Algorithmique
Partiel n° 3 (P3)

INFO-SPÉ - S3
EPITA

19 décembre 2023 - 9 : 30

Consignes (à lire) :

- Vous devez répondre sur **les feuilles de réponses prévues à cet effet**.
  - Aucune autre feuille ne sera ramassée (gardez vos brouillons pour vous).
  - Répondez dans les espaces prévus, **les réponses en dehors ne seront pas corrigées** : utilisez des brouillons !
  - Ne séparez pas les feuilles à moins de pouvoir les ré-agrafer pour les rendre.
  - Aucune réponse au crayon de papier ne sera corrigée.
- La présentation est notée en moins, c'est à dire que vous êtes noté sur 20 et que les points de présentation (2 au maximum) sont retirés de cette note.
- **Le code :**
  - Tout code doit être écrit dans le langage Python (pas de C, CAML, ALGO ou autre).
  - **Tout code Python non indenté ne sera pas corrigé.**
  - Tout ce dont vous avez besoin (classes, fonctions, méthodes) est indiqué dans l'énoncé !
  - Vous pouvez également écrire vos propres fonctions auxiliaires, dans ce cas **elles doivent être documentées** (on doit savoir ce qu'elles font).
  - Comme d'habitude l'optimisation est notée. Si vous écrivez des fonctions non optimisées, vous serez noté sur moins de points. [1]
- Durée : 2h00

1. Des fois, il vaut mieux moins de points que pas de points.

Exercice 1 (Connectivity – *5 points*)

Soit le graphe non orienté $G = < S, A >$, où les sommets sont numérotés de 0 à 9.

Les algorithmes `trouver` (avec compression des chemins) et `réunir` (union pondérée) appliqués à la liste des arêtes A de G ont permis de construire le vecteur P suivant.

	0	1	2	3	4	5	6	7	8	9
P	5	6	7	9	5	-3	7	-4	9	-3

1. Donner la matrice d'adjacence de G^* la fermeture transitive de G (pas de valeur = *faux*, 1 = *vrai*).

2. On applique cette fois-ci les versions non optimisées de `trouver` et `réunir` à la liste des arêtes A (mélangée aléatoirement à chaque fois) de G.
   Pour chaque vecteur donné sur les feuilles de réponses, indiquer s'il peut correspondre au résultat. Dans le cas contraire, entourer les valeurs qui posent problème.

3. À partir de P, on ajoute l'arête {1, 3} (avec l'union pondérée et la compression des chemins). Donner le nouveau vecteur P.

Exercice 2 (Longest Path – *10 points*)

On cherche dans un graphe orienté acyclique le plus long chemin.

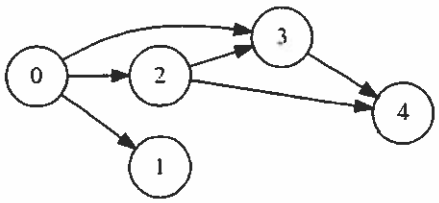

FIGURE 1 – Graphe G_1

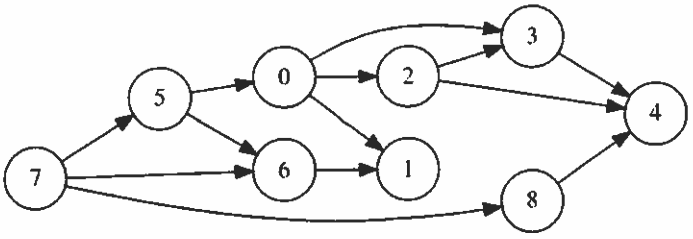

FIGURE 2 – Graphe G_2

Exemples :
— dans le graphe G_1 le plus long chemin est $0 \to 2 \to 3 \to 4$ de longueur 3 ;
— dans le graphe G_2 le plus long chemin est $7 \to 5 \to 0 \to 2 \to 3 \to 4$ de longueur 5.

