère mot A de longueur n et mot B de longueur m.

L'algorithme commence par vérifier si l'un des mots est vide, si c'est le cas il retourne une distance qui est égale au mot qui n'est pas vide, alors la complexité est **T = n ou T= m**

Sinon si les deux mots sont pas vide on effectue une suppression et une insertion et une substitution, alors la complexité vaut **T = (T(n-1,m) + T(n,m-1) + T(n-1,m-1)+1)**

La complexité de la version récursive est de l'ordre Θ **($3^{min(n,m)}$)**

On constate que la complexité de l'algorithme récursive est plus élevée que la complexité de la version itérative.

Longueur mot	Temps d'exécution
2	0,000002
5	0,000008
10	0,000052
11	0,5
13	10
15	255

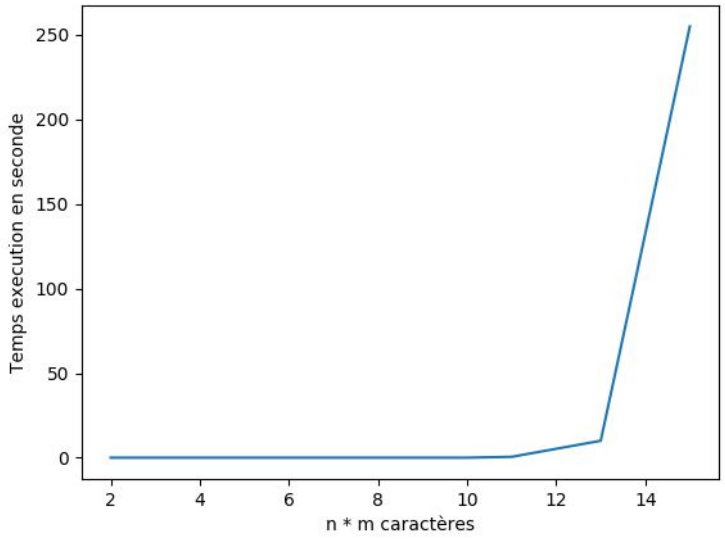


Figure 7 : Temps d'exécution en fonction de la longueur n * m (version récursive)

On constate que la complexité en temps de l'algorithme récursive de Levenshtein est beaucoup plus élevée, on peut dire que cet algorithme récursive n'est pas performant et inefficace.

III. Distance de Jaro :

1. Présentation de l'algorithme

L'algorithme de la distance de Levenshtein permet de mesurer la similarité et déférence entre deux mots. Il est principalement utilisé pour la détection de doublons.

L'algorithme de Jaro normalise le résultat et retourne une valeur entre 0 et 1. Le 0 présente que les deux mots n'ont pas des caractères identiques, et le 1 veut dire que les deux mots sont identiques. Cet algorithme est utilisé particulièrement au traitement des mots de petites tailles.

La distance de Jaro entre un mot A et un mot B est définie par :

$$\textbf{Distance Jaro} = \begin{cases} \textbf{0 si m = 0} \\ \textbf{Sinon} \\ \quad \frac{1}{3}\left(\frac{m}{longueurA} + \frac{m}{longueurB} + \frac{m-t}{m}\right) \end{cases}$$

Où :

m : le nombre de caractère similaire

t : nombre de transposition

Exemple d'application :

On prend le **mot A = crate** de longueur **5** et on prend le **mot B = trace** de longueur **5** et on va essayer de calculer la distance avec l'algorithme de Jaro. On va dresser leur table de correspondance, on met 1 lorsque les caractères sont identiques et 0 sinon

Et on a : éloignement maximum = (max(longueurA,longueurB)/2) -1 = 2-1 = **1**

Sur la 1er ligne on constate qu'on de deux lettres T, mais ils ne correspondent pas car ils sont éloignés de plus de **1** caractères donc on met un 0, sur la deuxième ligne on voit bien qu'il y a une correspondance entre les deux R car la valeur d'éloignement ne dépasse pas 1. Dans la 3 eme ligne, les deux A correspondent car la valeur d'éloignement ne dépasse pas 1. Dans la 4eme ligne les deux C ne correspondent pas car ils sont éloignés avec 3 alors plus de 1 caractères et on voit une correspondance entre les deux E, alors la valeur de **m = 3**

