Algorithmique
Partiel n° 3 (P3)

Info-spé - S3
Epita

5 janvier 2021 - 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, dans ce cas **elles doivent être documentées** (on doit savoir ce qu'elles font).
    Dans tous les cas, la dernière fonction écrite doit être celle qui répond à la question.
- ☐ Durée : 2h00

Exercice 1 (Dans les profondeurs de la forêt couvrante – *3 points*)

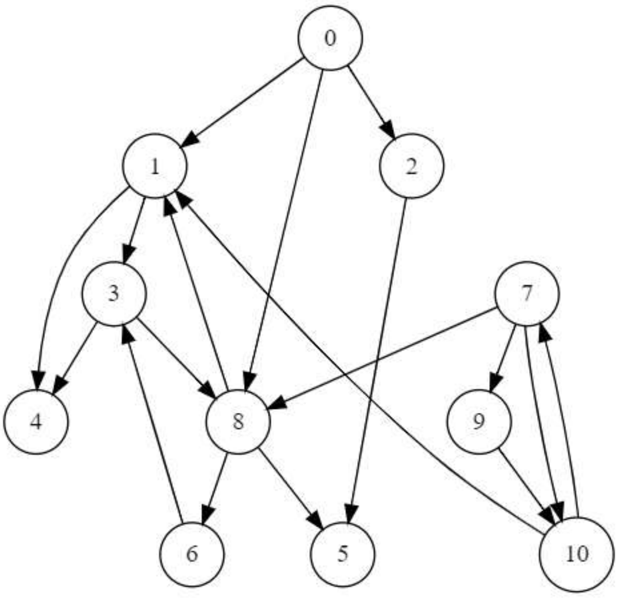

Figure 1 – Un graphe orienté

1. Représenter (dessiner) la forêt couvrante associée au parcours profondeur du graphe de la figure 1. *Ajouter aussi les autres arcs en les qualifiant à l'aide d'une légende explicite.* On considérera le sommet 0 comme base du parcours, les sommets devant être choisis en ordre numérique croissant.

2. Remplir les vecteurs d'ordres de rencontre en préfixe et suffixe, établis avec un compteur unique commençant à 1, correspondants au parcours de la question précédente.

Exercice 2 (Union-Find – *4 points*)

Soit le graphe non orienté $G = < S, A >$, où les sommets sont numérotés de 0 à 13. Les algorithmes vus en cours `trouver` (avec compression des chemins) et `réunir` (union pondérée) ont permis de construire, à partir de la liste des arêtes (A), le vecteur p suivant :

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
p	5	8	5	8	11	-4	5	10	-6	12	8	12	-4	8

1. Donner le nombre de sommets de chaque composante connexe de G (peu importe l'ordre).

2. Quelles arêtes **suffit-il** d'ajouter pour rendre le graphe connexe ?

3. Parmi les chaînes suivantes, quelles sont celles qui ne peuvent pas exister dans G ?
   — $3 \leftrightsquigarrow 7$
   — $11 \leftrightsquigarrow 6$
   — $0 \leftrightsquigarrow 13$
   — $4 \leftrightsquigarrow 9$

4. On ajoute l'arête 7 – 4 au graphe G (avec l'union pondérée et la compression des chemins). Donner le nouveau vecteur p.

Exercice 3 (Distance au départ – *5 points*)

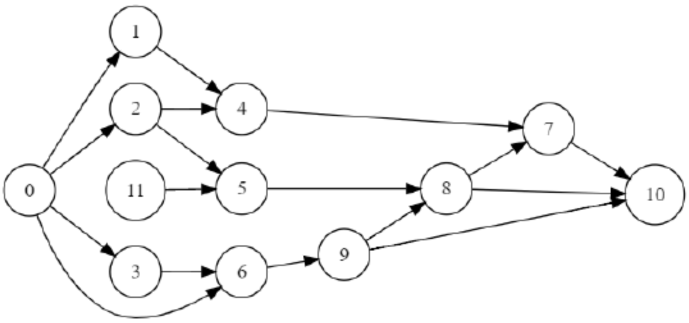

Figure 2 – Digraph `G1`

L'objectif dans cet exercice est de connaître les sommets qui se trouvent, à partir d'un sommet de départ, à une *distance* dans un intervalle donné $[d_{min}, d_{max}]$.

