907

Algorithmique du texte : exemples et applications.

⚠ Regarder le rapport du jury

Soit Σ un alphabet fini.

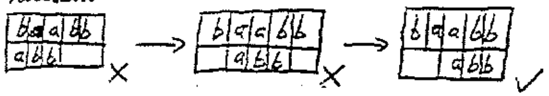
I Recherche d'un motif dans un texte

Problème fréquent dans les traitements de texte, la bio-informatique, l'analyse lexicale...
On note $P \in \Sigma^*$ le motif de longueur m, et $T \in \Sigma^*$ le texte de longueur n.

1) Algorithme naïf - Fenêtres glissantes [Cormen]

```
Pour k ∈ [[1, n-m+1]] :
    Si T[s..s+m-1] = P :
        Affiche "occurence de P en k"
```
} RECHERCHE_NAIVE(P,T)

Exemple : recherche de abb dans baabb



Prop 1 Dans le pire des cas, RECHERCHE_NAIVE a une complexité temporelle en $O(m(n-m+1))$ (comptée en nombre de comparaisons)

2) Algorithme de Rabin-Karp [Cormen]

Les prochains algorithmes tentent d'améliorer la complexité par un prétraitement de P ou T. Celui de Rabin-Karp a une complexité au pire de $O(m(n-m+1))$ aussi, mais il est très utile quand les textes sont courts, et dans la détection de plagiat.

Idée $\Sigma = \{0, 1, .., d\}$. Soit q un entier fixé.

- $W \in \Sigma^*$ est un nombre en base d, noté en minuscule.
  ⇝ Calcul par la méthode de Hörner : pour $w_1 .. w_k$, on calcule $w_k + d(w_{k-1} + d(\dots dw_1))\ [q]$.
- On note t_s le nombre associé à $T[s+1 \dots s+m]$
  $$t_{s+1} = d \cdot (t_s - T[s+1]\, h) + T[s+m-1] \mod q$$
  où $h = d^{m-1}\ [q]$ est calculé à l'avance.
- Dans l'algorithme naïf, on effectue la comparaison "$T[s..s+m-1] = P$" ssi $p = t_{s-1}$.

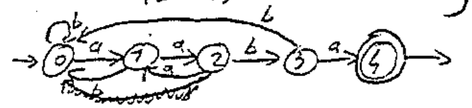
3) Algorithme de Knuth-Morris-Pratt [Cormen]

ⓐ Automate des occurences

Soit $\sigma : \Sigma^* \longrightarrow \{0, 1, .., m\}$ tq $\sigma(x)$ est la longueur du plus long préfixe de P qui soit suffixe de x.

On définit un automate qui reconnait $\Sigma^* P$:
vérifier si P apparait dans T est alors faisable en $O(n)$:
Les états sont $\{0, 1, .., m\}$, où 0 est initial et m final.
La fonction de transition δ vérifie pour un état q et une lettre a :
$$\delta(q, a) = \sigma(P[1..q] + a)$$
} Automate noté $\mathcal{A}_P$

Exemple : Pour aaba



Thm 2 $\mathcal{A}_P$ est l'automate minimal reconnaissant $\Sigma^* P$.

ⓑ Fonction préfixe

On préfère utiliser une table $\pi[1..m]$ tq
$\pi[i]$ est la longueur du plus long préfixe de P qui est suffixe propre de $P[1..q]$

Prop 3 Dans $\mathcal{A}_P$, si $q = m$ ou que $P[q+1] \neq a$, on a
$$\delta(q, a) = \delta(\pi[q], a)$$

On peut donc recalculer $\mathcal{A}_P$ à partir de π.

```
FONCTION_PREFIXE(P) :        // Renvoie le tableau π associé à P.
    π[1] ← 0
    k ← 0
    Pour q ∈ [[2, m]] :
        Tant que k > 0 et P[k+1] ≠ P[q] :
            k ← π[k]
        Si P[k+1] = P[q] alors k ← k+1
        π[q] ← k
    Renvoyer π
```

Exemple : Pour ababca :
$\pi = [0, 0, 1, 2, 0, 1]$

Thm 4 FONCTION_PREFIXE renvoie π en $O(m)$.

