TP1 - Programmation dynamique. Calcul de distances d'édition

Yonatan DELORO

Pour le 21 novembre 2016

Dans ce TP, on cherche à calculer par programmation dynamique les distances d'édition, au sens de Levenshtein et de Damerau-Levenshtein, entre deux chaînes de caractères que l'on notera $s = s_0....s_{m-1}$ et $t = t_0....t_{n-1}$ de tailles respectives m et n.

Pour $u = u_0....u_{p-1}$ une chaîne quelconque de caractères de taille p, on introduit également la notation $u[:i] = u_0...u_i$ $(0 \leq i \leq p-1)$ pour désigner le préfixe de longueur i de u. Enfin on notera $d(u1, u2)$ (resp. $d_M(u1, u2)$) la distance de Levenshtein (resp. Damerau-Levenshtein) entre deux chaînes de caractères $u1$ et $u2$.

1 Distance de Levenshtein

1.1 A quels sous-problèmes se ramener ? (Question 2)

On peut d'abord penser à restreindre le travail à des préfixes des chaînes de caractères s et t.

1.1.1 Renoncer aux sous-problèmes à une variable

Le sous-problème le plus simple auquel on peut penser consiste à évaluer la distance $d(s[:i], t)$ entre un préfixe $s[:i]$ de s et la chaîne t $(0 \leq i < m)$. Toutefois, ces sous-problèmes ne vérifient la propriété de sous-structure optimale. Pour s'en convaincre, on peut tout simplement regarder la distance de s à ses propres préfixes. Pour tout $0 \leq k < m-1$, la distance de $s[:k]$ à s est réalisée en insérant $s_{k+1}, ..., s_{m-1}$. S'il y avait sous-structure optimale, alors la distance de $s[:m-1] = s$ à s serait réalisée, à partir d'un certain sous-problème $0 \leq k < m-1$, en insérant $s_{k+1}, ..., s_{m-1}$, puis en supprimant ces mêmes caractères. Ce qui est évidemment faux $(d(s, s) = 0)$.

Ensuite, on pourrait considérer le sous-problème consistant à évaluer la distance entre un préfixe de s et un préfixe de t, et pour aller au plus simple, de même longueur, c'est-à-dire $d(s[:i], t[:i])$ $(0 \leq i < max(m, n))$, avec la convention $s[:i] = s$ pour $m \leq i$ (de même $t[:i] = t$ pour $n \leq i$) Toutefois, il manque là encore la propriété de sous-structure optimale. Considérons par exemple, avec $'a'$ et $'b'$ deux caractères distincts, les chaînes $s =' a'$ et $t =' ba'$. La distance de $s[:0] = a$ à $t[:0] = b$ est réalisée en substituant a par b. Si la propriété était vraie, la distance de $s[:1] = a$ à $t[:1] = ba$ serait alors réalisée en substituant a par b dans s $(s = b)$ puis en insérant a après b $(s = ba)$. Là encore le chemin est sous-optimal, puisqu'on aurait pu directement insérer b avant a dans s pour obtenir t. Pour le calcul de la distance de Damerau-Levenshtein, on aurait même pu considérer les mots de même longueur $s =' ab'$ et

$t =' ba'$. S'il y avait sous-structure optimale, on aurait obtenu comme chemin optimal de s à t, une substitution de a par b ($s = bb$) puis une autre de b par a ($s = ba$), bien sûr plus long qu'une transposition de a et b.

1.1.2 Sous-structure optimale pour les distances entre préfixes de longueurs indépendantes (2 variables)

Le paragraphe précédent montre qu'il faut renoncer aux sous-problèmes n'utilisant qu'une seule variable. Ainsi, nous allons voir que les sous-problèmes consistant à évaluer la distance de $s[:i]$ à $t[:j]$ pour $0 \leq i < m$ et $0 \leq j < n$ donnent la propriété de sous-structure optimale attendue.