1. *Un peu d'aide*
   (a) Construire la forêt couvrant du parcours profondeur complet du graphe G_2 (figure 2) en choisissant les sommets en ordre croissant. *Ajouter à la forêt les autres arcs rencontrés en les qualifiant d'une légende explicite.*
   (b) Remplir le vecteur *long*, avec $\forall s \in S_2, long[s]$ est la longueur du plus long chemin depuis s dans $G_2 = < S_2, A_2 >$.

2. Écrire la fonction `longest_path(G)`, utilisant **obligatoirement un parcours profondeur**
   — qui retourne `None` si le graphe G n'est pas acyclique ;
   — sinon, si G est acyclique (sans circuit), la fonction retourne
     — soit la liste des sommets du plus long chemin dans le graphe orienté G (tous les points),
     — soit uniquement la longueur de ce plus long chemin (moins de points [2]).
     Indiquer la version choisie sur les feuilles de réponses.

2. Des fois, il vaut mieux moins de points que pas de points.

Exercice 3 (Smallest Level – *5 points*)

Définition :

— La **distance** entre deux sommets d'un graphe est le nombre d'arêtes (d'arcs) d'une **plus courte chaîne (chemin)** entre ces deux sommets.

On cherche dans une graphe non orienté connexe le plus petit ensemble de sommets à une même distance strictement positive d'une source donnée.

Par exemple dans le graphe G3, à partir du sommet 0, il y a
— 4 sommets à la distance 1,
— 2 sommets à la distance 2,
— 2 à la distance 3
— et 1 seul à la distance 4 (le sommet 4).

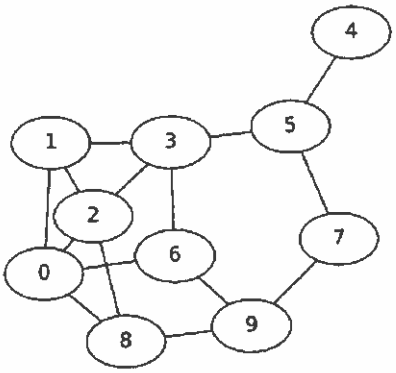

FIGURE 3 – Graphe G3

Si plusieurs possibilités, on prendra la liste des sommets à la plus petite distance. Par exemple, à partir du sommet 1 :
— il y a 3 sommets à la distance 1,
— 3 à la distance 2
— et 3 à la distance 3.
Ce sont les sommets à la distance 1 qui seront retenus (0, 2 et 3).

Écrire la fonction `smallest_level(G, src)` qui retourne la plus petite liste des sommets à une même distance (> 0) du sommet *src* dans le graphe non orienté connexe G.

Exemples d'applications avec G3 le graphe de la figure 3 :

```
>>> for s in range(10):
        print(s, "=>", smallest_level(G3, s))

0 => [4]
1 => [0, 2, 3]
2 => [4, 7]
3 => [1, 2, 5, 6]
4 => [5]
5 => [0, 8]
6 => [4]
7 => [5, 9]
8 => [5]
9 => [1, 4]
```

Annexes

Les classes `Graph` et `Queue` sont supposées importées.

Les graphes

Tous les exercices utilisent l'implémentation par listes d'adjacences des graphes. Les listes d'adjacence sont triées en ordre croissant.

Les graphes manipulés ne peuvent pas être vides (ni l'ensemble des sommets, ni celui des liaisons ne sont vides). Il n'y a pas de liaisons multiples ni boucles.

```
class Graph:
    def __init__(self, order, directed, labels=None):
        self.order = order
        self.directed = directed
        self.adjlists = []
        for i in range(order):
            self.adjlists.append([])
        self.labels = labels
```

Les files

- `Queue()` returns a new queue
- q`.enqueue(`e`)` enqueues e in q
- q`.dequeue()` returns the first element of q, dequeued
- q`.isempty()` tests whether q is empty

Autres

- `range`
- `min, max`
- sur les listes :
  - `len(L):int`
  - `L.append(elt):None`
  - `L.pop():element`
  - `L.pop(index):element`
  - `L.insert(index, elt):None`
  - `L.reverse():None`

  Et n'importe quel opérateur...

Vos fonctions

Vous pouvez également écrire vos propres fonctions, dans ce cas elles doivent être **documentées** (on doit savoir ce qu'elles font).