Pb : Développements trop similaires

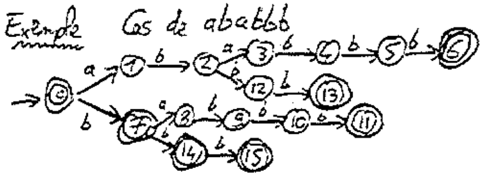
© Algorithme   Complexité temporelle en $O(n+m)$

```
KMP(T,P) :
  π ← FONCTION_PREFIXE(P)
  q ← 0
  Pour i ∈ [[1,n]] :
    Tant que q>0 et P[q+1] ≠ T[i] :   q ← π[q]
    Si : P[q+1] = T[i] :   q ← q+1
    Si : q = m :
      Affiche "occurence de P en i-m"
      q ← π[q]
```

4) Arbre des suffixes   [Crochemore]

Déf 5 L'arbre des suffixes de T, noté $\mathcal{A}(\mathrm{Suff}(T))$, est l'automate déterministe qui reconnait $\mathrm{Suff}(T)$ l'ensemble des suffixes de T, et dans lequel deux chemins de même origine ont toujours des fins distinctes.

Exemple Cas de ababbb

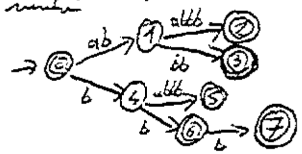


On insère les suffixes $T[i..n]$ dans l'ordre croissant des i.

C'est avec cette convention que les sommets ont été numérotés.

Déf 6 L'arbre compact des suffixes de T s'obtient à partir de l'arbre des suffixes en supprimant les noeuds de degré 1 non terminaux.

Exemple Cas de ababbb



Pour vérifier que P a une occurence dans T, il suffit de parcourir l'arbre compact : s'il existe un chemin étiqueté par P, c'est bon.

La recherche a donc une complexité $O(m)$, c'est le prétraitement qui prend une place immense !

Utilisé quand on a besoin de chercher de nombreux motifs dans T.

Thm 7 (Wiener) On peut calculer l'arbre compact des suffixes de T avec une complexité au pire linéaire en l'espace et en temps.

III Comparaisons de mots

Soit $x \in \Sigma^m$ et $y \in \Sigma^n$.

1) Distance de Levenshtein [Crochemore]   (dvpt)

Le but est d'étudier la façon la plus rapide de changer x en y.
Très utile en bio-informatique et dans les recherches approchées.

On utilise trois opérations d'édition :

- substitution d'une lettre a de x en une lettre b de y (en des positions données) Associée à un coût $\mathrm{Sub}(a,b)$
- suppression d'une lettre de x, notée a, de coût $\mathrm{Del}(a)$.
- insertion d'une lettre de y, notée b, dans x, de coût $\mathrm{Ins}(b)$.

Déf 8 On note $\mathrm{Lev}(x,y) = \min\{\text{coût de } \sigma \mid \sigma \text{ suite d'éditions changeant } x \text{ en } y\}$.
où le coût de σ est la somme des coûts d'opérations utilisées dans σ.

Exemple $x = ACGA$ et $y = ATGCTA$, où $\mathrm{Ins} = \mathrm{Del} = 1$ et $\mathrm{Sub}(a,b) = \begin{cases} 1 & \text{si } a=b \\ 0 & \text{sinon} \end{cases}$

$\mathrm{Lev}(x,y) = 3$, correspondant à un alignement $\begin{pmatrix} A & C & G & - & - & A \\ A & T & G & C & T & A \end{pmatrix}$

Prop 9 Lev est une distance sur Σ^* ssi Sub est une distance sur Σ et $\mathrm{Del}(a) = \mathrm{Ins}(a) > 0$ pour chaque $a \in \Sigma$.

Note : Un alignement de x et y est une façon de visualiser une suite d'éditions changeant x en y. Formellement, c'est un mot z sur l'alphabet $(A \cup \{\varepsilon\}) \times (A \cup \{\varepsilon\}) \setminus \{(\varepsilon,\varepsilon)\}$ dont la projection sur la première composante (resp. seconde composante) est x (resp. y).

Si $z = (\bar{x}_1, \bar{y}_1) \ldots (\bar{x}_p, \bar{y}_p)$, on écrit plutôt z sous la forme $\begin{pmatrix} \bar{x}_1 & \ldots & \bar{x}_p \\ \bar{y}_1 & \ldots & \bar{y}_p \end{pmatrix}$

Le calcul de la distance d'édition se fait par programmation dynamique :

En notant $T[i,j] = \mathrm{Lev}(x[1..i], y[1..j])$,

~~on a la proposition qui suit~~

Analyse lexicale ?
(↳ Pas trop sa place ici ?)

