Sujet d'étude : distance de Levenshtein

La distance de Levenshtein mesure le degré de similarité entre deux chaînes de caractères. Elle est égale au nombre minimal de caractères qu'il faut supprimer, insérer ou remplacer pour passer d'une chaîne à l'autre. C'est une distance au sens mathématique du terme, donc en particulier c'est un nombre positif ou nul, et deux chaînes sont identiques si et seulement si leur distance est nulle. On a aussi des propriétés de symétrie, et l'*inégalité triangulaire* de la géométrie est ici aussi vérifiée.
Par exemple, pour passer de la chaîne NICHE à la chaîne CHIENS, il faut :

- supprimer les lettres N et I,
- insérer les lettres I, N et S.

donc la distance de Levenshtein entre ces deux chaînes est au plus 5. On peut montrer[1] qu'elle est en fait *exactement* 5. Notons au passage que cette distance ne s'intéresse pas aux déplacements de caractères.
Une des applications de cette distance est la correction orthographique : lorsqu'une personne tape un mot, on le compare à un dictionnaire. Si le mot est présent, on ne fait rien, sinon, on cherche parmi les mots du dictionnaire ceux dont la distance de Levenshtein au mot tapé est inférieure à une limite donnée. Les mots les plus proches sont suggérés comme remplacement en premier.
La mesure de la distance de Levenshtein entre deux chaînes `Chaine1` et `Chaine2`, de longueurs respectives `n1` et `n2`, consiste en la mise en œuvre l'algorithme suivant :

```
fonction DistanceDeLevenshtein(Chaine1,Chaine2 : chaîne) : entier;

// N est un entier représentant la longueur
// de la plus grande chaîne manipulable

variables :
    n1,n2 : entiers      // tailles des deux chaînes
// Le tableau permettant de calculer la distance entre
// les deux chaînes. On en remplira que les n1+1 premières
// lignes, et les n2+1 premières colonnes.
    d     : tableau[0..N,0..N] d'entiers;
    i,j,c : entiers      // c est le coût d'une substitution

n1 <- longueur(Chaine1)
n2 <- longueur(Chaine2)

// initialisation du tableau
pour i de 0 à n1 faire d[i,0] <- i
pour j de 0 à n2 faire d[0,j] <- j

// remplissage du tableau d, ligne par ligne
pour i de 1 à n1 faire
    pour j de 1 à n2 faire
        si Chaine1[i] = Chaine2[i] alors c <- 0 sinon c <- 1
        d[i,j] <- minimum(
                          d[i-1,j  ] + 1  // effacement
                          d[i  ,j-1] + 1  // insertion
                          d[i-1,j-1] + c  // substitution
                         )
retourner d[n1,n2]
```

[1] Les remarques ne garantissent pas qu'on ne peut pas passer de NICHE à CHIENS en faisant moins de 5 opérations.

Pour illustrer le fonctionnement de cet algorithme, observons son déroulement avec `Chaine1='SAMEDI'` et `Chaine2='VENDREDI'`. On commence par initialiser le tableau :

		V	E	N	D	R	E	D	I
	0	1	2	3	4	5	6	7	8
S	1								
A	2								
M	3								
E	4								
D	5								
I	6								

Ensuite on le remplit ligne par ligne, de la gauche vers la droite. Voici deux étapes intermédiaires :

		V	E	N	D	R	E	D	I
	0	1	2	3	4	5	6	7	8
S	1	1	2	*3*	*4*	5	6	7	8
A	2	2	2	*3*	**4**				
M	3								
E	4								
D	5								
I	6								

		V	E	N	D	R	E	D	I
	0	1	2	3	4	5	6	7	8
S	1	1	2	3	4	5	6	7	8
A	2	2	2	3	4	5	6	7	8
M	3	*3*	*3*	3	4	5	6	7	8
E	4	*4*	**3**						
D	5								
I	6								

Dans le tableau de gauche, la case au dessus de celle calculée contient 4, la cas à gauche 3, le minimum est donc 3, auquel on ajoute 1. La case en diagonale contient 3, auquel il faut ajouter 1 parce que la lettre A de SAMEDI n'est pas la lettre D de VENDREDI. Le minimum des deux valeurs est 4, c'est donc ce qu'on inscrit dans cette case.
Dans le tableau de droite, le principe est le même, mais la lettre E étant commune aux deux mots, on n'ajoute rien à la case en diagonale, le minimum est donc 3.
À la fin du remplissage, on obtient le tableau suivant :

