Informatique — MPSI/PCSI, 2024–2025

Correction du TP n° 12

1 Visualisation et parcours en largeur

1.1 Visualisation

Exercice 12.1

Le code suivant permet de tester les deux procédures sur les graphes tests G1 et G2 fournis :

```
trace_graphe_adj(G1)
trace_graphe_adj_cc(G1)
trace_graphe_adj(G2)
trace_graphe_adj_cc(G2)
```

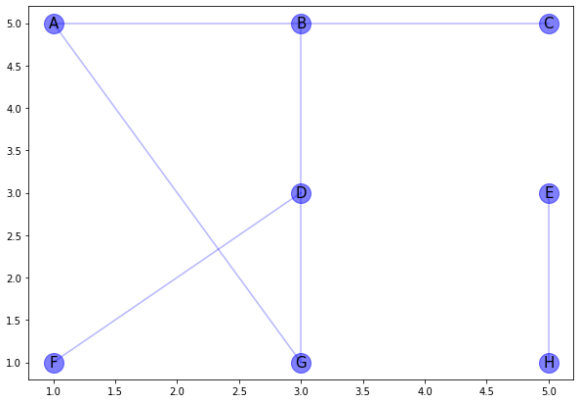

FIGURE 1 – Graphe G1

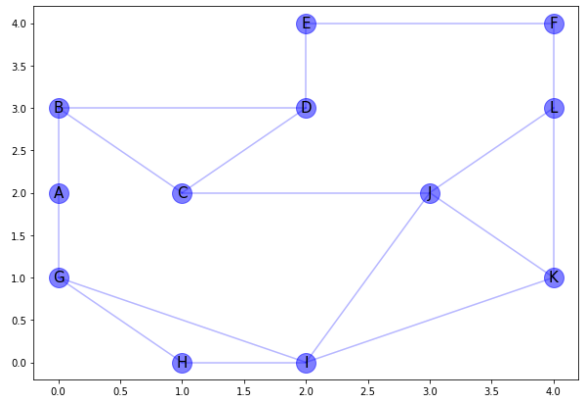

FIGURE 2 – Graphe G2

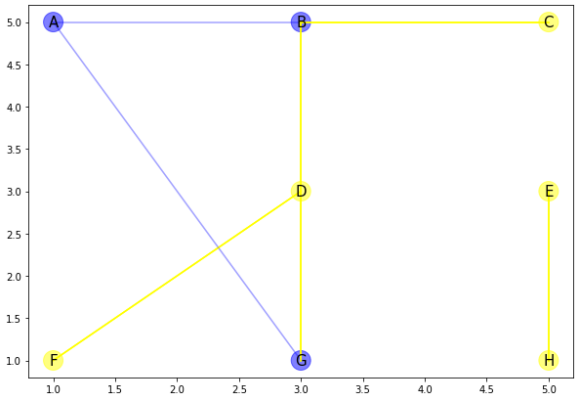

FIGURE 3 – Graphe G1 coloré

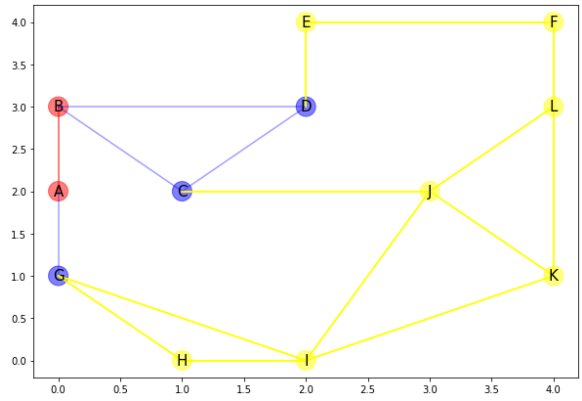

FIGURE 4 – Graphe G2 coloré

1.2 Visualisation du parcours en largeur (ou BFS, pour Breadth First Search)

Exercice 12.2

1. Tous les sommets sont d'abord colorés en JAUNE à l'initialisation du parcours. Lors de l'exploration, lorsqu'un sommet est marqué comme `"vu"`, il suffit d'ajouter sa coloration en `BLEU` et la commande de visualisation du graphe coloré. Lorsque toute l'adjacence d'un sommet a été visitée, il est coloré en `ROUGE` et la commande de visualisation est de nouveau ajoutée. Le code suivant est donc obtenu :

```
from copy import deepcopy

def init_parcours(G):
    # copie « profonde » du dictionnaire
    P = {s:deepcopy(G[s]) for s in G}
    # initialisation des sommets comme non marqués
    for s in P:
        P[s]["vu"]=0
        P[s]["coul"]=JAUNE
    return(P)

def exploration_composante_vis(G,r):
    G[r]["vu"]=1
    G[r]["coul"]=BLEU
    trace_graphe_adj_cc(G)
    E = [r]
    F = deque([r])
    while len(F)!=0:
        v = F.popleft()
        for s in G[v]["adj"]:
            if G[s]["vu"]==0:
                G[s]["vu"]=1
                G[s]["coul"]=BLEU
                trace_graphe_adj_cc(G)
                E.append(s)
                F.append(s)
        print(F)
        G[v]["coul"]=ROUGE
        trace_graphe_adj_cc(G)
    return(E)
```

2. Les commandes suivantes permettent d'explorer toute la composante connexe du sommet `A` du graphe `G1`.

```
P1 = init_parcours(G1)
exploration_composante_vis(P1,"A")
```

3. De même, les commandes suivantes permettent d'explorer toute la composante connexe du sommet du sommet `A` du graphe `G2`.

```
P2 = init_parcours(G2)
exploration_composante_vis(P2,"A")
```

4. Désormais la sortie de boucle doit s'effectuer dès que le sommet d'arrivée est marqué comme `vu` ou que la pile est vide. Si le sommet d'arrivée n'est pas marqué comme `vu` en sortie de boucle, c'est qu'il n'appartient pas à la composante connexe de `sa`. Une assertion doit détecter ce type d'erreur. D'où le code suivant :

```
def bfs2(G,sd,sa):
    G[sd]["vu"]=1
    E = [sd]