Autres algos de recherche de motif → Boyer Moore
notamment
Arbre des suffixes : pas d'algo

On a pour $i,j \geq 2$: $T[i,j] = \min \begin{cases} T[i-1,j-1] + Sub(x[i], y[j]) \\ T[i-1,j] + Del(x[i]) \\ T[i,j-1] + Ins(y[j]) \end{cases}$

En remarquant que seules deux lignes ou colonnes de T suffisent au calcul, on a :

Prop 10 On peut calculer Lev en un temps $O(mn)$ et en espace $O(\min\{m,n\})$.
Note : Pour exhiber la suite d'édition minimale, il faut y ajouter $O(m+n)$ en temps et en espace.

2) Plus longue sous-suite commune [Crochemore]

En notant $smc(x,y)$ la longueur de la plus longue sous-suite commune à x et y, on voit que ce problème est une spécialisation du précédent, il revient à enlever la substitution des opérations possibles.

Prop 11 Si $Del = Ins = 1$, et que pour $a,b \in \Sigma$ on a $Sub(a,b) > Del(a) + Ins(b) = 2$,
alors $Lev(x,y) = n + m - smc(x,y)$

On peut donc aussi résoudre ce problème par programmation dynamique.
Supposons dans la suite que $m \leq n$.

```
SSC_LONGUEUR(x,y) :    // Renvoie la valeur de smc(x[1...i],y) pour i∈[[1,m]].
  Pour i ∈ {0,1,...,m} : C1[i] ← 0
  Pour j ∈ [[1,n]] :
    C2[0] ← 0
    Pour i ∈ [[1,m]] :
      Si x[i] = y[j] :  C2[i] ← C1[i-1] + 1
      Sinon :           C2[i] ← max{C1[i], C2[i-1]}
    C1 ← C2
  Retourner C1
```

Prop 12 SSC_LONGUEUR nécessite une ~~[illegible]~~ complexité $O(mn)$ en temps et $O(m)$ en espace.

Thm 13 On peut calculer un plus long sous-mot commun à x et y en un temps $O(mn)$ et avec un espace $O(\min\{m,n\})$.

DEVLPT

ne pas parler de Hoffman

III Compression

↝ Utilisé dans les compressions d'image, de sons ou de vidéos. ← Pb : p... de sou...

On cherche à réduire la taille d'un texte en conservant son information.
Présentons ici l'algorithme de Lempel-Ziv-Welch.

On utilise un "dictionnaire" d, i.e. une table qui à chaque entier associe une chaîne de caractères.
Ce dictionnaire est généralement initialisé avec $2k$ cases, où $k = |\Sigma|$, et les k premières cases contiennent les lettres de Σ.

```
LZW(T) :    // T est une chaîne de caractères : un mot de Σ*.
  w = ε
  Pour i ∈ [[1, |T|]] :
    Si w+T[i] ∈ d :        // On a donc une fonction testant si un mot s
      w = w + T[i]         //   est dans le dictionnaire d.
    Sinon :
      Ajouter w+T[i] à d
      Imprimer code(w)     // code(w) est l'entier i tq d[i] = w.
      w = T[i]
  Imprimer code(w)
```

Exemple abab c ba ba ba aaaa aa
Sur un dictionnaire initialisé avec

0	1	2	
a	b	c	ε ...

On aura le code 0, 1, 3, 2, 4, 7, 0, 9, 10, 0.
Et le dictionnaire finira avec

0	1	2	3	4	5	6	7	8	9	10	11
a	b	c	ab	ba	abc	cb	bab	baba	aa	aaa	aaaa

Prop 14 L'algorithme de décompression n'a pas besoin du dictionnaire rempli en entrée.

On peut en avoir l'intuition avec l'exemple : si on veut décoder 0, 1, 3, 2, en sachant que le dictionnaire commence à | a | b | c | ... |, on a $0 \Leftrightarrow a$, $1 \Leftrightarrow b$. On devine donc que $d[3] = ab$. Donc $3 \Leftrightarrow ab$, et $2 \Leftrightarrow c$. On avait abab c avant compression.