	C	R	A	T	E
T	0	0	0	0	0
R	0	1	0	0	0
A	0	0	1	0	0
C	0	0	0	0	0
E	0	0	0	0	1

m = 3 (le nombre de 1 dans le tableau)

Longueur de mot A = **5**

Longueur de mot B = **5**

t= 0

On obtient une distance de :

$$\frac{1}{3}\left(\frac{3}{5} + \frac{3}{5} + \frac{3-0}{3}\right) = 0.73333$$

Algorithme de la distance de Jaro :

Algorithme distance de Jaro:

Entrée :
MotA de longueur (n)
MotB de longueur (m)
La longueurMotA **n**
La longueurMotB **m**
Sortie :
Distance :reel
Var :
max_eloign,deb,fin **: entier**
corresA,corresB **:tableau de bool**
corres = 0.0,trans =0.0 **: reél**

Début
Si n = 0 et m=0 **alors**
Retourner 1
Sinon n = 0 ou m= 0 **alors**
Retourner 0
max_eloign **=** (**max**(longueurA,longueurB)/2)-1

pour i = 0 **jusqua** longueurA faire
deb = max(0,i - max_eloign **)**
fin = min(i+ max_eloign +1, longueurB)
pour j = deb **jusqua** fin **faire**
si corres**B[j]** rien faire // si motB a déjà une correspondance
si motA[i] != motB[j] rien faire
corresA **[i] = 1**
corresB **[j] = 1**
corres++
Finpour
Finpour
si corres = 0 **Retourner 0.0**

j=0
pour i = 0 jusqua longueurA
si (! corresA **[i]) rien faire // si ya pas de correspondance dans motA**
tantque (! corresB**[j])**
j++
si motA[i] !=motB[j] **alors**
trans++
Fintanque
Finpour
trans = trans /2
retourner (((corres/longueurA)+(corres/longueurB)+(corres – trans))/3)
Fin

2. Implémentation de l'algorithme de Jaro :

a. Implémentation en C :

```c
double DistanceJaro(char* motA, char* motB,int longueurA,int longueurB){
    double correspondance = 0.0;
    double transposition = 0.0;
    /* si les deux mot sont vides le programme retroune 1
    si l'un des mots est vide le programme retourne 0 */
    if (longueurA == 0 && longueurB == 0)
    {
        return 1.0;
    } else if (longueurA == 0 || longueurB == 0){return 0.0;}
    /*on calcule le max eloignement entre deux caractères */
    int max_eloignement = (int) maximum(longueurA, longueurB)/2 - 1;
    /* deux tableau boolean pour savoir si un caractère correspont d'unmot à un autre caractère d'un autre mot */
    int *corresA = calloc(longueurA, sizeof(int));
    int *corresB = calloc(longueurB, sizeof(int));
    /* on cherche les correspondance */
    for (int i = 0; i < longueurA; i++) {
        int deb = maximum(0, i - max_eloignement);
        int fin = minimum(i + max_eloignement + 1, longueurB);
        for (int j = deb; j < fin; j++) {
            if (corresB[j]) continue; /* si motB a deja correspondance */
            if (motA[i] != motB[j]) continue; /* si les deux caractères ne coresspondent pas*/
            corresA[i] = TRUE;
            corresB[j] = TRUE;
            correspondance++;
            break;
        }
    }
    /*si le nombre de correspondance est null en retourne 0*/
    if (correspondance == 0) {
        free(corresA);
        free(corresB);
        return 0.0;
    }
    /*on calcule les transpositions*/
    int j = 0;
    for (int i = 0; i < longueurA; i++) {
        if (!corresA[i]) continue;
        while (!corresB[j]) j++;
        if (motA[i] != motB[j]) transposition++;
        j++;
    }
    /* on devise le nombre transposition par deux */
    transposition = transposition/2.0;
    free(corresA);
    free(corresB);
    /* on retourne la distance de jaro */
    return ((correspondance / longueurA) +(correspondance / longueurB) +((correspondance - transposition) / correspondance)) / 3.0;
```

b. Test de l'implémentation :

1. Test avec deux mots similaires de même longueur :

```
akli@akli:~/Bureau/projet AVP$ ./jaro trace trace
La distance de Jaro de (trace,trace) = 1.000000
```

Le programme nous retourne une valeur égale 1 puisque les deux mots sont identiques.