Montrons pour ce faire la relation de récurrence entre sous-problèmes :

$$\begin{cases} d(s[:i], t[:j]) & = & \min\{d(s[:i-1], t[:j]) + 1, \\ & & \quad d(s[:i], t[:j-1]) + 1, \\ & & \quad d(s[:i-1], t[:j-1]) + \mathbb{1}_{s_i = t_j}\} \end{cases} \quad (1)$$

Supposons connues les distances $d(s[:i-1], t[:j])$, $d(s[:i], t[:j-1])$ et $d(s[:i-1], t[:j-1])$. On peut transformer $s[:i]$ en $t[:j]$:

- soit en effectuant les modifications nécessaires sur $s[:i-1]$ pour le transformer en $t[:j]$ (sous-problème 1) puis en supprimant s_i. Ce qui conduirait à une distance $d1 = d(s[:i-1], t[:j]) + 1$.
- soit en effectuant les modifications nécessaires sur $s[:i]$ pour le transformer en $t[:j-1]$ (sous-problème 2) puis en insérant t_j à la fin de $s[:i]$. Ce qui conduirait à une distance $d2 = d(s[:i], t[:j-1]) + 1$.
- soit en effectuant les modifications nécessaires sur $s[:i-1]$ pour le transformer en $t[:j-1]$ (sous-problème 3) puis en effectuant une substitution de s_i par t_j s'ils sont différents, et en n'effectuant rien s'ils sont identiques. Ce qui conduirait à une distance $d3 = d(s[:i-1], t[:j-1]) + 1$ si $s_i \neq t_j$ ou $d3 = d(s[:i-1], t[:j-1])$ si $s_i = t_j$.

.

N'étant autorisées que les trois opérations possibles - insertion, suppression et substitution - de cout fixe, tout chemin d'opérations pour passer $s[:i]$ à $t[:j]$ faisant appel à un autre sous-problème que 1, 2 ou 3 (du type passer de $s[:k]$ à $t[:l]$ avec $k < i-1$ et $l < j-1$) aboutirait à une distance égale à l'une des trois distances candidates $d1, d2$ ou $d3$ (l'un des trois sous-problèmes appellerait cet autre sous-problème)

Ainsi, on a bien montré que $d(s[:i], t[:j]) = min(d1, d2, d3)$.

La procédure décrite ci-dessus montre, par elle-même, que la solution du problème de calcul de distance entre $s = s[:m-1]$ et $t = t[:n-1]$ se construit par récurrence à partir des solutions optimales de sous-problèmes. Par construction, elle présente la propriété de sous-structure optimale.

Aussi, le nombre de sous-problèmes est égal à $m * n$ donc est polynomial.

1.2 Algorithme itératif pour le calcul de distance (Q.3)

Pour calculer la distance $d(s[:m-1], t[:n-1])$ entre les deux chaînes, il faut enfin "instancier" sur les sous-problèmes initiaux. Il suffit d'écrire, en notant $\mathbf{0}$ la chaîne vide, que :

$$\begin{cases} d(\mathbf{0}, t[:j]) & = & j+1 & \forall 0 \leq j < n \\ d(s[:i], \mathbf{0}) & = & i+1 & \forall 0 \leq i < m \\ d(\mathbf{0}, \mathbf{0}) & = & 0 & \end{cases}$$

(Il faut en effet $i+1$ insertions dans la chaîne vide pour aboutir à $t[:j]$)

Ainsi, grâce à la formule de récurrence, on peut remplir itérativement, colonne par colonne, ligne par ligne, la matrice D de taille $(m+1, n+1)$ de termes :

$$\begin{cases} D[0][0] &= d(\mathbf{0}, \mathbf{0}) & \\ D[0][j] &= j & \forall 1 \leq j < n+1 \\ D[i][0] &= i & \forall 1 \leq i < m+1 \\ D[i][j] &= d(s[:i-1], t[:j-1]) & \forall 1 \leq i < m+1 \ \ \forall 1 \leq j < n+1 \end{cases}$$

Pour résumer : $\forall i, j, D[i][j] = d(s[:i-1], t[:j-1])$ avec la convention $s[:-1] = t[:-1] = \mathbf{0}$. Le terme $D[m+1][n+1]$ correspondra alors à la distance de Levenshtein entre s et t.