Références

[Cormen] Algorithmes, Cormen, Leiserson, Rivest, Stein
[Crochemore] Algorithmique du texte, Crochemore, Hancart, Lecroq
[Beauquier] Éléments d'algorithmique, Beauquier, Berstel, Chrétienne

Plus longue sous-suite commune (PLSSC) [Crochemore]

On utilise l'algorithme SSC_LONGUEUR$(x[0..m-1], y[0..n-1])$, implémentation dynamique de l'idée suivante :

Lemme 1 Si $S[i,j] = \begin{cases} 0 & \text{si } i=-1 \text{ ou } j=-1 \\ smc(x[0..i], y[0..j]) & \text{si } i \in \{0,\dots,m-1\},\ j \in \{0,\dots,n-1\} \end{cases}$,

alors pour $i,j \geq 0$, $S[i,j] = \begin{cases} S[i-1,j-1]+1 & \text{si } x[i]=y[j] \\ \max(S[i-1,j], S[i,j-1]) & \text{sinon} \end{cases}$

Preuve Soient $ua = x[0..i]$ et $vb = y[0..j]$, où $a, b \in \Sigma$.

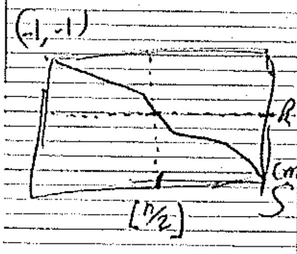
- Si $a=b$, une PLSSC de ua et vb se termine par a ! Autrement, on pourrait la prolonger par a, absurde par maximalité de la longueur.

Donc $S[i,j] = S[i-1,j-1]+1$.

- Si $a \neq b$, et ua et vb possèdent une PLSSC ne se terminant pas par a, on a $S[i,j] = S[i-1,j]$ $(*)$

Par symétrie de a et b, on a $S[i,j] = \max(S[i-1,j], S[i,j-1])$

En reliant (i,j) à $(i-1,j-1)$ si $x[i]=y[j]$,
à $(i-1,j)$ si on est dans le cas $(*)$ du lemme,
à $(i,j-1)$ sinon,
on obtient une ligne joignant $(-1,-1)$ à $(m-1,n-1)$ dans S.



On va chercher la coordonnée k par laquelle passe la ligne dans la colonne $\lfloor n/2 \rfloor$, pour pouvoir appliquer un algorithme SSC, renvoyant une PLSSC, ~~...~~ avec une méthode "diviser pour régner".

k est l'entier maximisant $smc(x[0..k-1], y[0..\lfloor n/2 \rfloor-1]) + smc(x[k..m-1], y[\lfloor n/2 \rfloor..n-1])$

Pour le trouver, on calcule d'abord la colonne d'indice $\lfloor n/2 \rfloor - 1$ de S par SSC_LONGUEUR$(x, y[0..\lfloor n/2 \rfloor-1])$.

Puis on termine la construction de S en mémorisant des pointeurs vers la colonne du milieu dans P_1 et P_2.

Ces deux tables implantent la table P vérifiant

$$\begin{cases} \forall i \in \{-1, 0, \dots, m-1\},\ \forall j \in \{\lfloor n/2 \rfloor - 1, \dots, n-1\}, \\ \qquad P[i,j] = k \in \{0, \dots, i+1\} \text{ si} \\ smc(x[0..i], y[0..j]) = smc(x[0..k-1], y[0..\lfloor n/2 \rfloor - 1]) \\ \qquad + smc(x[k..i], y[\lfloor n/2 \rfloor..j]) \end{cases}$$

Donc $P[m-1, n-1]$ sera le k cherché.

__Lemme 2__

- $\forall i \in \{-1, \dots, m-1\}$, $P[i, \lfloor n/2 \rfloor - 1] = i+1$
- $\forall j \geq \lfloor n/2 \rfloor$, $P[-1, j] = 0$
- Si $i \geq 0$ et $j \geq \lfloor n/2 \rfloor$, $P[i,j] = \begin{cases} P[i-1, j-1] & \text{si } x[i] = y[j] \\ P[i-1, j] & \text{si } x[i] \neq y[j] \text{ et } S[i-1,j] \geq S[i, j-1] \\ P[i, j-1] & \text{sinon} \end{cases}$

__Preuve__ : on se ramène à la définition de P en faisant une récurrence sur (i,j), par ordre lexicographique.

- Si $j = \lfloor n/2 \rfloor - 1$, $y[\lfloor n/2 \rfloor..j] = \varepsilon$, donc $k = i+1$.
- Si $i = -1$, les facteurs de x sont vides donc $k = 0$.
- Si $i \geq 0$ et $j \geq \lfloor n/2 \rfloor$, supposons $x[i] = y[j]$ (les autres cas sont semblables)

