CALCUL DE LA PLUS LONGUE SOUS-CHAÎNE COMMUNE

Cademan p242 (7.3).

Pour intro : consulter LIEN ENTRE LA PLUS LONGUE SOUS-CHAÎNE COMMUNE ET LA DISTANCE DE LEVENSHTEIN.

En s'inspirant de ce qu'on a fait pour calculer la distance de Levenshtein on fait un algo de programmation dynamique.

Cependant notez quelques changements :

→ Les trois options INS, DEL et SUB n'ont plus le même statut. On ne réalise une substitution que si elle est triviale, en fait on la réalise soi $x_i = y_j$. Si $x_i \neq y_j$ on départage INS et DEL par un max

→ Un max plutôt qu'un min ? Oui parce que maintenant on stocke la longueur de la PLSCC construite jusque lors.
Se rappeler que maximiser la longueur d'une SSC c'est minimiser le coût d'alignt.

→ L'amélioration consistant à ne retenir que la colonne courante et la précédente pour que l'espace utilisé soit réduit de $O(nk\,m)$ à $O(n)$ est enfin réalisée.

Algo 1

```
PLSCC (x, y)
|  n ← |x|
|  m ← |y|
|  C1 ← tableau indicé de 0 à n init à 0      // colonne "mémoire"
|  C2 ← ————————————————————                  // colonne "courante"
|  Pour j allant de 1 à m
|  |  Pour i allant de 1 à n
|  |  |  Si x_i = y_j
|  |  |  alors C2[i] ← C1[i-1] + 1   // une lettre de plus dans notre SSC
|  |  |  sinon C2[i] ← max(C1[i], C2[i-1])
|  |  C1 ← C2   // la colonne courante est "mise en mémoire".
|  Retourner C2.   // on pourrait a priori retourner C1[n]
                    // mais on aura besoin de C1 entière pour la suite.
```

Complexité

Complexité temporelle en $O(m \times n)$

Complexité spatiale en $O(n)$

ex

x = RAPE $\quad n = 4$

y = LAPINÉ $\quad m = 6$

$j = 1 \quad y_j = L$

0	0	
0	0	= max (0,0) car R ≠ L
0	0	= ——— A ≠ L
0	0	= ——— P ≠ L
0	0	= ——— É ≠ L
0	0	

C_1 C_2

$j = 2 \quad y_j = A$

0	0	
0	0	= max (0,0) car R ≠ A
0	1	= 0+1 car A = A
0	1	= max (0,1) car P ≠ A
0	1	= ——— É ≠ A
0		

C_1 C_2

$j = 3 \quad y_j = P$

0	0	
0	0	= max (0,0) car R ≠ P
1	1	= max (0,1) — A ≠ P
1	2	= 1+1 car P = P
1	2	= max (1,2) car É ≠ P
1		

C_1 C_2

$j = 4 \quad y_j = i$

0	0	
0	0	= max (0,0) car R ≠ i
1	1	= max (0,1) — A ≠ i
2	2	= max (1,2) — P ≠ i
2	2	= max (2,2) — É ≠ i
2		

C_1 C_2

$j = 5 \quad y_j = N$

0	0	
0	0	R ≠ N
1	1	idem car A ≠ N
2	2	P ≠ N
2	2	É ≠ N
2		

C_1 C_2

$j = 6 \quad y_j = É$

0	0	
0	0	= max (0,0) car R ≠ É
1	1	= max (0,1) car A ≠ É
2	2	= max (1,2) car P ≠ É
2	3	= 2+1 car É = É
2		

C_1

→ la plus longue sous chaine est de longueur 3.

(En effet c'est APÉ

RAPE

APE

LAPINÉ)

Comme précédemment on va au delà de la longueur de la PLSCC et on demande une PLSCC.

L'idée est de stocker d'où vient notre max ou si on a fait une substitution triviale dans un tableau qui nous permettra de remonter la solution, l'alignement qui justifie cette longueur de SCC.
C'est la même idée que lorsqu'on veut fournir un alignement minimal justifiant la distance de Levenshtein annoncée.

Mais cet algorithme nécessiterait un espace mémoire en $O(n \times m)$, puisque l'astuce des 2 colonnes n'est plus possible, parce qu'on veut garder tout le tableau en mémoire pour remonter.

