# Algorithmique
# Correction Partiel n° 3 (P3)

Info-spé - S3# – Epita

*12 mai 2021* - 9 : 30

***Solution 1*** **(Warshall - Trouver-Réunir – *4 points*)**

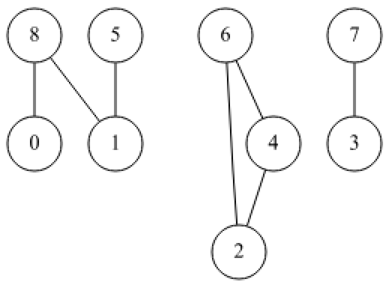


Figure 1 – Graph $G_1$

1. Les composantes connexes (ensembles de sommets) :
   - $C_1$ : $\{0, 1, 5, 8\}$
   - $C_2$ : $\{2, 4, 6\}$
   - $C_3$ : $\{3, 7\}$

2. La matrice d'adjacence de la **fermeture transitive** de $G_1$ (1 = vrai, vide = faux) :

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | | | | 1 | | | 1 |
| 1 | 1 | 1 | | | | 1 | | | 1 |
| 2 | | | 1 | | 1 | | 1 | | |
| 3 | | | | 1 | | | | 1 | |
| 4 | | | 1 | | 1 | | 1 | | |
| 5 | 1 | 1 | | | | 1 | | | 1 |
| 6 | | | 1 | | 1 | | 1 | | |
| 7 | | | | 1 | | | | 1 | |
| 8 | 1 | 1 | | | | 1 | | | 1 |

3. *Quels vecteurs pourraient correspondre au résultat ?*

| | oui | non |
|---|---|---|
| $P_1$ | ✓ | |
| $P_2$ | | ✓ |
| $P_3$ | | ✓ |
| $P_4$ | ✓ | |

***Solution 2*** **(Get back –** ***4 points*****)**

**Rappels :**

– S'il y un arc retour dans le parcours profondeur dans un graphe orienté alors il y a un circuit.
– L'arc retour est le seul arc non couvrant où l'extrémité finale n'a pas encore été rencontrés en suffixe.

**Spécifications :**

la fonction `acyclic(`$G$`)` détermine si le graphe orienté $G$ est acyclique.

```python
"""
__acyclic(G, x, M): DFS of G from x,
M mark vector: unmarked=None, prefix=1, suffix=2
return a boolean: False is a back edge was found
"""

def __acyclic(G, x, M):
    M[x] = 1
    for y in G.adjlists[x]:
        if M[y] == None:
            if not __acyclic(G, y, M):
                return False
        else:
            if M[y] != 2:
                return False
    M[x] = 2
    return True

def acyclic(G):
    M = [None] * G.order
    for s in range(G.order):
        if M[s] == None:
            if not __acyclic(G, s, M):
                return False
    return True
```

***Solution 3*** **(Density –** ***6 points*****)**

1. Pour un **graphe simple connexe** :
   (a) **Le moins dense** valeur minimale de $p$ : $n-1$ - Type de graphe : arbre (connexe sans cycle)
   (b) **Le plus dense** valeur maximale de $p$ : $n(n-1)/2$ - Type de graphe : complet

2. **Spécifications :**

   La fonction `density_components(`$G$`)` retourne la liste des *densités* des composantes connexes du graphe non orienté simple $G$.

```python
def __measures_cc(G, x, M):
    """
    return (n: nb vertices, p: nb edges) met during DFS of G from x
    """

    M[x] = True
    n = 1
    p = len(G.adjlists[x])
    for y in G.adjlists[x]:
        if not M[y]:
            (n_, p_) = __measures_cc(G, y, M)
            n += n_
            p += p_
    return (n, p)

```

```python
def __measures_cc_bfs(G, x, M):
    """
    return (n: nb vertices, p: nb edges) met during BFS of G from x
    """

    q = queue.Queue()
    q.enqueue(x)
    M[x] = True
    n = 0
    p = 0
    while not q.isempty():
        x = q.dequeue()
        n += 1
        p += len(G.adjlists[x])
        for y in G.adjlists[x]:
            if not M[y]:
                M[y] = True
                q.enqueue(y)
    return (n, p)

def density_components(G):
    M = [False] * G.order
    L = []
    for s in range(G.order):
        if not M[s]:
            (n, p) = __measures_cc(G, s, M)
            L.append((p // 2) / n)
    return L
```

### *Solution 4* (Levels – *6 points*)

Obligatoirement un parcours largeur ici !

**Spécifications :**

La fonction `levels`($G$, $src$) retourne la liste $L$ de longueur $exc(src) + 1$ dans laquelle chaque valeur $L[i]$ contient les sommets à une distance $i$ de $src$.

```python
# build L during the BFS

def levels(G, src):
    dist = [None] * G.order
    dist[src] = 0
    Levels = []
    L = []
    curdist = 0

    q = queue.Queue()
    q.enqueue(src)
    while not q.isempty():
        x = q.dequeue()
        if dist[x] > curdist:
            Levels.append(L)
            L = [x]
            curdist += 1
        else:
            L.append(x)

        for y in G.adjlists[x]:
            if dist[y] == None:
                dist[y] = dist[x] + 1
                q.enqueue(y)

    Levels.append(L)
    return Levels
```

```python


# build L after
def __distances(G, src, dist):
    """
    return src's eccentricity (only for v3)
    """
    dist[src] = 0
    q = queue.Queue()
    q.enqueue(src)
    while not q.isempty():
        x = q.dequeue()
        for y in G.adjlists[x]:
            if dist[y] == None:
                dist[y] = dist[x] + 1
                q.enqueue(y)
    return dist[x]

def levels2(G, src):
    dist = [None] * G.order
    __distances(G, src, dist)
    Levels = []
    for x in range(G.order):
        while dist[x] >= len(Levels):
            Levels.append([])
        Levels[dist[x]].append(x)
    return Levels

def levels3(G, src):
    dist = [None] * G.order
    ecc = __distances(G, src, dist)
    Levels = [[] for _ in range(ecc+1)]

    for x in range(G.order):
        Levels[dist[x]].append(x)
    return Levels
```