Algorithmique
Partiel n° 3 (P3)

Info-spé (S3)
Epita

19 décembre 2017 - 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

Cours

Exercice 1 (Union-Find – *3 points*)

Soit le graphe non orienté $G = < S, A >$, où les sommets sont numérotés de 0 à 13. Les algorithmes vus en cours `trouver` (sans compression) et `réunir` (version optimisée : 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 ?
   — $3 \rightsquigarrow 7$
   — $11 \rightsquigarrow 6$
   — $0 \rightsquigarrow 13$
   — $4 \rightsquigarrow 9$

Exercice 2 (Dessiner c'est gagner – *4 points*)

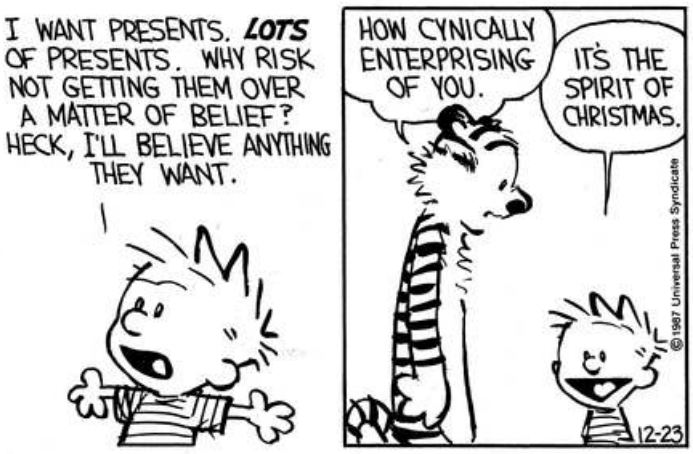
Soit le graphe `G=<S,A>` orienté avec :

```
  S={1,2,3,4,5,6,7,8,9,10}
et A={(1,2),(1,6),(1,7),(2,3),(2,6),(3,1),(3,5),(4,3),(4,8),(4,9),(4,10),(5,1),(7,6),
      (8,5),(8,10),(10,9)}
```

1. Représenter graphiquement le graphe correspondant à `G`.

2. Donner le tableau `DemiDegréIntérieur` tel que $\forall\ i \in [1, Card(S)]$, `DemiDegréIntérieur[i]` soit égal au demi-degré intérieur de `i` dans `G`.

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



Python

Exercice 3 (Graphes bipartis (Bipartite graph) – *6 points*)

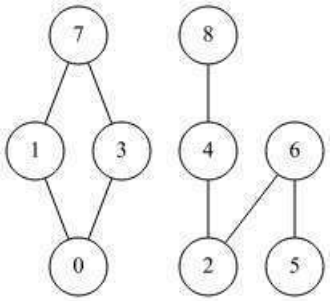


FIGURE 1 – Graphe biparti G_1

Un graphe biparti est un graphe (un multigraphe) non orienté $G =< S, A >$, dans lequel S peut être partitionné en deux ensembles S_1 et S_2 tels que $(u, v) \in A$ implique soit que $u \in S_1$ et $v \in S_2$, soit que $u \in S_2$ et $v \in S_1$. Aucune arête ne doit relier deux sommets d'un même ensemble.

Une autre manière de voir le graphe biparti : ses sommets peuvent être "colorés" en deux couleurs de sorte que deux sommets de même couleurs ne sont pas adjacents.

Le graphe de la figure 1 est biparti avec, par exemple $S_1 = \{1, 3, 4, 6\}$ et $S_2 = \{0, 2, 5, 7, 8\}$.

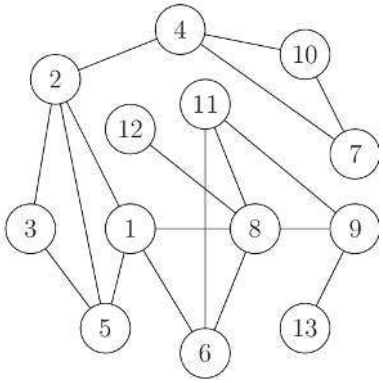


FIGURE 2 – Graphe G_2

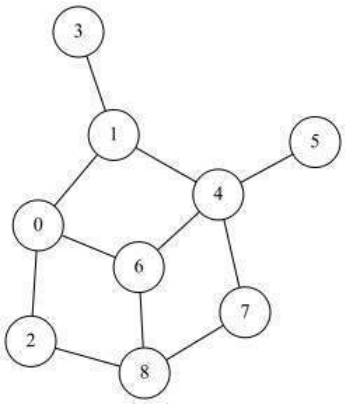


FIGURE 3 – Graphe G_3

1. Les graphes G_2 et G_3 des figures 2 et 3 sont-ils bipartis ? Si oui, donner les deux ensembles S_1 et S_2.

2. Écrire une fonction, qui utilise **obligatoirement** un parcours largeur, qui teste si un graphe, représenté par listes d'adjacence, est biparti.

Exercice 4 (Acyclic – *5 points*)

Définition :
Un graphe orienté *acyclique* est un graphe sans circuits.

Écrire la fonction `is_acyclic` qui vérifie si un graphe orienté représenté par listes d'adjacence est *acyclique*.

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

```
1  import Graph, Queue
2
3  def __aux(G, v, m, res, count):
4      q = Queue()
5      q.enqueue(v)
6      m[v] = True
7      while not q.isempty():
8          cur = q.dequeue()
9          res[cur] = count
10         count += 1
11         for succ in G.adjlists[cur]:
12             if not m[succ]:
13                 m[succ] = True
14                 q.enqueue(succ)
15     return count
16
17 def mystery(G, src):
18     m = [False] * G.order
19     res = [0] * G.order
20     count = __aux(G, src, m, res, 0)
21     for v in range(G.order):
22         if not m[v]:
23             count = __aux(G, v, m, res, count)
24     NG = Graph(G.order, directed=G.directed)
25     for v in range(G.order):
26         for s in G.adjlists[v]:
27             if G.directed or res[v] <= res[s]:
28                 NG.addedge(res[v], res[s])
29     return NG
```

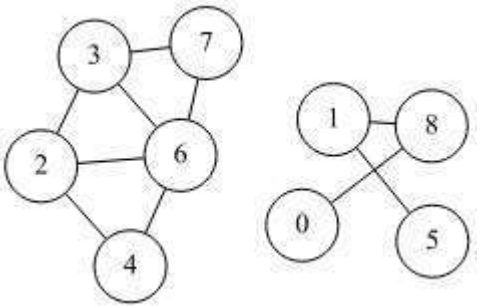


FIGURE 4 – Graphe G_4

1. Dessiner le graphe résultat de l'appel `mystery(`G_4`, 0)` (les sommets sont choisis en ordre croissant) avec G_4 le graphe de la figure 4.

2. Donner la liste des sommets dans l'ordre où ils ont été traités.

3. Soit un graphe non orienté possédant k composantes connexes. Combien de composantes contiendra le graphe obtenu par l'application de la fonction `mystery` ?

Annexes

Les classes `Graph` et `Queue` sont supposées importées. Les graphes manipulés ne peuvent pas être vides.

Les graphes

```
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

- sur les listes : `len`
- `range`.

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.