**TD02 : Programmation dynamique**

# 1 Distance d'édition ou « distance de Levenshtein »

## 1.1 Présentation du problème

On considère deux mots $M$ et $P$ sur un alphabet donné Σ.
On cherche à minimiser le nombre d'opérations pour passer du mot $M$ au mot $P$, en ne s'autorisant que trois opérations élémentaires suivantes :

- le remplacement d'un caractère
- l'ajout d'un caractère
- la suppression d'un caractère

Par exemple, on peut passer du mot « TARTE » au mot « PARTI » à l'aide de deux remplacements, du mot « DEVOIR » au mot « AVOIR » par un ajout et un remplacement.
La question est de savoir si le nombre d'opérations pour passer d'une chaîne à l'autre est minimal.

- Définition :

On appelle **distance de Levenshtein**, ou encore **distance d'édition**, entre deux mots ce nombre minimal d'opérations pour passer de $M$ à $P$.
Il s'agit d'une distance au sens mathématique sur l'ensemble des mots sur un alphabet Σ.

On notera $d(M,P)$ le nombre minimal d'opérations élémentaires pour passer d'un mot $M$ à un mot $P$.

**Question 1.** En considérant un pire cas, donner un majorant, en fonction de $m$ et $p$, du nombre minimal d'opérations élémentaires, $d(M,P)$, pour passer d'un mot $M$de longueur $m$ à un mot $P$ de longueur $p$.

**Question 2.** En reprenant les notations du *slicing*, et en considérant le devenir de la dernière lettre du mot $M$, justifier que, pour passer d'un mot $M$ à un mot $P$, $d(M,P)$ nécessairement égal à l'une des valeurs suivantes :

$d(M[:-1],P[:-1])$, $d(M[:-1],P[:-1])+1$, $d(M[:-1],P)+1$ ou $d(M,P[:-1])+1$.

## 1.2 Relation de récurrence

Notons $m$ et $p$ les longueurs des mots $M$ et $P$, et $L(i,j)$ la distance de $M[:i]$ à $P[:j]$ pour $(i,j) \in [\![0;m]\!] \times [\![0;p]\!]$.

On donne les relations de récurrence suivantes :

$$(1)\ \forall\, i \in [\![0;m]\!], \qquad L(i,0) = i$$
$$(2)\ \forall\, j \in [\![0;p]\!], \qquad L(0,j) = j$$
$$(3)\ \forall\, (i,j) \in [\![1;m]\!] \times [\![1;p]\!] \quad L(i,j) = \min\big(L(i-1,j)+1, L(i,j-1)+1, L(i-1,j-1)+\delta(i,j)\big)$$

où $\delta(i,j)$ vaut 0 si $M[i-1] = P[j-1]$ et 1 sinon.

(1) traduit que pour passer de façon optimale de $M[:i]$ à une chaîne vide, il en coûte au mieux $i$ suppressions.
(2) traduit que pour passer de façon optimale d'une chaîne vide à $P[:j]$, il en coûte au mieux $j$ ajouts.
(3) traduit que pour passer de façon optimale de $M[:i]$ à $P[:j]$, on peut :

- soit, supprimer la dernière lettre de $M[:i]$, puis passer de façon optimale de $M[:i-1]$ à $P[:j]$ ;
- soit, passer de façon optimale de $M[:i]$ à $P[:j-1]$, puis ajouter la dernière lettre de $P[:j]$ ;
- soit, passer de façon optimale de $M[:i-1]$ à $P[:j-1]$, et alors, soit conserver la lettre en position $i-1$ dans $M$ et en position $j-1$ dans $P$ si elles sont égale (on a alors $\delta(i,j)=0$), soit, sinon, remplacer la lettre en position $i-1$ dans $M$ par la lettre en position $j-1$ dans $P$ (on a alors $\delta(i,j)=1$).

