TP 4 : Programmation dynamique

1 Chemins dans un tableau

Le but du cette partie du TP est de résoudre le problème suivant. On se donne un tableau contenant des nombres positifs. On souhaite le parcourir du coin en haut à gauche au coin en bas à droite par un chemin de coût minimal. Le coût d'un chemin est la somme des nombres figurant sur les cases traversées. On impose que les déplacements ne puissent se faire que d'un case vers la droite ou d'une case vers le bas.
Résoudre tout d'abord ce problème à la main sur le tableau suivant :

1	2	3	3
1	1	4	4
1	1	1	4
3	2	1	2

Afin de représenter ces tableaux, on les affichera en niveau de gris (en supposant qu'ils contiennent des entiers compris entre 0 et 255. Afin d'effectuer cet affichage, on utilisera des tableaux numpy pour représenter ces tableaux. Par exemple pour le précédent :

```
import matplotlib.pyplot as plt
import numpy as np

M1 = np.array([[1,2,3,3],
               [1,1,4,4],
               [1,1,1,4],
               [3,2,1,2]])
plt.imshow(M1,cmap='gray')
```

Les fonctions seront tout d'abord testées sur ce premier tableau. Ensuite, on créera un tableau (de plus grande taille) contenant des entiers aléatoires compris entre 0 et 255, par la commande :

```
import random as rd

n , p = 100, 100
M = np.zeros((n,p),dtype = np.uint8)
for i in range(n):
    for j in range(p):
        M[i,j] = rd.randint(0,255)
```

Le but sera alors d'afficher ce tableau en niveaux de gris et de représenter en noir le chemin de coût minimal.
Les fonctions à compléter se trouvent dans le fichier `progDynTPTableau_ACompleter.py`.

Analyse du problème

Nous allons résoudre ce problème par une méthode de programmation dynamique.

1. Mettre en évidence la façon dont le problème de parcourt de coût minimal jusqu'à la case (i,j) peut être résolu à partir de sous-problèmes.
2. Écrire les équations de Bellman associées au problème.
3. Identifier les risques de chevauchement des sous-problèmes.

Résolution par une stratégie de bas en haut

Résoudre ce problème par un algorithme itératif en partant de la case en haut à gauche du tableau. On propagera le coût minimal dans un tableau en prenant bien garde à l'ordre des cases pour cette propagation (on n'hésitera pas à faire un schéma).
On écrira pour cela une fonction `cheminOptimalIter(M)` prenant en argument un tableau numpy `M` et renvoyant le chemin optimal ainsi que sont coût (on pourra dans un premier temps écrire une version `coutOptimalIterV1(M)` de cette fonction qui ne renvoie que le coût optimal).
Une trame pour chacune de ces fonctions se trouve dans le fichier à compléter.

Résolution par mémoïsation

1. Dans un premier temps, on va résoudre le problème de coût sans se soucier de retrouver le chemin optimal. On pourra écrire une fonction commençant ainsi :

```
def coutOptimal(M):
    n , p = np.shape(M)
    C = np.zeros((n,p))
    for i in range(n):
        for j in range(p):
            C[i,j] = -1   # -1 signale une case non visit\'ee

    def remplissage(i,j):
        ...
    ...
```

   La fonction `remplissage` est une fonction récursive interne dont le but est de mettre dans chaque case du tableau `C` le coût minimal pour y arriver.

2. À partir de la fonction précédente, écrire une fonction `cheminOptimal(M)` qui en plus de trouver le coût optimal détermine le chemin optimal. Pour cela, on stockera dans la case i,j le coût minimal du chemin pour y parvenir mais également le sommet la précédant sur ce chemin. Remarquons que pour faire cela, on ne peut plus utiliser de tableau numpy pour `C` ; on prendra une liste de listes (chaque case i,j contenant à la fin une liste constituée d'un nombre (le coût minimal) et d'un couple (représentant la case précédente sur le chemin optimal)).
3. Utiliser la fonction précédente afin de tracer le graphique dont il est question au début du TP (chemin optimal dans un tableau contenant des entiers aléatoires).
4. On pourra dans une deuxième temps réécrire la fonction précédente en utilisant un dictionnaire pour stocker les données (même si cela n'a pas vraiment d'avantage ici).