(lemme 1) $smc(x[0..i], y[0..j]) = smc(x[0..i-1], y[0..j-1]) + 1$

(récurrence) $smc(x[0..i-1], y[0..j-1]) = smc(x[0..k-1], y[0..\lfloor n/2 \rfloor - 1]) + smc(x[k..i-1], y[\lfloor n/2 \rfloor..j-1])$ où $k = P[i-1, j-1]$

(lemme 1) $smc(x[k..i], y[\lfloor n/2 \rfloor..j]) = smc(x[k..i-1], y[\lfloor n/2 \rfloor..j-1]) + 1$

D'où $P[i,j] = k = P[i-1, j-1]$.

Écrivons enfin l'algorithme SSC.

La preuve de correction découle essentiellement du lemme 2, et de la justification de SSC_LONGUEUR.

$SSC(x,y)$:

Si $m=1$ et $x[0]$ apparait dans y : Renvoyer $x[0]$

Sinon, Si $n=1$ et $y[0]$ apparait dans x : Renvoyer $y[0]$

Sinon, si $m=0$ ou $m=1$ ou $n=0$ ou $n=1$: Renvoyer ε

Sinon

$C_1 \leftarrow$ SSC_LONGUEUR$(x, y[0..\lfloor n/2 \rfloor -1])$

Pour $i \in \{-1, .., m-1\}$: $P_1[i] \leftarrow i+1$

Pour $j \in \{\lfloor n/2 \rfloor, .., n-1\}$:

$C_2[-1], P_2[-1] \leftarrow 0$

Pour $i \in \{0, .., m-1\}$:

Si $x[i] = y[j]$:

$C_2[i] \leftarrow C_1[i-1] + 1$

$P_2[i] \leftarrow P_1[i-1]$

Sinon, si $C_1[i] > C_2[i-1]$:

$C_2[i] \leftarrow C_1[i]$

$P_2[i] \leftarrow P_1[i]$

Sinon

$C_2[i] \leftarrow C_2[i-1]$

$P_2[i] \leftarrow P_2[i-1]$

$C_1 \leftarrow C_2$, $P_1 \leftarrow P_2$

$k \leftarrow P_1[m-1]$

$u \leftarrow SSC(x[0..k-1], y[0..\lfloor n/2 \rfloor -1])$

$v \leftarrow SSC(x[k..m-1], y[\lfloor n/2 \rfloor..n-1])$

Retourner $u \cdot v$

Pour les complexités : SSC s'exécute en temps $O(mn)$ dans un espace $O(m)$.

Preuve : • Si $\mathcal{C}(m,n)$ désigne ~~le nombre~~ la complexité de $SSC(x,y)$ pour $\begin{cases} |x| = m \\ |y| = n \end{cases}$,

alors $\mathcal{C}(m,n) = O(mn) + \mathcal{C}(k, \lfloor n/2 \rfloor) + \mathcal{C}(m-k, n-\lfloor n/2 \rfloor)$.

Ainsi, comme $O(k\lfloor n/2 \rfloor) + O((m-k)(n-\lfloor n/2 \rfloor)) = O(\frac{mn}{2})$ et que $\sum_i \frac{mn}{2^i} \leqslant 2mn$, on a $\mathcal{C}(m,n) = O(mn)$.

• La mémoire nécessaire consiste en la taille de C_1, C_2, P_1 et P_2, le tout en $O(m)$. En effet, comme les appels récursifs n'utilisent pas les informations stockées dans ces tables, on peut réutiliser le même espace dans la suite du calcul.

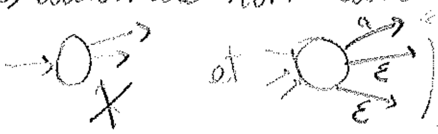
Questions :

1) Recherche de motif : est-ce qu'on peut détecter l'occurrence d'une expression rationnelle ?