Ces relations de récurrences étant données, on peut construire un algorithme de détermination de la distance entre deux mots :

- soit de façon récursive ;
- soit par programmation dynamique.

## 1.3 Programmation dynamique

Pour deux mots $M$ et $P$ donnés, de longueurs respectives $m$ et $p$, on note $T_{\substack{0\leq i\leq m\\0\leq j\leq p}}$ la matrice définie par

$$T_{i,j} = L(i,j).$$

Pour tous $i \in [\![0,m]\!]$ et $j \in [\![0,p]\!]$, on peut calculer directement les valeurs « bordantes » $T_{i,0} = i$ et $T_{0,j} = j$.
Puis, pour tout $i \in [\![1,m]\!]$ et $j \in [\![1,p]\!]$, on peut calculer $T_{i,j}$ connaissant les valeurs des coefficients $T_{i-1,j}$, $T_{i,j-1}$ et $T_{i-1,j-1}$, *i.e.* les valeurs situées immédiatement à gauche, au-dessus ou au-dessus à gauche, de $T_{i,j}$.

**Question 3.** Construire explicitement la matrice $T$ pour passer des mots « ADA » à « DATA ».

**Question 4.** Définir une fonction `delta(M, P, i, j)` calculant la valeur de $\delta(i,j)$ pour deux mots $M$ et $P$ de longueurs respectives $m$ et $p$ et tout $(i,j) \in [\![0,m-1]\!] \times [\![0,p-1]\!]$.

**Question 5.** Définir une fonction `init(M, P)` ayant pour arguments deux mots $M$ et $P$ et renvoyant, sous la forme d'une liste de listes, la matrice $T$, initialisée avec $T_{i,0} = i$ et $T_{0,j} = j$ tous $i \in [\![0,m]\!]$ et $j \in [\![0,p]\!]$ et tous ses autres coefficients nuls.

**Question 6.** Définir une fonction `mise_a_jour(T, i, j, M, P)`, faisant appel à la fonction delta et mettant à jour le coefficient $T_{i,j}$, en supposant que les coefficients $T_{i-1,j}$, $T_{i,j-1}$ et $T_{i-1,j-1}$ ont déjà été calculés.

**Question 7.** Définir une fonction `matrice_distances(M, P)`, construisant la matrice $T$ pour deux mots $M$ et $P$ et qui en calcule tous les coefficients, puis la renvoie.
On fera appel aux fonctions `init` et `mise_a_jour`.

**Question 8.** Définir une fonction `distance(M, P)`, renvoyant la distance de Levenshtein entre les mots $M$ et $P$.
On fera appel à la fonction `matrice_distances`.

**Question 9.** Définir une fonction `optimum_distance(M, P)`, qui renvoie, sous la forme d'une liste de mots, une succession de transformations de $M$ à $P$ par des opérations élémentaires de suppression, remplacement ou ajout d'un caractère, réalisant la distance de Levenshtein de $M$ à $P$.
Indication : on pourra commencer par définir une fonction `chemin_optimum(M, P)` renvoyant, sous la forme d'une liste de couples $(i,j)$ une succession de cellules correspondant à une transformation de $M$ à $P$ réalisant l'optimum.

# 2 Plus longue sous-séquence commune

Une chaîne de caractères $X = x_1 x_2 \ldots x_{m-1}$ étant donnée, on appelle sous-séquence de $X$ toute chaîne $Z = x_{i_1} x_{i_2} \ldots x_{i_p}$ telle que $0 \leq p < m$ et $1 \leq i_1 < i_2 < \cdots < i_p < m$.
On s'intéresse ici à la recherche d'une plus longue sous-séquence commune entre deux chaînes de caractères $X$ et $Y$.
Ce problème a des applications en bio-informatique où l'on s'intéresse à trouver des similitudes entre des brins d'ADN qui peuvent être modélisés comme des mots sur l'alphabet des quatre nucléotides A, C, G et T.