2 Rendu de monnaie

Le but de cette partie du TP est de revoir la notion d'algorithme glouton sur le problème de rendu de monnaie, puis de voir que l'on peut résoudre ce même problème par une méthode de programmation dynamique.
On rappelle ici la nature du problème. On souhaite rendre une certaine somme S de monnaie (on supposera que S est un entier) en utilisant des pièces (qui peuvent être des billets en pratique) contenus dans une liste (dans un premier temps 1, 2, 5, 10, 20, 50, 100, 200, 500), et ce en utilisant un nombre minimum de pièces.
On supposera dans ce TP que la liste de pièce contient systématiquement une pièce de valeur 1 et que S est un entier ; de sorte que le problème a toujours une solution.

2.1 Algorithme glouton

Le principe de cet algorithme est le suivant. On construit la somme S en prenant tout d'abord un maximum de pièces (billets) de 500, on obtient une nouvelle somme à rendre ; on diminue au maximum cette nouvelle somme en utilisant des pièces de 200 ...
Remarquons que le fait que nous avons supposé S entière et qu'il existe une pièce de valeur 1 entraîne l'existence d'une solution.

1. Écrire une fonction `renduMonnaieGlouton(listePieces,S)` qui renvoie la liste des pièces à rendre (le i-ème terme de cette liste correspondant au nombre de pièces numéro i).
   Par exemple, appliquée à la liste de pièces `[1,2,5,10,20,50,100,200,500]` avec la somme 345, cette fonction doit renvoyer `[0,0,1,0,2,0,1,1,0]`.

2. Que donne cette fonction avec la liste de pièces `[1,4,6]` avec la somme 8 ?
   Est-ce la solution optimale ?

2.2 Programmation dynamique

On peut résoudre ce problème en utilisant la résolution de sous-problème.
Par exemple, si l'on sait résoudre ce problème avec la liste `[1,2,5,10,20,50,100,200]` pour n'importe quelle somme S, il suffira, pour résoudre le problème avec la liste `[1,2,5,10,20,50,100,200,500]` et une somme S donnée :

- de calculer, pour chaque valeur de k telle que $S - 500k \geq 0$, la solution optimale du problème n_k (nombre minimal de pièces) pour la valeur $S - 500k$ avec la liste de pièces `[1,2,5,10,20,50,100,200]` ;
- puis de prendre la valeur minimale parmi les $n_k + k$.

Programmer cette méthode par mémoïsation (on utilisera un dictionnaire). On pourra tout d'abord écrire une fonction `renduMonnaieProgDynMemo(listePieces,SommeTotale)` qui renvoie juste le nombre de pièces de monnaie nécessaire (et non pas la liste).

3 Distance d'édition

3.1 Description du problème

Définition de la distance

La distance de Levenshtein entre deux mots (chaînes de caractères) $a_1 \dots a_n$ et $b_1 \dots b_m$, que nous noterons $d(a_1 \dots a_n, b_1 \dots b_m)$ est définie comme le nombre d'opérations nécessaires pour passer de $a_1 \dots a_n$ à $b_1 \dots b_m$, les opérations autorisées étant :

- suppression d'une lettre dans une des deux chaînes ;
- insertion d'une lettre ;
- changement d'un caractère par un autre.

Cette distance est également appelée distance d'édition.
Le but de ce TP est de programmer un algorithme permettant de déterminer cette distance et de déterminer les opérations à effectuer.

Équations de récurrence

On va établir des équations permettant de calculer la distance entre deux mots à partir des distances entre des mots comprenant les mêmes lettres à l'exception de la dernière (en un sens à préciser).
Avant d'établir ces équations de récurrence, on a besoin d'initialiser. Pour cela, on utilise la remarque suivante : si "" désiqne la chaîne vide, on a $d(a_1 \dots a_n, "") = n$.
On montre par ailleurs que :

- si $a_n = b_m$, alors
$$d(a_1 \dots a_n, b_1 \dots b_m) = d(a_1 \dots a_{n-1}, b_1 \dots b_{m-1}) \ ;$$
- que si $a_n \neq b_m$,

$$d(a_1 \dots a_n, b_1 \dots b_m) \leq 1+\min\left(d(a_1 \dots a_{n-1}, b_1 \dots b_{m-1}), d(a_1 \dots a_n, b_1 \dots b_{m-1}), d(a_1 \dots a_{n-1}, b_1 \dots b_m)\right.$$