		V	E	N	D	R	E	D	I
	0	**1**	**2**	3	4	5	6	7	8
S	1	1	2	**3**	4	5	6	7	8
A	2	2	2	3	**4**	5	6	7	8
M	3	3	3	3	4	**5**	6	7	8
E	4	4	3	4	4	5	**5**	6	7
D	5	5	4	4	4	5	6	**5**	6
I	6	6	5	5	5	5	6	6	**5**

Le chemin en gras indique les opérations à exécuter pour passer du premier mot au second :

- un déplacement horizontal indique une insertion,
- un déplacement vertical indique une suppression,
- un déplacement en diagonal indique une substitution.

Ainsi, la séquence permettant de passer de `SAMEDI` à `VENDREDI` est :

```
SAMEDI -> VSAMEDI -> VESAMEDI -> VENAMEDI -> VENDMEDI -> VENDREDI
```

soit deux insertions et trois substitutions. D'autres solutions sont possibles : il suffit d'aller de case en case en partant de la case inférieure droite, et en allant vers une case réalisant le minimum calculé dans cette case. Dans l'exemple qui nous occupe, peu importe à quel moment on fait les insertions et les substitutions. On aurait ainsi pu faire :

```
SAMEDI -> VAMEDI -> VEMEDI -> VENEDI -> VENDEDI -> VENDREDI
```

Le but de ce sujet d'étude est d'étudier cet algorithme sur quelques exemples, et de réaliser quelques programmes le mettant en œuvre et illustrant son fonctionnement.

Étude de quelques exemples

Pour bien comprendre comment fonctionne cet algorithme, étudions des exemples dans lesquels une seule opération est à réaliser.

1) Construire la matrice issue du calcul de la distance entre `PAPA` et `PARA` (exemple d'une unique **substitution**).

2) Construire la matrice issue du calcul de la distance entre `FORT` et `FORET` (exemple d'une unique **insertion**).

3) Construire la matrice issue du calcul de la distance entre `LEON` et `LEO` (exemple d'une unique **suppression**).

4) Construire la matrice issue du calcul de la distance entre `NICHES` et `CHINE`.

   Déduire de cette matrice tous les moyens de passer d'un mot à l'autre en effectuant à chaque étape une seule opération.

Mise en œuvre de l'algorithme

1) Écrire une fonction Pascal prenant en argument deux chaînes, et calculant la distance de Levenshtein entre elles.

   Insérer cette fonction dans un programme permettant de la tester sur quelques chaînes saisies par l'utilisateur.

2) Améliorer la fonction précédente de manière à lui faire afficher la matrice calculée, ainsi qu'une suite de transformations possibles permettant de passer d'une chaîne à l'autre.

3) Améliorer encore la fonction précédente de manière à lui faire afficher *toutes* les suites de transformations possibles.

Un exemple de correction orthographique

On va étudier sur un exemple le mécanisme de suggestion de mots de substitutions lors d'une correction orthographique.

1) Écrire une fonction prenant en argument une chaîne `c`, un entier `k` et un tableau `t` de `N` chaînes, et renvoyant un tableau `r` de chaînes, ne contenant aucune chaîne si la chaîne `c` est une des chaînes de `t`, et contenant sinon *toutes les chaînes* de `t` dont la distance à `c` est inférieure à `k`.

2) Tester votre fonction à l'aide d'un programme demandant à l'utilisateur de remplir un dictionnaire (le tableau `t`), puis lui demandant une chaîne (la chaîne `c`) et une borne (l'entier `k`), et renvoyant toutes les suggestions issues du dictionnaire si le mot tapé n'est pas dans le dictionnaire.

   On pourra ajouter une option permettant à l'utilisateur d'ajouter le mot tapé au dictionnaire !

Recherches complémentaires

Se documenter sur les sujets connexes aux notions étudiées dans ce sujet d'étude : distance entre deux mots, correction orthographique...

Sources :

- http://en.wikipedia.org/wiki/Levenshtein_distance
- http://jean-paul.davalan.pagesperso-orange.fr/lang/algo/lev/index.html
- http://rosettacode.org/wiki/Levenshtein_distance