Pour palier à cela on utilise le paradigme pour régner : on va chercher à aligner des morceaux plus petits.
Mais ⚠ si on veut couper y en deux au milieu, couper x aussi au milieu n'est pas une bonne idée.
On doit alors calculer là où on doit couper x, c'est ce à quoi sert P dans l'algo ci-dessous.

ex
BAL LON } SCC → BON
BON NET }

mais BAL } SCC → B
BON }
LON } SCC → N
NET }

on n'a pas pu aligner le O !

Algo 2

```
PLSCC_2(x, y)
    n ← longueur(x)
    m ← longueur(y)
    d ← ⌊m/2⌋

    Si n = 0 ou m = 0
    alors retourner ε

    Si n = 1
    alors pour j allant de 1 à m
              si y_j = x_1
              alors retourner x_1
          retourner ε

    Si m = 1
    alors pour i allant de 1 à n
              si x_i = y_1
              alors retourner y_1
          retourner ε

    C_1, C_2, P_1, P_2 ← tableaux indicés de 0 à n
                         initialisés à 0

    C_1 ← PLSCC_1(x, y_1—y_d)

    Pour i allant de 0 à n
        P_1[i] ← i

    Pour j allant de d+1 à m
        Pour i allant de 1 à n
            Si x_i = y_j
            alors C_2[i] ← C_1[i-1]
                  P_2[i] ← P_1[i-1]
            sinon si C_2[i-1] ≥ C_1[i]
                  alors C_2[i] ← C_2[i-1]
                        P_2[i] ← P_2[i-1]
                  sinon C_2[i] ← C_1[i]
                        P_2[i] ← P_1[i]
        C_1 ← C_2 , C_2 ← 0
        P_1 ← P_2 , P_2 ← 0

    k ← P_2[m]   // oublier C_1, C_2, P_1, P_2
    u = PLSCC_2(x_1—x_k, y_1—y_d)
    v = PLSCC_2(x_{k+1}—x_n, y_{d+1}—y_m)
    retourner uv.
```

SUB triviale
DEL
INS

ex

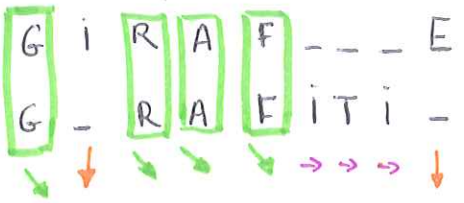
x = GIRAFE $n = 6$

y = GRAFITi $m = 7$

↘ = substitution triviale

↓ = suppression (dans x)

→ = insertion (dans x)



	/	G	R	A	F	i	T	i
/	0	0	0	0	0	0	0	0
G	0	1	1	1	1	1	1	1
i	0	1	1	1	1	2	2	2
R	0	1	2	2	2	2	2	2
A	0	1	2	3	3	3	3	3
F	0	1	2	3	4	4	4	4
E	0	1	2	3	4	4	4	4

R_4 Si on coupait aussi x au milieu on ne pourrait pas aligner le A :

GiR / GRA → G i R _ / G _ R A

AFE / FiTi → A F E _ _ _ / _ F _ i T i

PLSCC_2 (x, y)

$n = 6$; $m = 7$.

$d = \lfloor 3,5 \rfloor = 3$

C_1 = PLSCC_1 (GIRAFE, GRA) =  ; P_1 =  ... C_i =  P_i = 

$k = 4$

u_1 = PLSCC_2 (GIRA, GRA)

$n = 4$ $m = 3$

$d = 1$

C_1 = PLSCC_1 (GIRA, G) =  , P_1 =  ... C_i =  P_i = 

$k = 2$

$u_{1,1}$ = PLSCC (GI, G) = G car $m = 1$ et G apparaît dans GI

$u_{1,2}$ = PLSCC (RA, RA) ... = RA

= GRA

u_2 = PLSCC_2 (FE, FiTi)

$n = 2$, $m = 3$, $d = 1$, C_1 = PLSCC_1 (FE, F) =  , P_1  ... C_i =  , P_i = 

$k = 1$

$u_{2,1}$ = PLSCC (F, FiTi) = F car $m = 1$ et F apparaît dans FiTi

$u_{2,2}$ = PLSCC (E, FiTi) = E ——— En' ——— pas —

= F

= GRAF

Manque complexité !