↳ OUI → automate

Pb: pour qu'il soit déterministe MAIS : 2^n états ⇝ Bof

↳ Beauquier : méthode linéaire en la taille de l'expr. mais avec des bonnes ppts. (⇝ automate normalisé



← 1 flèche par lettre) pas plus d'ε-transit°

2) Plus courte sur-séquence commune ⇝ un mot le + court possible, qui inclue les deux mots de départ
$(x_1 \dots x_n,\ y_1 \dots y_m)$

AT GCTA
ACGA

PLSSC ⇝ AGA.
$z_1 \dots z_k$

A T G CT A
A C G A
⟶ ATC GCTA
(ou ACTGCTA)

Algo:

```
w ← ε
tant que l < |z|
    tant que x_i ≠ z_l :
        w = w + x_i
        i = i + 1
    tant que y_j ≠ z_l :
        w = w + y_j
        j = j + 1
    w = w + z_l
    l++ ; i++ ; j++
w = w + x[i ... n] + y[j ... m]
retourne w .
```

← DVPT : PLSSC sans optimisat°
+ cet application

DVPT : automate des occurrences et minimal

DVPT : Rabin Karp

Distance de Levenshtein sur les mots

Référence : M. Crochemore, W. Rytter
Text Algorithms

2015-2016

Étant donnés deux mots $X = x_1 \cdots x_m$ et $Y = y_1 \cdots y_n$ sur un alphabet Σ, on cherche à savoir quel est distance de Levenshtein entre X et Y, à savoir le coût minimal pour aller de X à Y en effectuant les opérations élémentaires suivantes :

- suppression d'un caractère de X,
- ajout d'un caractère de X dans Y,
- substitution d'un caractère de X dans Y.

On note m la longueur de X, n celle de Y et $X[i]$ le i-ème caractère de X. On suppose $n \geq m$.

Considérons G, un graphe composé de $m \times n$ nœuds notés (i, j) (pour $i \in [0, m]$ et $j \in [0, n]$). Chaque nœud (i, j) est relié aux nœuds $(i + 1, j)$ et $(i, j + 1)$ par une arrête de poids 1 et au nœud $(i+1, j+1)$ par une arrête de poids $\delta(X[i+1], Y[j+1])$. La distance de Levenshtein entre X et Y est égale au poids du plus court chemin dans G reliant les sommets $(0, 0)$ et (m, n).

Soit $EDIT$ un tableau de taille $m \times n$ défini par :

- $\forall i \in [0, m], EDIT[i, 0] = i$,
- $\forall j \in [0, n], EDIT[0, j] = j$,
- $\forall i \in [1, m], j \in [1, n], EDIT[i, j] = min(EDIT[i - 1, j] + 1, EDIT[i, j - 1] + 1, EDIT[i - 1, j - 1] + \delta(X[i], Y[j]))$.

La distance de Levenshtein entre X et Y est alors égale à $EDIT(m, n)$, la valeur $EDIT(i, j)$ représentant la distance minimale entre $(0, 0)$ et (i, j) dans G.

Theorem 1 *La procédure EDIT a une complexité en temps de $\mathcal{O}(m \times n)$ et en mémoire de $\mathcal{O}(m)$.*

Proof
Dans EDIT, on a deux boucles "pour" imbriquées, et dans chacune de ces boucles des opérations en $\mathcal{O}(1)$. D'où la complexité en $\mathcal{O}(m \times n)$.

On peut toutefois se contenter de garder en mémoire les informations sur la colonne (ou la ligne) actuelle ainsi que la précédente à tout moment du calcul, réduisant ainsi la complexité en mémoire à du $\mathcal{O}(m)$. Ceci dit, garder tout le tableau en mémoire nous permet de reconstruire les opérations pour passer de X à Y. $\square$

Algorithm 1 EDIT

```
m ← X
n ← Y
for i = 1 to m do
    EDIT[i, 0] ← i
end for
for j = 0 to n do
    c[0, j] ← j
end for
for i = 1 to m do
    for j = 1 to n do
        if x_i = y_j then
            delta ← 0
        else
            delta ← 1
        end if
        EDIT[i, j] ← min(EDIT[i − 1, j] + 1, EDIT[i, j − 1] + 1, EDIT[i −
1, j − 1] + delta)
    end for
end for
return EDIT[m, n]
```

Example 1 On cherche à comparer les chaînes NICHE et CHIENS. Le tableau construit par EDIT est :

		C	H	I	E	N	S
	0	1	2	3	4	5	6
N	**1**	1	2	3	4	4	5
I	**2**	2	2	2	3	4	5
C	3	**2**	3	3	3	4	5
H	4	3	**2**	**3**	4	4	5
E	5	4	3	3	**3**	**4**	**5**

La distance entre NICHE et CHIENS est de 5 et une façon de passer de l'un à l'autre est de supprimer les lettres N et I de NICHE pour obtenir CHE puis de rajouter les lettres I, N et S aux endroits adéquats.