Algorithmique
TP : Distance d'édition

Raphaël Ginoulhac

September 21, 2019

1 Calcul de d_{ij}

Soient deux chaînes s et s'. On note d_{ij} la distance d'édition entre le préfixe de taille i de la chaîne s et le préfixe de taille j de la chaîne s'.

Déjà, si j = 0, on a $d_{ij} = i$ vu que c'est la distance d'un mot de taille i au mot vide. De même, si i = 0, on a $d_{ij} = j$.

Il y a 3 cas à examiner pour déterminer d_{ij} :

1. S'il faut effacer un caractère du préfixe de s de taille i, alors la distance est $d_{i-1,j} + 1$
2. S'il faut insérer un caractère, la distance est $d_{i,j-1} + 1$
3. S'il faut substituer un caractère, la distance est $d_{i-1,j-1} + c$ où c est le coût de substitution. Il vaut 0 si $s_i = s'_j$ et 1 sinon.

Finalement, la distance se définit comme :

$$d_{ij} = min(d_{i-1,j} + 1, d_{i,j-1} + 1, d_{i-1,j-1} + c)$$

2 Algorithme récursif

On implémente la fonction récursive sans mémoïsation
void distance_levenshtein_rec(const string &s1, const string &s2) qui se sert de la fonction auxiliaire int distance_levenshtein_rec_aux(const string &s1, const string &s2, int i, int j). Cette dernière renvoie la distance entre le préfixe de taille i de s1 et celui de taille j de s2.

De même, on implémente la fonction récursive avec mémoïsation void distance_levenshtein_rec_memo(const string &s1, const string &s2), qui initialise un tableau d'entiers memo et appelle la fonction auxiliaire qui le remplit à chaque appel si la case (i,j) n'est pas déjà remplie.

3 Complexité

La complexité de la fonction sans mémoïsation est exponentielle en temps et en espace, car on fait 3 appels à chaque appel non terminal (i et j différents de 0). Donc la complexité est un $O(3^N)$ où N est la somme des deux longueurs de chaînes. Celle de la fonction avec mémoïsation est polynomiale en temps et en espace : le tableau est de taille fixée m*n avec m,n les longueurs des chaînes, et on n'effectue un calcul que si la case (i,j) demandée n'est pas déjà remplie. Donc la complexité spatiale (et temporelle) est un $O(m * n)$.

4 Temps d'exécution

Pour un premier cas test avec s1="ecoles" et s2="eclose", les temps d'exécution sont similaires sur ma machine avec 0.001s.

Un deuxième test avec des mots plus long s1 = "testpluslong", s2 = "longmottest" donne un temps de calcul de 1.456s pour la version non mémoïsée contre toujours 0.001s pour la version mémoïsée. On voit bien là l'effet de la complexité exponentielle.

5 Algorithme itératif

On propose l'algorithme suivant :

On initialise un tableau d de taille $(m + 1) * (n + 1)$ (où m et n sont les tailles des chaînes s et s'). La valeur de la ième ligne et de la jème colonne sera la distance d_{ij} . On fait deux boucles imbriquées, sur les lignes puis sur les colonnes, en calculant itérativement d_{ij} avec la formule de la partie 1. On renvoie d_{mn} à la fin de l'algorithme, qui vaut la distance d'édition entre s et s'.

6 Complexité de l'algorithme

Cet algorithme présente une complexité spatiale en $O(m * n)$, à cause de la création du tableau d. Sa complexité temporelle est également $O(m*n)$ à cause des deux boucles imbriquées.

7 Temps d'exécution

La fonction distance_levenshtein affiche la matrice d avec son comportement de base (en appelant affiche_matrice). On l'appelle avec bool affichage=false pour comparer effectivement les temps d'exécution avec les autres fonctions, car sinon on prendrait en compte le temps d'affichage.

Sa vitesse d'exécution sur ma machine est de 0.001s, même pour des longueurs de mots de l'ordre de 30 (et la vitesse d'exécution est la même pour la version récursive mémoïsée).

8 Suite de modifications à effectuer

Pour retrouver la suite de modifications pour passer de s à s', on remonte le tableau d depuis d_{mn}. On prend le voisin de gauche, diagonal gauche ou du haut qui rend la distance minimale. On crée ainsi un chemin dans la matrice d, et en fonction de la direction prise à chaque étape, on peut déterminer quelle action il faut effectuer. Si on choisit l'élément du dessus, cela correspond à un effacement, l'élément de gauche à une insertion et l'élément diagonal gauche à une substitution. On implémente ceci dans affiche_modifications, qui crée une pile, et empile à chaque étape la modification élémentaire à effectuer pour suivre le chemin, puis affiche la suite de modifications.

9 Distance de Damerau-Levenshtein

Pour prendre en compte la possibilité de permuter deux caractères adjacents, on rajoute une opération dans la boucle. Si on peut effectivement permuter les caractères, alors $d_{ij} = d_{i-2,j-2} + c$. Alors,

$$d_{ij} = min(d_{i-1,j} + 1, d_{i,j-1} + 1, d_{i-1,j-1} + c, d_{i-2,j-2} + c)$$

La distance est implémentée dans distance_levenshtein_damerau. On modifie aussi la fonction affiche_modifications pour qu'elle prenne en compte la possibilité d'une transposition.

10 Une version linéaire en espace

En fait, on ne se sert que de la ligne précédente pour calculer la nouvelle ligne de d. Il suffit donc de ne stocker que la ligne précédente et la ligne actuelle, qui représentent $2m$ données, donc une complexité spatiale de $O(m)$. Cette version est implémentée dans distance_levenshtein_lineaire.