TP : Programmation dynamique

Fichier à télécharger. Sur la page web, vous trouverez `annexe_TP_prog_dyn.py`, contenant des exemples de graphes.

1 Implémentation de l'algorithme de Floyd-Warshall

Le but est d'implémenter l'algorithme de Floyd-Warshall rappelé ci-dessous.

Algorithme 1 : Algorithme de Floyd-Warshall

Entrée : Un graphe pondéré $G = (V, E, \omega)$ donné par sa matrice d'adjacence M
Sortie : La matrice $(\delta(i,j))_{0 \leq i,j < n}$ des plus petits poids des chemins entre deux sommets quelconques du graphe

$A \leftarrow \text{copie}(M)$;
pour *tout* k *entre 0 et* $n-1$ *faire* **faire**
  pour *tout* i *entre 0 et* $n-1$ *faire* **faire**
    pour *tout* j *entre 0 et* $n-1$ *faire* **faire**
      $a_{i,j} \leftarrow \min(a_{i,j}, a_{i,k} + a_{k,j})$
Renvoyer A

Question 1. *Copie d'une liste de listes*. On rappelle que pour faire une copie d'une liste `L` « simple » (c'est-à-dire constituée d'éléments de type immuable, comme des entiers ou des flottants), `L[:]` suffit. Pour faire une copie en profondeur d'une liste de listes, ceci est insuffisant car les listes internes ne sont pas copiées. Écrire une fonction `copie_matrice(L)` renvoyant une copie d'une liste de listes (qu'on pourra supposer rectangulaire). Vérifiez que le comportement est bon comme ceci :

```
>>> M = [[0,1,2], [3,4,5]]
>>> X=copie_matrice(M)
>>> X[0][0] = 6 # modification de X
>>> M
[[0, 1, 2], [3, 4, 5]] # M inchangée
>>> X
[[6, 1, 2], [3, 4, 5]] # X changée
```

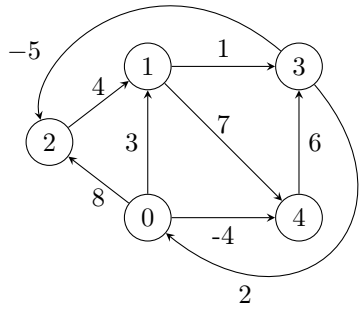

FIGURE 1: Le graphe du cours (graphe `g1`)

Question 2. *Matrice d'adjacence associée à un graphe.* Les graphes de l'annexe sont donnés par listes d'adajcence : avec $V = [\![0, n-1]\!]$ l'ensemble des sommets, un graphe orienté pondéré (V, E, ω) est représenté comme une liste `G` à n éléments, où `G[i]` est la liste des couples $(j, \omega(i,j))$, avec j voisin de i et $\omega(i,j) \in \mathbb{R}$ le poids de l'arc (i,j). Écrire une fonction `rep_mat(G)` renvoyant la représentation de `G` par matrice d'adjacence : votre fonction renverra $M = (m_{i,j})$ où :

— si (i,j) est un arc du graphe, $m_{i,j} = \omega(i,j)$;

— si $i = j$, $m_{i,j} = 0$;

— si $i \neq j$ et si (i,j) n'est pas un arc du graphe, $m_{i,j} = +\infty$.

On rappelle que l'élément $+\infty$ s'encode facilement en Python comme le flottant `float('inf')`.

```
>>> rep_mat(G1)
[[0, 3, 8, inf, -4], [inf, 0, inf, 1, 7], [inf, 4, 0, inf, inf], [-2, inf, -5, 0, inf],
[inf, inf, inf, 6, 0]]
```

Question 3. Implémentez l'algorithme de Floyd Warshall, qui prend en entrée un graphe représenté par matrice d'adjacence.

```
>>> floyd_warshall(rep_mat(G1))
[[0, 1, -3, 2, -4], [-1, 0, -4, 1, -5], [3, 4, 0, 5, -1], [-2, -1, -5, 0, -6], [4, 5, 1, 6, 0]]
>>> floyd_warshall(rep_mat(G2))
[[0, 5, 3, 7, -1, 9], [-1, 0, 2, 2, -2, 8], [1, 2, 0, 4, 0, 6], [-3, 2, 0, 0, -4, 6], [5, 6, 4, 8, 0, 10],
[-3, -2, -4, 0, -4, 0]]
>>> floyd_warshall(rep_mat(G3))
[[0, 8, 9, 7, 5], [11, 0, 1, 4, 2], [11, 19, 0, 4, 16], [7, 15, 6, 0, 12], [9, 3, 4, 2, 0]]
>>> floyd_warshall(rep_mat(G4))
[[0, 5, 1, 8, 3, 6], [inf, 0, inf, 4, 4, 1], [inf, 4, 0, 7, 2, 5], [inf, inf, inf, 0, inf, inf],
[inf, 2, inf, 5, 0, 3], [inf, 5, inf, 8, 3, 0]]
```

