Université Côte d'Azur
Automates & Langages

L3 Info, L3 MI, L3 ST
2023–24

TP n°4

Distance d'édition de Levenshtein

Présentation

La distance d'édition de Levenshtein date de 1965 et mesure la différence entre deux chaînes de caractères. Elle intervient dans les applications qui proposent une correction ou une complétion d'un mot lors de sa saisie, comme les correcteurs orthographiques et les éditeurs en tout genre (texto, mail, moteur de recherche *etc*). La *reconnaissance automatique de la parole* et plus généralement la *reconnaissance des formes* peuvent l'utiliser. Cette mesure a aussi des applications en biologie pour repérer les variations entre différentes séquences d'ADN.

Il existe d'autres distances entre deux chaînes de caractères, on peut mentionner la distance bien connue de Hamming (qui la majore) mais aussi les distances de Damerau-Levenshtein, Jaccard, Jaro, Jaro-Winkler et Stoilos. A noter que la distance de Levenshtein est une *distance* au sens mathématique du terme, elle en vérifie les axiomes. Ainsi, pour toutes chaînes x, y et z, on a :

$$\begin{aligned} d(x,x) &= 0 \\ d(x,y) &= d(y,x) \\ d(x,z) &\leq d(x,y) + d(y,z) \end{aligned}$$

Du point de vue informatique, son calcul est une application intéressante de la *programmation dynamique*. L'idée est que l'on conserve les résultats du problème pour tous les couples de préfixes des chaînes considérées avant de résoudre le problème dans son entier.

Principe

Une distance entre deux chaînes de caractères mesure en fait le nombre d'étapes pour transformer la première chaîne en la seconde. Si on ne s'autorise que des **substitutions** lettre-à-lettre, on peut par exemple transformer le mot `maison` en `raison` puis en `raisin`. On peut se permettre des **insertions**, par exemple en passant de `main` à `matin` ou bien à l'inverse des **suppressions**, par exemple de `piton` à `pion`. On pourrait aussi utiliser d'autres opérations comme des transpositions, des décalages *etc*.

La distance de Levenshtein compte le nombre minimal d'opérations de substitution, d'insertion ou de suppression permettant de passer d'un mot à un autre. Par exemple, en 4 étapes, on peut passer de `rame` à `marin` comme suit :

		r	a	m	e	
1	:	m	a	m	e	
2	:	m	a	r	m	e
3	:	m	a	r	i	e
4	:	m	a	r	i	n

Algorithmiquement, comment résoudre ce problème ? On a recourt à la programmation dynamique : à tout moment dans le mot de départ, on prévoit le coût de chacune des 3 opérations et *a posteriori*, on recherche quels étaient les chemins les moins coûteux. Chaque opération (substitution, insertion, suppression) coûte 1 sauf la substitution quand la lettre substituée serait la même que celle de départ auquel cas le coût devient 0. L'algorithme obtenu a une complexité en $\mathcal{O}(n.m)$ où n et m sont les longueurs respectives de 2 chaînes.

Calcul de la distance

Supposons que les 2 chaînes x et y soient de longueurs respectives n et m. On construit une matrice de dimensions $(n+1) \times (m+1)$ dont les premières ligne et colonne sont initialisées (en rouge ici) comme suit :

		m	a	r	i	n
	0	1	2	3	4	5
r	1					
a	2					
m	3					
e	4					

On remplit de proche en proche selon la formule de récurrence suivante pour i de 0 à n et j de 0 à m :

$$T[i+1, j+1] = \min \begin{cases} T[i,j] + \text{Subst}(x[i], y[j]) & \textit{substitution} \\ T[i, j+1] + 1 & \textit{suppression} \\ T[i+1, j] + 1 & \textit{insertion} \end{cases}$$

avec Subst$(x[i], y[j])$ prenant respectivement la valeur 0 ou 1 selon que $x[i]$ et $y[j]$ coïncident ou pas. Pour tout $1 \leq i \leq n$ et $1 \leq j \leq m$, chaque case (i, j) du tableau va ainsi contenir la distance entre les préfixes $x[1..i]$ et $y[1..j]$.

Exercice 1.

1. Remplissez *à la main* le tableau de l'exemple précédent afin de vous assurer que la méthode de calcul est bien comprise.
2. Ecrivez la fonction `table()` qui prend en entrée les chaînes x et y et qui retourne le tableau des distances. Vous vérifierez ainsi que vous avez répondu correctement à la question précédente.
3. Déduisez-en la fonction `distance()` qui retourne la distance d'édition entre deux chaînes passées en paramètres.

Calcul d'un chemin optimal

A présent que l'on sait calculer la distance d'édition entre deux chaînes de caractères, on cherche à connaître la suite d'opérations à appliquer à la première pour obtenir la seconde. Cela est réalisable en cherchant à rebours, à partir de la case (n, m) *un plus court chemin* vers la case $(0, 0)$.

A priori, il n'y a pas unicité et l'ordre dans lequel seront programmées les comparaisons privilégiera une solution parmi d'autres. Une fois qu'un plus court chemin est obtenu, on peut énumérer dans le bon sens la suite des opérations de substitution (à l'identique ou pas), d'insertion ou de suppression qui ont permis de transformer la chaîne x en la chaîne y.

Exercice 2.

Ecrivez la fonction `chemin()` qui prend en entrée deux chaînes x et y et qui commence par appeler la fonction `table()`. L'algorithme cherche à remonter le plus efficacement possible les calculs depuis la case (n, m) et jusqu'à la case $(0, 0)$. A chaque étape, on mémorisera dans une liste et sous forme de couple laquelle des opérations a permis de passer efficacement de sa prédécesseuse à la case courante (i, j). On adoptera le codage suivant :

substitution :	$(x[i-1], \quad y[j-1])$
insertion :	$(\quad - \quad , \quad y[j-1])$
suppression :	$(x[i-1], \quad - \quad)$

Que la substitution ait lieu à l'identique ou pas, on génère le couple correspondant sachant qu'il réflète plutôt, dans le premier cas, une absence d'opération.
Sur l'exemple précédent, la transformation de `rame` en `marin` est codée par la liste suivante :

```
[('r', 'm'), ('a', 'a'), ('-', 'r'), ('m', 'i'), ('e', 'n')]
```

Enumération de tous les chemins optimaux

L'ordre dans lequel les comparaisons sont faites peut privilégier une solution optimale à une autre. Ici, l'idée est de toutes les énumérer, c'est-à-dire de faire la liste de tous les chemins dont le coût correspond à la distance d'édition.

Exercice 3.

Ecrivez une fonction `tousChemins()` qui cette fois énumére toutes les possibilités de transformation d'une chaîne en une autre dans la mesure où le nombre d'étapes est celui de la distance d'édition entre les deux chaînes. Vous pouvez écrire une sous-fonction `parcours()` dont le rôle est de remonter récursivement toutes les possibilités de solutions efficaces.

Pour l'exemple précédent, on obtient 3 transformations possibles :

```
[('r', 'm'), ('a', 'a'), ('-', 'r'), ('m', 'i'), ('e', 'n')]
[('r', 'm'), ('a', 'a'), ('m', 'r'), ('-', 'i'), ('e', 'n')]
[('r', 'm'), ('a', 'a'), ('m', 'r'), ('e', 'i'), ('-', 'n')]
```