1.3 Complexités (Q.4)

La complexité en temps de cet algorithme est donc de $O(m*n)$ puisque il y a $n*m$ sous-problèmes et que le coût d'un sous problème est constant. Sa complexité en espace est aussi en $O(m*n)$ puisqu'on alloue de la mémoire à un tableau de taille $m*n$. Nous verrons dans la dernière partie que l'on peut toujours calculer la distance d'édition en complexité linéaire en espace (en $O(min(m,n))$).

1.4 Affichage de la suite des opérations élémentaires de s à t (Q.6)

De plus, la matrice des distances entre préfixes des deux chaînes permet aussi de déterminer un chemin de modifications qui a permis de réaliser la distance d'édition entre les deux chaînes.

Une première solution pour retenir ce chemin serait de mémoriser (dans une deuxième matrice) "au vol", c'est-à-dire simultanément au calcul de $d(s[:i], t[:j])$, la dernière modification retenue sur le chemin permettant de réaliser cette distance.

Toutefois, au lieu de stocker les dernières modifications pour la solution de chacun des sous-problèmes, on peut gagner de l'espace en mémoire en déterminant uniquement les modifications appartenant à un chemin optimal permettant de réaliser $d(s,t) = d(s[:m-1], t[:n-1])$. En effet, l'identification des modifications peut être effectuée a posteriori (après calcul de la matrice des distances), car, par construction de la formule de récurrence, la dernière modification du chemin optimal est directement identifiable à travers la distance (optimale), avec un temps de calcul en $O(1)$.

En effet, si on opère sur s pour atteindre t, il y a stricte équivalence entre :
- $D[i,j] = D[i-1, j-1], s_{i-1} = t_{j-1}$ ET une absence de modification
- $D[i,j] = D[i-1, j-1] + 1, s_{i-1} \neq t_{j-1}$ ET la substitution de s_{i-1} par t_{j-1}
- $D[i,j] = D[i-1, j] + 1$ ET la suppression de s_{i-1}
- $D[i,j] = D[i, j-1] + 1$ ET l'insertion à l'indice $i-1$ de t_{j-1}

Ainsi, on peut remonter le chemin optimal dans la matrice depuis la case donnant la distance entre les deux chaînes (en bas à droite), puisqu'un déplacement horizontal, resp. vertical, resp. diagonal équivaut à une insertion, resp. suppression, resp. substitution ou absence de modification (selon la comparaison de s_{i-1} et t_{j-1}).

CF. FIGURE 1 à la fin du document pour exemple.

2 Distance de Damerau-Levenshtein (Q.7 + Code)

On autorise maintenant les transpositions entre deux caractères successifs.

Les sous-problèmes consistant à évaluer la distance de $s[:i]$ à $t[:j]$ pour $0 \leq i < m; 0 \leq j < n$ donnent encore la propriété de sous-structure optimale attendue.

L'équation de récurrence entre sous-problèmes s'écrit alors, avec d la distance de Levenshtein :

$$\begin{cases} d_M(s[:i], t[:j]) & = & \min\{d(s[:i], t[:j]), d_M(s[:i-2], t[:j-2]) + 1\} & si \quad i > 0, j > 0, s_{i-1} = t_j, s_i = t_{j-1} \\ & = & d(s[:i], t[:j]) & sinon \end{cases} \tag{2}$$

(avec la convention que $s[:-1]$ et $t[:-1]$ correspondent à la chaîne vide)

En effet, en plus des 3 possibilités évoquées pour le calcul de la distance de Levenshtein, on peut aussi, sous réserve que $i > 0, j > 0, s_{i-1} = t_j, s_i = t_{j-1}$, transformer $s[:i]$ en $t[:j]$ en effectuant les modifications nécessaires pour transformer $s[:i-2]$ en $t[:j-2]$ (sous-problème 4) puis en transposant s_{i-1} et s_i. Ce qui conduirait à une distance $d4 = d_M(s[:i-2], t[:j-2]) + 1$.