Question 4. On peut montrer facilement que s'il existe dans le graphe un circuit de poids total strictement négatif, alors la matrice A renvoyée par l'algorithme de Floyd-Warshall possèdera à la fin de l'algorithme un coefficient diagonal strictement négatif (pour un graphe sans tel circuit, la diagonale reste remplie de zéro). Modifiez l'algorithme pour qu'il échoue dans le cas de l'existence d'un circuit de poids total strictement négatif. On utilisera `assert`, on rappelle que `assert condition, message` teste `condition`, si celle-ci s'évalue en `False` alors l'exécution est interrompue et le message d'erreur `message` (chaîne de caractères), est affiché.

```
>>> floyd_warshall(rep_mat(G1))
[[0, 1, -3, 2, -4], [-1, 0, -4, 1, -5], [3, 4, 0, 5, -1], [-2, -1, -5, 0, -6], [4, 5, 1, 6, 0]]
>>> floyd_warshall(rep_mat(Gneg))
[...]
AssertionError: le graphe possède un circuit de poids total < 0 !
```

Question 5. Modifier l'algorithme pour calculer en même temps la *matrice de liaison* $\Pi = (\pi_{i,j})_{0 \leq i,j < n}$, définie par :

$$\pi_{i,j} = \begin{cases} \text{le prédecesseur de } j \text{ dans un plus court chemin de } i \text{ à } j, \text{ s'il en existe un} \\ -1 \text{ sinon (on pourrait mettre autre chose)} \end{cases}$$

Pour ce faire, il suffit d'intialiser la matrice Π comme suit : $\pi_{i,j} = \begin{cases} i \text{ si } (i,j) \in E \\ -1 \text{ sinon} \end{cases}$. Lorsque dans l'algorithme, on a $a_{i,j} > a_{i,k} + a_{k,j}$, on réalise l'affectation $a_{i,j} \leftarrow a_{i,k} + a_{k,j}$ et parallèlement $\pi_{i,j} \leftarrow \pi_{k,j}$.

```
>>> floyd_warshall(rep_mat(G1))
([[0, 1, -3, 2, -4], [-1, 0, -4, 1, -5], [3, 4, 0, 5, -1], [-2, -1, -5, 0, -6], [4, 5, 1, 6, 0]],
[[-1, 2, 3, 4, 0], [3, -1, 3, 1, 0], [3, 2, -1, 1, 0], [3, 2, 3, -1, 0], [3, 2, 3, 4, -1]])
```

Question 6. Écrire une fonction `imprime_chemin(Pi, i, j)` d'impression à l'écran d'un plus court chemin entre i et j s'il en existe un, avec `Pi` la matrice de liaison. On construira d'abord le chemin (à l'envers) avant d'imprimer à l'écran.

```
>>> imprime_chemin(floyd_warshall(rep_mat(G1))[1], 0, 1)
0 -> 4 -> 3 -> 2 -> 1
>>> imprime_chemin(floyd_warshall(rep_mat(G4))[1], 1, 0)
[...]
AssertionError: il n'existe pas de tel chemin !
```

2 Distance d'édition entre deux chaînes de caractères

Définition du problème. Un problème important en génétique, est le calcul de la *distance d'édition* entre deux chaînes de caractères. On convient que sur une chaîne de caractères, on peut effectuer les opérations *d'édition* suivantes :

— insérer un caractère à un endroit donné ;

— supprimer un caractère donné ;

— modifier un caractère en un autre.

La distance d'édition (ou distance de Levenshtein) entre deux chaînes s et t est définie par le nombre minimal $d(s,t)$ d'opérations [1] à effectuer pour passer de s à t. Il est facile de vérifier que d est bien une distance au sens mathématique, c'est-à-dire que pour toutes chaînes s et t :

— $d(s,s) = 0$;

— $d(s,t) = d(t,s)$;

— $d(s,t) \leq d(s,u) + d(t,u)$ pour toute chaîne u (inégalité triangulaire).