Explications

On peut dans un premier temps admettre les relations de récurrence et laisser de côté ces explications.
L'initialisation est claire.
Il est clair également que si $a_n = b_m$, alors $d(a_1 \dots a_n, b_1 \dots b_m) \leq d(a_1 \dots a_{n-1}, b_1 \dots b_{m-1})$, car si l'on sait passer de $a_1 \dots a_{n-1}$ à $b_1 \dots b_{m-1}$ en d étapes, alors on sait passer de $a_1 \dots a_n$ à $b_1 \dots b_m$ en d étapes (en gardant la dernière lettre inchangée).
Supposons maintenant $a_n \neq b_m$. Si l'on sait passer de $a_1 \dots a_{n-1}$ à $b_1 \dots b_m$ en d étapes, alors on sait passer de $a_1 \dots a_n$ à $b_1 \dots b_m$ en $d+1$ étapes (l'étape supplémentaire étant l'ajout de a_n). On a donc $d(a_1 \dots a_n, b_1 \dots b_m) \leq d(a_1 \dots a_{n-1}, b_1 \dots b_m) + 1$. De même on montre que $d(a_1 \dots a_n, b_1 \dots b_m) \leq d(a_1 \dots a_{n-1}, b_1 \dots b_m)+1$ et $d(a_1 \dots a_n, b_1 \dots b_m) \leq d(a_1 \dots a_{n-1}, b_1 \dots b_{m-1})+1$.
On en déduit :

$$d(a_1 \dots a_n, b_1 \dots b_m) \leq 1+\min\left(d(a_1 \dots a_{n-1}, b_1 \dots b_{m-1}), d(a_1 \dots a_n, b_1 \dots b_{m-1}), d(a_1 \dots a_{n-1}, b_1 \dots b_m)\right) \ .$$

Il n'est pas tout à fait clair que ces inégalités sont des égalités. En effet, le plus court chemin pour passer de $a_1 \dots a_n$ à $b_1 \dots b_m$ ne passe peut-être pas par la suppression de la dernière lettre d'un des deux mots ou leur substitution.
Une façon un peu différente de présenter les choses va montrer que l'on peut supposer qu'il en est ainsi.
Prenons l'exemple des mots CHIENS et NICHER. On passe de l'un à l'autre avec les opérations suivantes : insérer N (NCHIENS), insérer I (NICHIENS), enlever I (NICHENS), enlever N (NICHES), substituer R à S (NICHER). On montre que l'on ne peut pas faire mieux.
On peut visualiser cela ainsi :

-	-	C	H	I	E	N	S
N	I	C	H	-	E	-	R

On lit les opérations décrites auparavant en passant de la première à la deuxième ligne.
remarquons que le tableau suivant aurait convenu :

-	-	C	H	I	E	N	S
N	I	C	H	-	E	R	-

Ce que cette visualisation permet de comprendre, c'est que si une opération sur les derniers caractères n'intervient pas comme première opération, on peut supposer qu'il en est ainsi (dit autrement, l'ordre des opérations ne compte pas). Sur notre exemple, la première opération aurait très bien pu être être la substitution du R et du S, sans rien changer aux autres opérations (ou bien, en considérant la deuxième version, la suppression du S).
On en conclut que les inégalités démontrées sont bien des égalités.

3.2 Mise en œuvre

Les fonctions à compléter se trouvent dans le fichier `Levenshtein_Acompleter.py`.

Traitement d'un exemple à la main

Remplir le tableau suivant : la case (i, j) doit contenir la distance entre les mots constitués des i premières lettres de NICHER et des j premières lettres de CHIENS. On appliquera les relations de récurrences vues ci-dessus.

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

Une première version

Compléter la fonction `levenProgDyn(mot1,mot2)` afin qu'elle remplisse un tableau sur le modèle de l'exemple précédent puis renvoie la distance entre `mot1` et `mot2`.

Reconstruction de la suite des substitutions

Reprendre la fonction suivante en une fonction `levenProgDyn2(mot1,mot2)` qui renvoie un tableau de même dimension mais dont chaque case est une liste de deux éléments : le premier est le même que dans la fonction initiale, le second est la case précédant la case (i, j) lors de la suite des substitutions (on pourra initialiser cet élément à `None`).
On pourra auparavant représenter cela par des flèches dans le tableau traité à la main.
Compléter ensuite la fonction `reconstructionLeven(mot1,mot2,tabDist)` qui effectue cette reconstruction à partir de `mot1`, `mot2` et du tableau obtenu par la fonction `levenProgDyn2(mot1,mot2)`.

Résolution par mémoïsation

Compléter tout d'abord la fonction `levenRecNaive(mot1,mot2)` qui calcule la distance de Levenshtein de manière récursive sans mémoïsation.
Quel est le problème posé par cette façon de procéder ?

Compléter ensuite la fonction `levenMemo(mot1,mot2)` qui calcule cette distance par une méthode de mémoïsation, en utilisant un dictionnaire.