Le nombre de sous-problèmes est toujours égal à $m * n$ donc est polynomial.

L'implémentation est la même, les complexités temporelles et spatiales sont identiques (coût constant d'une transposition). L'affichage du chemin des opérations se fait sur le même principe.

CF. FIGURE 2 à la fin du document pour exemple.

3 Version linéaire en espace des algorithmes (Q.8 + Code)

Je propose ici une version linéaire en espace qui permet de calculer la distance de Levenshtein (ou de Damerau-Levenshtein) mais qui a la limite de ne pas donner la suite des opérations du chemin optimal.

Pour calculer la distance de Levenshtein entre les deux préfixes $s[:i-1]$ et $t[:j-1]$ à stocker dans la case $D[i][j]$, on a seulement besoin (formule de récurrence) des termes stockés dans la case adjacente à gauche $(i, j-1)$, dans la case adjacente au-dessus $(i-1, j)$, et dans la case dans la diagonale nord-ouest $(i-1, j-1)$. Ainsi, pour calculer une nouvelle ligne i de la matrice D, on peut se contenter de garder en mémoire la ligne $i-1$ juste au-dessus et d'initialiser le terme $D[i, 0]$ à i (i suppressions pour passer de $s[:i-1]$ à la chaîne vide). On peut donc travailler en permanence avec un tableau à deux lignes, la première pour stoker les distances des préfixes de t à la chaîne $s[:i]$, la seconde pour calculer les distances des préfixes de t à la chaîne $s[:i+1]$ que l'on sauvegardera ensuite dans la première ligne.

Dans cet algorithme, on alloue donc de la mémoire à un tableau à 2 lignes et à un nombre de colonnes égal à la taille de t. Sans perte de généralité, en choisissant pour t la chaîne la moins longue, la complexité spatiale est ainsi réduite à $O(min(m, n))$. L'algorithme permet donc de calculer, en complexité spatiale linéaire, la distance de Levenshtein entre s et t, correspondant au terme dans le coin bas-droite de la matrice. Toutefois, cette amélioration a le défaut d'écraser le chemin des opérations permettant de réaliser la distance entre s et t.

Pour obtenir un algorithme linéaire en espace similaire pour le calcul de la distance de Damerau-Levenshtein, il faut travailler avec un tableau à trois lignes, puisque les transpositions obligent à sauvegarder les distances entre préfixes plus courts de deux caractères (voir code).

	0	R	E	T	I	R	E	R
0	0	1	2	3	4	5	6	7
T	1	1	2	2	3	4	5	6
R	2	1	2	3	3	3	4	5
I	3	2	2	3	3	4	4	5
E	4	3	2	3	4	4	4	5
R	5	4	3	3	4	4	5	4

FIGURE 1 – Exemple : Matrice des distances de Levenshtein entre préfixes de "trier" et préfixes de "retirer". Un chemin optimal est désigné par les flèches (en bleu coût de 1, en noir coût nul)

.

	0	R	E	T	I	R	E	R
0	0	1	2	3	4	5	6	7
T	1	1	2	2	3	4	5	6
R	2	1	2	3	3	3	4	5
I	3	2	2	3	3	3	4	5
E	4	3	2	3	4	4	3	4
R	5	4	3	3	4	4	4	3

FIGURE 2 – Exemple : Matrice des distances de Damerau-Levenshtein entre préfixes de "trier" et préfixes de "retirer". Un chemin optimal est désigné par les flèches (en bleu coût de 1, en noir coût nul).

.