On peut montrer que la distance entre les chaînes $s =$ « ACTGTAA » et $t =$ « AACTGC » est 4. Une suite possible d'opérations d'édition permettant de passer de s à t est la suivante :

$$\text{ACTGTAA} \underset{\text{suppression}}{\longrightarrow} \text{ACTGTA} \underset{\text{suppression}}{\longrightarrow} \text{ACTGT} \underset{\text{modification}}{\longrightarrow} \text{ACTGC} \underset{\text{insertion}}{\longrightarrow} \text{AACTGC}$$

Relation de récurrence. De même que pour le problème de la PLSSC, on peut introduire $d_{i,j}$, distance d'édition entre les préfixes de taille i de s et j de t, pour $0 \leq i \leq n$ et $0 \leq j \leq m$, avec n et m les tailles de s et t. On a alors la relation suivante :

$$d_{i,j} = \begin{cases} i & \text{si } j = 0 \\ j & \text{si } i = 0 \\ d_{i-1,j-1} & \text{si } s_{i-1} = t_{j-1} \\ 1 + \min(d_{i-1,j}, d_{i,j-1}, d_{i-1,j-1}) & \text{sinon.} \end{cases}$$

Preuve : les trois premiers points sont assez évidents, prouvons le dernier.

- Déja, $d_{i,j} \leq 1 + \min(d_{i-1,j}, d_{i,j-1}, d_{i-1,j-1})$. En effet, on a par exemple $d_{i,j} \leq 1 + d_{i-1,j}$, car à partir d'une suite d'opérations de longueur $d_{i-1,j}$ permettant de passer de $s[:i-1]$ à $t[:j]$, on en obtient une de longueur $1 + d_{i-1,j}$ de $s[:i]$ à $t[:j]$ en lui rajoutant au début la suppression du dernier caractère de $s[:i]$. On montre de même que $d_{i,j} \leq 1 + d_{i,j-1}$ et $d_{i,j} \leq 1 + d_{i-1,j-1}$, d'où la conclusion.
- Pour montrer l'inégalité inverse, considérons une suite d'opérations de longueur $d_{i,j}$ menant de $s[:i]$ à $t[:j]$. Le dernier caractère de $t[:j]$ apparaît dans cette suite après une insertion, une modification, ou une suppression. Quite à modifier un peu la suite d'opérations, supposons que cette opération est faite en dernier. Si par exemple c'est une insertion, la dernière étape passe de $t[:j-1]$ à $t[:j]$, en enlevant cette étape on obtient une suite de longueur $d_{i,j} - 1$ de $s[:i]$ à $t[:j-1]$. Et donc, $\min(d_{i-1,j}, d_{i,j-1}, d_{i-1,j-1}) \leq d_{i,j-1} \leq d_{i,j} - 1$. De même s'il s'agit d'une insertion ou d'une modification.

Question 7. Écrire une fonction `calcul_D`(s,t)|. Avec n et m les longueurs de ces chaînes, votre fonction calculera et renverra tous les $(d_{i,j})_{0 \leq i \leq n, 0 \leq j \leq m}$ (attention, la matrice est de taille $(n+1) \times (m+1)$. Voici la matrice obtenue avec les chaînes « ACTGTAA » et « AACTGC » :

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 0 & 1 & 2 & 3 & 4 & 5 \\ 2 & 1 & 1 & 1 & 2 & 3 & 4 \\ 3 & 2 & 2 & 2 & 1 & 2 & 3 \\ 4 & 3 & 3 & 3 & 2 & 1 & 2 \\ 5 & 4 & 4 & 4 & 3 & 2 & 2 \\ 6 & 5 & 4 & 5 & 4 & 3 & 3 \\ 7 & 6 & 5 & 5 & 5 & 4 & 4 \end{pmatrix}$$

Quelle est la complexité de votre fonction ?

1. On peut généraliser en introduisant un coût différent, par exemple en considérant qu'une insertion / suppression coûte plus cher qu'une modification. L'approche proposée s'étend facilement.

Question 8. *Plus difficile.* Écrire une fonction `affiche_transformations(s,t)` affichant une suite optimale de tranformations permettant de passer de s à t. Par exemple :

```
>>> affiche_transformations('ACTGTAA', 'AACTGC')
départ, s =  ACTGTAA. But : t =  AACTGC.
suppression :  ACTGTA
suppression :  ACTGT
modification :  ACTGC
insertion :  AACTGC
```

Indication : après calcul de la matrice des distances, on pourra procéder avec une fonction récursive `aux(i,j,reste)`, telle que les opérations restantes soient le passage de `s[:i]+reste` à `t[:j]+reste` (on rappelle que `+` est la concaténation de chaînes).

3 Pour aller plus loin...

Coder l'exercice 4 du TD !