    F = deque([sd])
    while G[sa]["vu"]!=1 and len(F)!=0:
        v = F.popleft()
        for s in G[v]["adj"]:
            if G[s]["vu"]==0:
                G[s]["vu"]=1
                G[s]["p"]=v
                E.append(s)
                F.append(s)
        print(F)
    assert G[sa]["vu"]==1,
    "Le sommet d'arrivée "+sa+" n'appartient pas àla composante connexe de "+sd
    return(E)
```

5. Il suffit de remonter les sommets parents du sommet d'arrivée au sommet de départ puis de renvoyer la liste inversée. Le code suivant convient :

```
def chemin(P,sd,sa):
    ch = [sa]
    while ch[-1] != sd:
        ch.append(P[ch[-1]]["p"])
    return ch[::-1]
```

6. Fonction testée grâce aux commandes :

```
# Parcours et chemin dans G1 de A à F
P1 = init_parcours(G1)
bfs2(P1,"A","F")
print(chemin(P1,"A","F"))

# Parcours et chemin dans G2 de A à L
P2 = init_parcours(G2)
bfs2(P2,"A","L")
print(chemin(P2,"A","L"))
```

7. La fonction `exploration_composante` est modifiée légèrement : le graphe est parcouru en largeur à partir d'un sommet quelconque du dictionnaire. La longueur de la liste des sommets parcourus est égale à la longueur du dictionnaire (implémentant le graphe) pour un graphe connexe. Le graphe n'est pas connexe dans le cas contraire. D'où le code :

```
def estconnexe(G):
    c1 = list(G.keys())[0] #c1 première clé de la liste des clés du dictionnaire
    E = [c1]
    F = deque([c1])
    while len(F)!=0:
        v = F.popleft()
        for s in G[v]["adj"]:
            if G[s]["vu"]==0:
                G[s]["vu"]=1
                E.append(s)
                F.append(s)
    return(len(E)==len(G))
```

Il est aussi possible de définir un compteur initialisé à `n=len(G)-1`, de lui enlever 1 à chaque nouveau sommet découvert puis de renvoyer le test de fin de parcours de composante connexe `n==0` ; soit :

```
from copy import deepcopy

def init_parcours(G):
    P = {s:deepcopy(G[s]) for s in G}
    for s in P:
        P[s]["vu"]=0
        P[s]["p"]=None
    return(P)

def estconnexe(G):
        P = init_parcours(G2)
        d = list(G.keys())[0]
        P[d]["vu"]=1
        F = deque([d])
        n = len(G)-1
        while len(F)>0:
                v = F.popleft()
                for s in P[v]["adj"]:
                if P[s]["vu"]==0:
                P[s]["vu"]=1
                n-=1
                F.append(s)
        return(n==0)
```

Le test de vérification peut s'effectuer grâce à la fonction suivante :

```
def test_ec():
    P1 = init_parcours(G1)
    P2 = init_parcours(G2)
    estconnexe(P1) == False
    estconnexe(P2) == True
    print("Tests réussis")
```