**Question 10.** Écrire une fonction `gen_sequence(n)` qui génère aléatoirement un mot de longueur $n$ sur l'alphabet $\{'A','C','G','T'\}$.

**Question 11.** Écrire une fonction `sous_sequences(X)`, qui renvoie une liste de toutes les sous-séquences d'une chaîne $X$. En donner une version récursive et une version itérative.
Évaluer la complexité de cette fonction.

**Question 12.** Écrire une fonction `PLSC_rec(X, Y)` renvoyant la plus longue sous-séquence commune à deux chaînes $X$ et $Y$ en faisant appel à la fonction `sous_sequences`.
Évaluer la complexité de cette fonction.

TOURNER LA PAGE

## 2.1 Sous-structure optimale

On note PLSC(X, Y) la plus longue sous-séquence commune à deux chaînes X et Y.
On reprend les notations du *slicing* pour désigner les séquences X et Y privées de leur dernier caractère : X[:-1] et Y[:-1].

**Question 13.** On note $Z$ la plus longue sous-séquence commune à X et Y.

**a.** si les derniers caractères de X et de Y sont les mêmes, que peut-on dire de Z ?

**b.** si les derniers caractères de X et Y sont différents, que peut-on dire de Z si son dernier caractère est le dernier caractère de X ou de Y ?

**c.** Déduire de **a.** et **b.** une décomposition de la recherche d'une PLSC en trois sous-problèmes.

**Question 14.** Justifier que l'on peut ramener le problème de la recherche d'une PLSC au calcul des valeurs $c(i,j)$ pour $(i,j) \in [\![0;n]\!] \times [\![0;m]\!]$ où $c(i,j)$ est la longueur de la plus longue sous-séquence commune entre X[:i] et Y[:j], avec les formules de récurrence suivantes :

$$c(i,j) = \begin{cases} 0 & \text{si } i = 0 \text{ ou } j = 0 \\ c(i-1,j-1)+1 & \text{si } i,j > 0 \text{ et } x_i = y_j \\ \max\big(c(i,j-1), c(i-1,j)\big) & \text{si } i,j > 0 \text{ et } x_i \neq y_j \end{cases}$$

**Question 15.** En déduire un algorithme de résolution par programmation dynamique du problème de la recherche d'une plus longue sous-séquence commune à deux chaînes X et Y, similaire à celle utilisée pour la distance d'édition, utilisant un tableau à deux dimensions.
Évaluer la complexité en temps et en espace de cet algorithme.

**Question 16.** Proposer une implémentation de cet algorithme, inspirée de celle proposée pour la distance d'édition, renvoyant une plus longue sous-séquence commune à deux chaînes X et Y.

# 3 Algorithme de Bellman-Ford

- Source : https://fr.wikipedia.org/wiki/Algorithme_de_Bellman-Ford

L'**algorithme de Bellman-Ford**, aussi appelé **algorithme de Bellman–Ford–Moore**[3], est un algorithme qui calcule des plus courts chemins depuis un sommet source donné dans un graphe orienté pondéré. Il porte le nom de ses inventeurs Richard Bellman et Lester Randolph Ford junior (publications en 1956 et 1958), et de Edward Forrest Moore qui le redécouvrit en 1959.

Contrairement à l'algorithme de Dijkstra, l'algorithme de Bellman-Ford autorise la présence de certains arcs de poids négatif et permet de détecter l'existence d'un circuit absorbant, c'est-à-dire de poids total strictement négatif, accessible depuis le sommet source.

La complexité de l'algorithme est en $O(|S||A|)$ où $|S|$ est le nombre de sommets $|A|$ est le nombre d'arcs.

### Description [ modifier | modifier le code ]

L'algorithme de Bellman-Ford calcule, étant donnés un graphe $G = (S, A)$ sans cycle de poids négatif et un sommet source $s \in S$, un plus court chemin de $s$ à chaque sommet de $G$. Il permet en plus de trouver un tel chemin en retournant les prédécesseurs de chaque sommet dans un tel chemin. En outre, il permet de détecter les cas ou il existe un cycle de poids négatif et donc dans lesquels il n'existe pas nécessairement de plus court chemin entre deux sommets.