2. Test avec deux mots différents de même longueur :

```
akli@akli:~/Bureau/projet AVP$ ./jaro crate trace
La distance de Jaro de (crate,trace) = 0.733333
```

Comme dans notre exemple présenté ci-dessus, L'implémentation nous retourne une distance de **0.73333.**

3. Test avec deux mots d'une longueur déférentes :

```
akli@akli:~/Bureau/projet AVP$ ./jaro maison maçon
La distance de Jaro de (maison,maçon) = 0.777778
```

4. Test avec deux mots (deux mots vides) :

```
akli@akli:~/Bureau/projet AVP$ ./jaro "" ""
La distance de Jaro de (,) = 1.000000
```

Le programme nous retourne une valeur égale 1 puisque les deux mots sont vides donc identiques.

5. Test avec deux mots (mot vide et un mot quelconque)

```
akli@akli:~/Bureau/projet AVP$ ./jaro "" algorihthme
La distance de Jaro de (,algorihthme) = 0.000000
```

Le programme nous retourne une valeur égale 0 puisque on trouve aucune correspondance entre un mot vide et le mot « algorithme »

3. Etude de la complexité de l'algorithme de Jaro :

Algorithme distance de Jaro:

```
Début
 Si n = 0 et m=0  alors
        Retourner 1
Sinon n = 0 ou m= 0 alors
        Retourner 0
max_eloign = (max(longueurA,longueurB)/2)-1
corresA[] :tableau de bool
corresB[] :tableau de bool
corres = 0.0
trans = 0.0
pour i = 0 jusqua longueurA faire                          Θ(n)
        deb = max(0,i - max_eloign )
        fin = min(i+ max_eloign +1, longueurB)
        pour j = deb jusqua fin faire                      Θ(m-1)
                si corresB[j] rien faire  // si motB a déjà une correspondance
                si motA[i] != motB[j] rien faire
                corresA [i] = 1
                corresB [j] = 1
                corres++
        Finpour
Finpour
si corres = 0 Retourner 0

j=0
pour i = 0 jusqua longueurA                                Θ(n)
        si (! corresA [i]) rien faire  // si ya pas de correspondance dans motA
        tantque (! corresB[j])                             Θ(m)
                j++
                si motA[i] !=motB[j] alors
                trans++
        Fintanque
Finpour
trans = trans /2
retourner (((corres/longueurA)+(corres/longueurB)+(corres – trans))/3)
Fin
```

On considère mot A de longueur n et mot B de longueur m.

Dans le meilleur de cas si les deux mots sont vide (n=0 et m=0) alors la complexité de l'algorithme de **Jaro** est exécuté en **Θ(1).**

Dans l'algorithme on voit bien qu'on a deux partie, la première partie qui permet de trouver les correspondances entre les deux mots A et B. Cette partie est constitué par deux boucles imbriquée « **pour** », la 1^{ere} boucle est de l'ordre **Θ(n)** et la deuxième boucle de l'ordre **Θ(m),** donc on une complexité sur la **1^{ere}** partie de **Θ(n*m).**

La deuxième partie permet de calculer le nombre de transpositions dans les deux mots. Cette partie est constitué par deux boucles imbriquée (une boucle for et while). La boucle pour est de l'ordre de **Θ(n)** et la boucle **while** dans le pire de cas est de **Θ(m),** Donc on obtient **:**

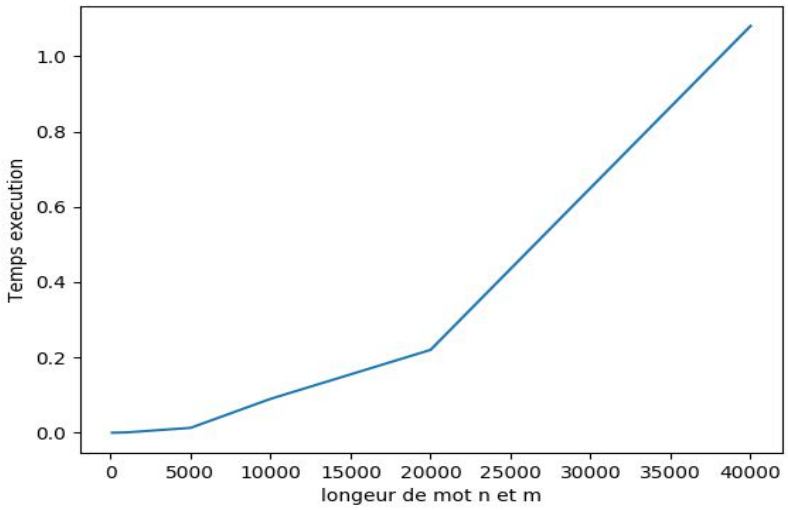
$$T = n*m + m*n \rightarrow T = m(n+n) \rightarrow T = (m*2n) \rightarrow T = (m*n)$$