Écrire la fonction `dist_range(`G`, `src`, `$dmin$`, `$dmax$`)` qui retourne la liste des sommets à une *distance* comprise entre $dmin$ et $dmax$ du sommet src dans la graphe G (avec $0 < dmin \leq dmax$).

```
>>> dist_range(G1, 0, 2,3)
[4, 5, 9, 7, 8, 10]

>>> dist_range(G1, 0, 2,2)
[4, 5, 9]

>>> dist_range(G1, 0, 1,2)
[1, 2, 3, 6, 4, 5, 9]
```

Exercice 4 (Get cycle – *5 points*)

En utilisant **obligatoirement un parcours profondeur**, écrire la fonction `get_cycle(`G`)` qui cherche un cycle dans G graphe non orienté. Si un cycle est trouvé (n'importe lequel, voir les exemples ci-dessous), il est retourné sous la forme d'une liste de sommets. Sinon la fonction retourne une liste vide.

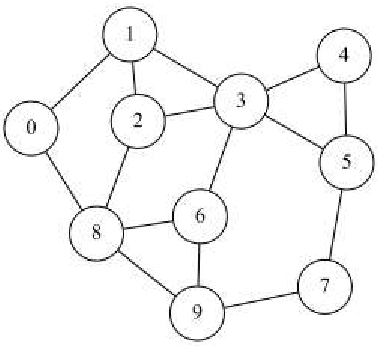

Figure 3 – Graph `G2`

Exemples de résultats différents (différentes versions de la fonction) sur le graphe de la figure 3 :

```
>>> get_cycle(G2)
[1, 0, 8, 2, 1]

>>> get_cycle_2(G2)
[0, 8, 2, 1, 0]

>>> get_cycle_3(G2)
[1, 2, 3, 1]
```

Exercice 5 (What is this ? – *3 points*)

Les fonctions suivantes sont définies :

```python
def __build(G, x, D, P, NG):
    for y in G.adjlists[x]:
        if D[y] == None:
            D[y] = D[x] + 1
            __build(G, y, D, P, NG)
            NG.addedge(x, y)
        else:
            if D[y] < D[x] and not P[y]:
                NG.addedge(x, y)
    P[x] = True

def build(G):
    D = [None] * G.order
    P = [False] * G.order
    NG = Graph(G.order, True)
    for s in range(G.order):
        if D[s] == None:
            D[s] = 0
            __build(G, s, D, P, NG)
    return NG
```

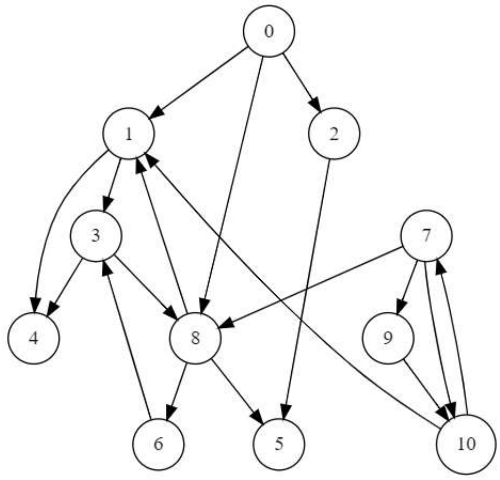

Figure 4 – Graphe orienté G_4

1. Dessiner le graphe résultat de l'appel `build(`G_4`)` avec G_4 le graphe de la figure 4 (les listes d'adjacence sont triées en ordre croissant) .

2. Pendant le parcours, pour chaque sommet s :
   (a) Que représente `D[`s`]` ?
   (b) Que représente `P[`s`]` ?

Annexes

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

Les graphes

Tous les exercices utilisent l'implémentation par listes d'adjacences des graphes.

Les graphes manipulés ne peuvent pas être vides. Il n'y a pas de liaisons multiples ni boucles.

```
class Graph:
    def __init__(self, order, directed = False):
        self.order = order
        self.directed = directed
        self.adjlists = []
        for i in range(order):
            self.adjlists.append([])

    def addedge(self, src, dst):
        self.adjlists[src].append(dst)
        if not self.directed and dst != src:
            self.adjlists[dst].append(src)
```

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)`
  — `L.append(elt)`
  — `L.pop()`
  — `L.pop(index)`
  — `L.insert(index, elt)`
  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).

Dans tous les cas, la dernière fonction écrite doit être celle qui répond à la question.