L'algorithme utilise le principe de la programmation dynamique (il est traité dans le chapitre portant sur la programmation dynamique dans certains livres d'algorithmique[4]). Les sous-problèmes sont :

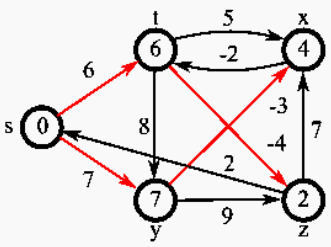
- $d[t,k]$ est la distance du sommet source s à t avec un chemin qui contient au plus k arcs.

On a :

- $d[t,0] = +\infty$ pour $t \neq s$ et $d[s,0] = 0$ ;
- $d[t,k] = min\left[d[t,k-1], min_{\text{arc }(u,t)}(d[u,k-1] + poids(u,t))\right]$ .

L'algorithme calcule les valeurs $d[t,k]$ par valeur de k croissante. La distance de s à t est $d[t,|S|-1]$.

**Algorithme de Bellman-Ford**




| | |
|---|---|
| **Découvreurs ou inventeurs** | Richard Bellman (1958), L. R. Ford, Jr. (1956), Edward F. Moore (1957) |
| **Problèmes liés** | Algorithme de recherche de chemin (d), algorithme de la théorie des graphes (d), concept mathématique (d) |
| **Structure des données** | Graphe |
| **À l'origine de** | Routing Information Protocol[1], Babel[2] |
| **Complexité en temps** | |
| **Pire cas** | $\Theta(\|V\|\|E\|)$ |
| **Meilleur cas** | $\Theta(\|E\|)$ |
| **Complexité en espace** | |
| **Pire cas** | $\Theta(\|V\|)$ |

modifier - modifier le code - modifier Wikidata

**Question 17.** Définir une représentation du graphe en exemple par une matrice $M$ dont chaque coefficient, $m_{i,j}$, est égal au poids de l'arc $i \to j$ si l'arc $i \to j$ existe dans le graphe, où à la valeur « infinie » `float('inf')` si l'arc $i \to j$ n'existe pas.

On considère l'algorithme en pseudo-code donné par la source :

### Pseudo-code [ modifier | modifier le code ]

L'algorithme de Bellman-Ford s'écrit donc en utilisant directement la relation de récurrence donnée dans la section précédente[5].

```
fonction Bellman-Ford(G = (S, A), poids, s)
  pour u dans S faire
  |       d[u] = +∞
  |       pred[u] = null
  d[s] = 0
  //Boucle principale
  pour k = 1 jusqu'à taille(S) - 1 faire
   |      pour chaque arc (u, v) du graphe faire
   |      |    si d[u] + poids(u, v) < d[v] alors
   |      |    |    d[v] := d[u] + poids(u, v)
   |      |    |    pred[v]:= u

  retourner d, pred
```

À l'issue de l'exécution de cet algorithme, $d[u]$ représente la longueur d'un plus court chemin de $s$ à $u$ dans $G$, alors que $pred[u]$ représente le prédécesseur de $u$ dans un plus court chemin de $s$ à $u$. La valeur null signifie qu'aucun prédécesseur n'a pour l'instant été assigné à $pred[u]$.

**Question 18.** Donner une implémentation en Python de l'algorithme sous la forme d'une fonction `bellman-ford(M, s)` prenant en entrée une matrice $M$ décrivant le graphe sous la forme adoptée en question 15., en supposant que les sommets du graphe sont numérotés de 0 à $n-1$ où $n$ est l'ordre du graphe orienté considéré, et un sommet, désigné par son numéro, $s$, qui est utilisé comme sommet-origine, et qui renvoie la liste des plus courts chemins de $s$ à tout sommet $u$ du graphe, sous la forme d'une liste d, telle que $d[u]$ est longueur du plus court chemin du sommet numéroté $s$ au sommet numéroté $u$.

