Algorithmique
Contrôle n° 4 (C4)

Info-spé (S4)
Epita

1 mars 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 les annexes (dernière page). **Lisez-les !**
  - — 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 (CFC – *4 points*)

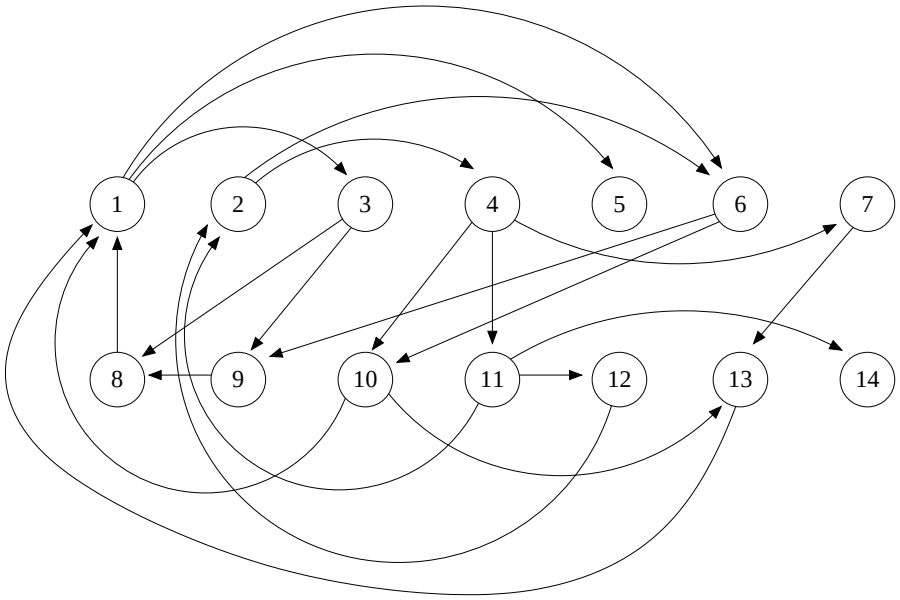


FIGURE 1 – Graphe orienté.

1. Représenter (dessiner) la forêt couvrante associée au parcours en profondeur du graphe de la figure 1. On considérera le sommet 1 comme base du parcours et les successeurs seront traités en ordre croissant. On ne représentera que les arcs couvrants du parcours.
2. Combien y a t-il de composantes fortement connexes dans ce graphe ?
3. Représenter les différentes composantes fortement connexes de ce même graphe. Pour faciliter la représentation, vous vous contenterez de colorier les sommets d'une même composante avec la même couleur (une couleur par composante). Inutile de représenter les arcs.

Exercice 2 (Bi-connexité – *2 points*)

Dans cet exercice nous travaillons avec des graphes simples, non-orientés sans boucle.

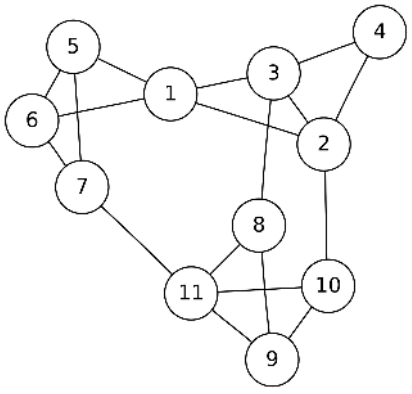


FIGURE 2 – Graphe exemple

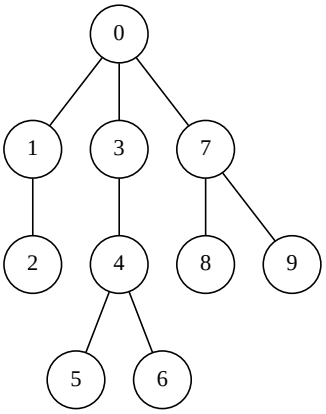
1. Rappelez la définition de la bi-connexité.
2. Le graphe de la figure 2 est-il bi-connexe ?
3. Donnez la définition d'un point d'articulation.
4. Donnez la définition d'un isthme.

Exercice 3 (Even Tree – *5 points*)

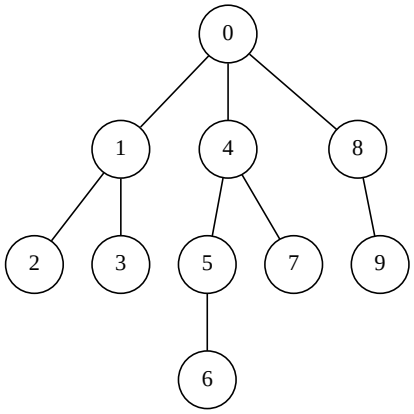
Nous travaillons ici avec des *arbres* (graphes simples connexes sans cycles). Le but ici est de trouver le nombre **maximum** d'arêtes que l'on peut retirer d'un arbre pour obtenir une forêt telle que chaque composante connexe de la forêt contienne un nombre **pair** de sommets.

Note : Le graphe donné est tel qu'il peut toujours être décomposé en composantes contenant un nombre pair de sommets.