On obtient :

```
>>> test_ec():
Tests réussis
```

8. La méthode est celle donnée dans le cours. Une implémentation possible :

```
def exploration_composante(G,r):
    G[r]["vu"]=1
    E = [r]
    F = deque([r])
    while len(F)!=0:
        v = F.popleft()
        for s in G[v]["adj"]:
            if G[s]["vu"]==0:
                G[s]["vu"]=1
                E.append(s)
                F.append(s)
    return(E)
```

```
def compcon(G):
    """
    Parameters
    ----------
    G : dictionnaire (graphe).

    Returns
    -------
    Dictionnaire :
    Clés : chaines de caractère
    Valeurs : Composantes connexes du graphe G (liste de sommets de G).

    """
    i,cc = 1,{}
    for s in G:
        if G[s]["vu"]==0:
            st = "Composante connexe n° " + str(i)
            cc[st]=exploration_composante(G, s)
            i+=1
    return(cc)
```

La version récursive de l'exploration est un peu "artificielle", mais assez intéressante puisque cela oblige à être clair sur **le variant** (longueur de la file et sommets non marqués) et **l'invariant** (principe de l'exploration). Le code suivant convient :

```
def ec_rec(P,E,F):
        v = F.popleft()
        for s in P[v]["adj"]:
                if P[s]["vu"]==0:
                        P[s]["vu"]=1
                        E.append(s)
                        F.append(s)
                if len(F)==0:
                        return(E)
                else:
                        return(ec(P,E,F))
```

9. Le test de la fonction `Compcon` peut s'effectuer de la façon suivante :

```
# Test sur le graphe G1
P11 = init_parcours(G1)
CC11 = Compcon(P11)
print(CC11)

# Test sur le graphe G2
P2 = init_parcours(G2)
CC2 = Compcon(P2)
print(CC2)

# Test sur le graphes G1, après ajout du sommet I
sommet(G1,"I")
P12 = init_parcours(G1)
CC12 = Compcon(P12)
print(CC12)
```

Sorties obtenues :

```
{'Composante connexe n° 1': ['A', 'B', 'G', 'C', 'D', 'F'],
'Composante connexe n° 2': ['E', 'H']}
{'Composante connexe n° 1': ['A', 'B', 'G', 'C', 'D', 'H', 'I', 'J', 'E', 'K', 'L', 'F']}
{'Composante connexe n° 1': ['A', 'B', 'G', 'C', 'D', 'F'],
'Composante connexe n° 2': ['E', 'H'], 'Composante connexe n° 3': ['I']}
```

1.3 Parcours en profondeur (ou DFS, pour Depth First Search)

Exercice 12.3

1. Le pseudo-code de l'algorithme est donné dans le cours. La commande `break` est ici très utile...Par ailleurs, tous les sommets sont d'abord colorés en JAUNE à l'initialisation du parcours. Lors de l'exploration, lorsqu'un sommet est marqué comme `"vu"`, il suffit d'ajouter sa coloration en `BLEU` et la commande de visualisation du graphe coloré.

```
from copy import deepcopy

def init_parcours_c(G):
    # copie « profonde » du dictionnaire
    P = {s:deepcopy(G[s]) for s in G}
    # initialisation des sommets comme non marqués
    for s in P:
        P[s]["vu"]=0
        P[s]["coul"]=JAUNE
    return(P)

def dfs_coul(G,sd):
    if not(sd in G):
        return None
    G[sd]["vu"]= 1
    G[sd]["coul"]=BLEU
    pile=deque([sd])
    trace_graphe_adj_cc(G)
    while len(pile)!=0:
        v=pile[-1]
        c=0
        for s in G[v]["adj"]:
            if G[s]["vu"]==0:
                G[s]["vu"]=1
                G[s]["coul"]=BLEU
                pile.append(s)
                trace_graphe_adj_cc(G)
                c+=1
                break
        if c==0:
            pile.pop()
```

Remarque : la coloration en ROUGE des sommets dont toute l'adjacence a été visitée est plus complexe que lors du parcours en largeur. Sauriez-vous expliquer pourquoi ?