L'algorithme de Bellman-Ford n'est correct que dans un graphe sans cycle de poids négatif. On peut détecter la présence d'un tel cycle (à condition qu'il soit accessible depuis le sommet $s$) de la façon suivante : il y a un cycle de poids négatif si et seulement si un nouveau tour de boucle fait diminuer une distance. Ainsi, à la fin de l'algorithme, on fait :

```
pour chaque arc (u, v) du graphe faire
 |    si d[v] > d[u] + poids(u, v) alors
 |         afficher "il existe un cycle absorbant"
```

**Question 19.** Compléter votre implémentation en conséquence.

On donne la complexité de cet algorithme

La complexité de l'algorithme est en $O(|S||A|)$ où $|S|$ est le nombre de sommets et $|A|$ est le nombre d'arcs. Cela correspond à une complexité en $O(|S|^3)$ pour un graphe simple dense[5].

**Question 20.** Justifier la complexité annoncée à partir de votre implémentation.

On donne à titre d'information la preuve de correction de l'algorithme :

La démonstration de la correction de l'algorithme peut se faire par récurrence :

**Lemme.** Après $i$ répétitions de la boucle principale :

- Si $d[u] \neq +\infty$, alors $d[u]$ est le poids d'un chemin de $s$ à $u$ ;
- S'il existe un chemin de $s$ à $u$ d'au plus $i$ arêtes, alors $d[u]$ est au plus la longueur du plus court chemin de $s$ à $u$ comprenant au plus $i$ arêtes.

**Preuve.** Pour le cas $i = 0$, correspondant à la première exécution de la boucle `for`, alors, d'une part, $d[s] = 0$ et, pour tout sommet $u \neq s$, $d[u] = +\infty$, prouvant le premier point et, d'autre part, $s$ est le seul sommet tel qu'il existe un chemin d'au plus 0 arêtes le reliant à $s$.

Pour le cas $i$ non nul quelconque, alors :

- D'une part, supposons $d[v]$ soit mis à jour avec $d[v] := d[u] + \text{poids}(u, v)$. Alors, comme $d[u] \neq +\infty$ (car sinon, la mise à jour n'aurait pas lieu), $d[u]$ représente le poids d'un chemin de $s$ à $u$. Donc, par construction, $d[v]$ est le poids d'un chemin de $s$ à $v$ ;
- D'autre part, soit $C$, un plus court chemin de $s$ à $v$ d'au plus $i$ arêtes. Soit $u$ le dernier sommet avant $v$ sur ce chemin. Soit $C'$, le sous-chemin de $C$ allant de $s$ à $u$. Alors, par induction, $C'$ est un plus court chemin de $s$ à $u$ d'au plus $i-1$ arêtes (en effet, sinon, il existe un chemin strictement plus court de $s$ à $u$. En y ajoutant l'arête de $u$ à $v$, on trouve un chemin strictement plus court que $C$ allant de $s$ à $v$, ce qui est contradictoire avec la définition de $C$). Alors, par hypothèse de récurrence, à l'issue de l'itération $i-1$, $d[u]$ était au plus la longueur d'un plus court chemin de $s$ à $u$ d'au plus $i-1$ arêtes. Ainsi, à la $i$-ème itération, $d[v] \leq d[u] + \text{poids}(u, v)$, en raison de la structure de l'algorithme. Or, cette dernière longueur est au plus la longueur d'un plus court chemin de $s$ à $v$.

Le lemme est donc démontré. Il s'ensuit que la longueur $d[u]$ est la longueur d'un plus court chemin de $s$ à $u$, ce qui prouve la correction de l'algorithme de Bellman-Ford dans le cas où $G$ n'a pas de cycle de poids négatif[5].