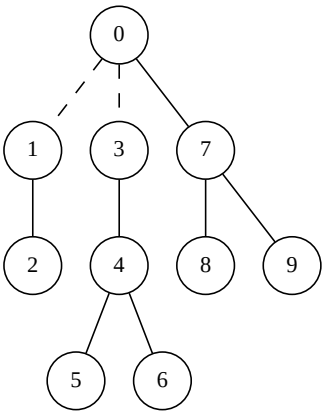
Par exemple, voici ce que l'on trouve lors d'un parcours profondeur :

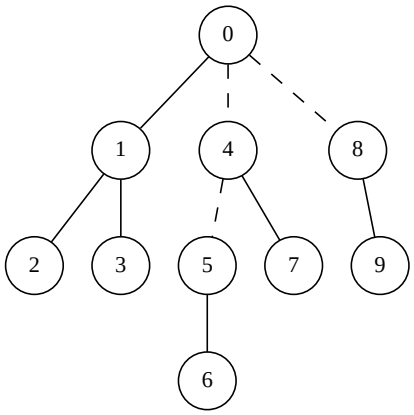


2 arêtes peuvent être enlevées :
{0, 1} et {0, 3}



3 arêtes peuvent être enlevées :
{4, 5}, {0, 4} et {0, 8}





Dans les deux cas, le résultat est une forêt dont les composantes connexes sont de tailles paires.

Note : cela fonctionne quelque soit le sommet de départ.

Écrire la fonction `evenTree(`G`)` qui calcule le nombre maximum d'arêtes qui peuvent être enlevées de G selon les spécifications précédentes.

Dans la suite, on appelle *vecteur des composantes connexes* du graphe non orienté $G =< S, A >$ contenant k composantes connexes, le vecteur cc tel que $\forall s \in S, cc[s]$ est le numéro de la composante connexe à laquelle appartient s (un entier compris entre 1 et k).

Exercice 4 (Warshall – *3,5 points*)

Rappels :

- On appelle *fermeture transitive* d'un graphe non orienté G défini par le couple $< S, A >$, le graphe $G*$ défini par le couple $< S, A* >$ tel que pour toutes paires de sommets $x, y \in S$, il existe une arête $\{x, y\}$ dans $G*$ si-et-seulement-si il existe une chaîne entre x et y dans G.
- L'algorithme de Warshall calcule la matrice d'adjacence de la fermeture transitive d'un graphe.

Écrire la fonction `CCFromWarshall(`M`)` qui, à partir de M, matrice résultat de l'algorithme de Warshall appliquée au graphe G, retourne le couple (k, cc), avec k le nombre de composantes connexes de G, et cc le vecteur (une liste en Python) des composantes connexes de G.

Exercice 5 (Union-Find – *3 points*)

Soit un graphe $G =< S, A >$. À partir de n le nombre de sommets, et L la liste des arêtes de G (liste de couples de sommets), écrire la fonction `CCFromEdges(`n`, `L`)` qui retourne le couple (k, cc), avec k le nombre de composantes connexes de G, et cc le vecteur (une liste en Python) des composantes connexes de G.

Exercice 6 (Journey To The Moon – *3,5 points*)

Les états membres de l'UN désirent envoyer 2 personnes sur la lune. Conformément à leur principe d'unité mondiale, il leur faut trouver deux astronautes de pays différents.

Il y a N astronautes entrainés, numérotés de 0 à $N-1$. Mais les responsables de la mission n'ont pas reçu d'informations sur la citoyenneté de chaque astronaute. La seule information dont ils disposent est que certaines paires d'astronautes appartiennent au même pays.

Votre tâche consiste à calculer de combien de façons ils peuvent choisir une paire d'astronautes appartenant à différents pays. Vous disposez de suffisamment de couples pour vous permettre d'identifier les groupes d'astronautes. Par exemple, si 1, 2, 3 sont des astronautes du même pays ; il suffit de mentionner que (1, 2) et (2, 3) sont des paires d'astronautes du même pays sans fournir d'informations sur une troisième paire (1, 3).

L est une liste de couples (A, B) tels que A et B, tous deux dans $[0, N-1]$, sont des astronautes du même pays.

Exemples :

- Avec $N = 5$, L = [(0, 1), (0, 2), (3, 4)], le résultat est $6 = 3 * 2$
- Avec $N = 9$, L = [(0, 1), (0, 2), (1, 3), (4, 5), (6, 7), (7, 8)], le résultat est $26 = 4 * 2 + 4 * 3 + 2 * 3$

Écrire la fonction `Moon(`N`, `L`)` qui calcule le nombre de paires différentes d'astronautes pouvant être sélectionnées.

Annexes

Graphs

Le graphe de l'exercice 4 est représenté par listes d'adjacence.

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

Les fonctions que vous pouvez utiliser

— sur les listes : `len`, `append`

— `range`

Fonctions données

```
    def find(x, p):
        rx = x
        while p[rx] >= 0:
            rx = p[rx]
        return rx

    def union(x, y, p):
        rx = find(x, p)
        ry = find(y, p)
        if rx != ry:
            if p[rx] < p[ry]:
                p[rx] = p[rx] + p[ry]
                p[ry] = rx
            else:
                p[ry] = p[ry] + p[rx]
                p[rx] = ry

    def build(L, n):
        '''
        n: integer > 0
        L: list of pairs (a, b) with a and b in [0, n[
        '''
        p = [-1]*n
        for (x, y) in L:
            union(x, y, p)
        return p
```

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.