Alors la complexité de l'algorithme de Jaro est de l'ordre **Θ(n*m).**

Nous allons calculer le temps d'exécution de l'algorithme **Jaro**, on considère que les deux mots de même taille **(m= n)**

Longueur mot	Temps d'exécution
100	0,000012
1000	0,00095
5000	0,013
10000	0,06
20000	0,22
40000	1,12

On obtient la courbe suivante



On constate que l'algorithme de **Jaro** est efficace et performant.

IV. Limites et axes d'améliorations

Dans ce projet, comme nous avons pu le constater, il existe plusieurs algorithmes. La distance de Hamming nous permet parfaitement de calculer la différence entre deux mots donnés, et il est l'algorithme le plus efficace parmi les algorithmes étudiés dans ce rapport d'une complexité **Θ(n) en** temps et en mémoire. Cependant, Cet algorithme fonctionne qu'avec des chaines de caractères de la même taille.

Pour régler ce problème, on a étudié deux autres algorithmes qui calcule la distance de deux chaines de caractères de différentes tailles.

Le premier est l'algorithme de la distance de Levenshtein, avec ces deux versions récursive et itérative. Nous avons constaté que la version récursive et très couteuse, et inefficace puisque des caractères identiques sont calculé plusieurs fois. Donc, il serait mieux d'utiliser la version itérative d'une complexité de **Θ(m*n)** en temps et en mémoire qui est meilleure que la version récursive**.** Le deuxième algorithme est l'algorithme de Jaro qui calcule la similarité de deux chaines de caractères qui retourne une distance comme un taux de similarité qui est borné entre 1 et 0. La complexité de cet algorithme est de **Θ(m*n).** On peut améliorer cet algorithme **avec** la méthode de **Winkler** qui utilise un coefficient de préfixe **P** qui favorise les chaînes commençant par un préfixe de longueur **L<4.** La distance de **Jaro-Winkler** $\mathbf{d_w}$ de deux mots est :

$$d_w = d_j + (L_p(1-d_j))$$

- d_j est la distance de Jaro entre mot A et mot B
- **L** est la longueur du préfixe commun (maximum 4 caractères)
- **P** est un coefficient qui permet de favoriser les chaînes avec un préfixe commun.

On peut aussi calculer la distance entre deux mots utilisant l'algorithme de Jaccard, qui est le rapport entre le cardinal de l'intersection de deux mots et le cardinal de l'union de ces deux mots. On considère mot A et mot B, alors la distance de Jaccard est :

$$DistanceJaccard = \frac{|A \cap B|}{|A \cup B|}$$

Conclusion

Ce projet m'a permis de mettre en pratique les différentes notions de l'analyses des algorithmes, et de comprendre l'importance d'une étude d'un algorithme et tests d'un programme. Ce projet m'a permis d'acquérir des connaissances qui seront utiles pour réaliser des études et des tests sur les projets et les programmes dans mes futures expériences professionnelles ou je serais obligé d'optimiser des programmes pour qu'ils seront plus performants pour gagner en temps et en espace mémoire.

Bibliographie

http://www-igm.univ-mlv.fr/~lecroq/cours/distances.pdf*

http://www.bonne-mesure.com/distance_de_levenshtein.php

http://www.xavierdupre.fr/blog/2013-12-02_nojs.html#:~:text=La%20distance%20d'%C3%A9dition%20(ou,%2C%20plus%20g%C3%A9n%C3%A9ralement%2C%20deux%20s%C3%A9quences.

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

https://rosettacode.org/wiki/Jaro_distance

https://fr.wikipedia.org/wiki/Distance_de_Jaro-Winkler

http://www.cheikh-ibra-fall.fr/a1/s1_cpp/tp7%20vectors,%20matrices/7%20-%20Distance%20de%20Jaro-Winkler.pdf

https://rosettacode.org/wiki/Jaro_distance#C*
https://fr.wikipedia.org/wiki/Indice_et_distance_de_Jaccard