2. Pour renvoyer la liste des sommets parcourus, plusieurs méthodes possibles : par exemple : l'ajout à une liste (`liste_sommets` dans le code ci-dessous) des différents sommets visités successivement lors du parcours, l'ajout du sommet parent dans le dictionnaire de parcours puis la reconstitution du parcours à l'aide de la fonction `chemin`. Une implémentation possible :

```
from copy import deepcopy

def init_parcours(G):
    # copie « profonde » du dictionnaire
    P = {s:deepcopy(G[s]) for s in G}
    # initialisation des sommets comme non marqués
    for s in P:
        P[s]["vu"]=0
    return(P)
```

```
def dfs_som(G,sd):
    if not(sd in G):
        return None
    G[sd]["vu"]= 1
    pile=deque([sd])
    liste_sommets=[sd]
    while len(pile)!=0:
        v=pile[-1]
        c=0
        for s in G[v]["adj"]:
            if G[s]["vu"]==0:
                G[s]["vu"]=1
                pile.append(s)
                liste_sommets.append(s)
                c+=1
                break
        if c==0:
            pile.pop()
    return liste_sommets
```

3. Tests des fonctions sur les graphes `G1` et `G2` via les commandes :

```
# Test sur le graphes G2
P1 = init_parcours_c(G1)
LS1c = dfs_coul(P1,"A")
P1 = init_parcours(G1)
LS1s = dfs_som(P2,"A")
print(LS1s)

# Test sur le graphes G2
P2 = init_parcours_c(G2)
LS2c = dfs_coul(P2,"D")
P2 = init_parcours(G2)
LS2s = dfs_som(P2,"D")
print(LS2s)
```

4. Les sorties suivantes sont obtenues :

```
>>> LS1s
['A', 'B', 'C', 'D', 'F', 'G']

>>> LS2s
['D', 'B', 'A', 'G', 'H', 'I', 'J', 'C', 'K', 'L', 'F', 'E']
```

2 Application : un graphe de collaboration

Exercice 12.4

1. `"dErd"` est un dictionnaire d'adjacence dont les valeurs sont des listes d'adjacence. Il s'agit donc de le convertir en un dictionnaire de dictionnaires, pour lequel les valeurs des clés `"adj"` des "dictionnaires valeurs" seront les listes d'adjacence de `"dErd"`. Le code suivant convient :

```
def conversion(dadj):
    return {s:{"adj":dadj[s]} for s in dadj}
```

La commande de conversion de dErd s'écrit :

```
GErd = conversion(dErd)
```

2. La fonction `numtonom` prend en argument l'étiquette d'un sommet du graphe (un entier) et renvoie le nom du mathématicien associé. Inversement pour la fonction `nomtonum`.

```
>>> nomtonum("ERDOS,PAUL")
6926

>>> numtonom(6926)
'ERDOS,PAUL'

>>> numtonom(42)
'BOES, DUANE C.'
```

3. Soient `max_collab` et `max_collab_2` les fonctions répondant aux deux questions posées. Pour `max_collab`, une variante de la méthode à un candidat est utilisée (si plusieurs mathématiciens ont le même nombre de collaborations, le nom de chacun est ajouté à la liste). Pour `max_collab_2`, l'une des solutions consiste à réutiliser la fonction `max_collab` sur une copie du graphe dont les noms des mathématiciens ayant le plus de collaborations ont été supprimés. Une implémentation possible :

```
def max_collab(G):
    max, L = 0, []
    for s in G:
        if len(G[s]["adj"])>max:
            max = len(G[s]["adj"])
            L=[[numtonom(s)],max]
        elif len(G[s]["adj"])==max:
            L[0].append(numtonom(s))
    return L

def max_collab_2(G):
    P = {s:G[s].copy() for s in G}
    m,max = max_collab(G)
    for t in m:
        P.pop(nomtonum(t))
    return max_collab(P)
```

Résultats :

```
>>> max_collab(GErd)
[['ERDOS,PAUL'], 507]

>>> max_collab_2(GErd)
[['SZEMEREDI, ENDRE'], 63
```

4. La liste d'adjacence des sommets du graphe associés à ces chercheurs est nulle. D'où le code :

```
def sscollab(G):
    L = []
    for s in G:
        if len(G[s]["adj"])==0:
            L.append(numtonom(s))
    return [L,len(L)]
```

Sortie obtenue :

```
>>> sscollab(G)
[['Aarts, Emile H. L.', 'Abbw-Jackson, Daniel', 'Abdelbar, Ashraf M.',...
'Yacobi, Yacov', 'Yamron, Jonathan P.', 'Zemirline, Abdallah', 'Zhang, Li'],
1393]
```

5. Après simplification de la fonction `exploration_composante`, le nombre de composantes connexes est comptabilisé via la fonction `Nbcc` suivante :

```
def exploration_composante(G,r):
    G[r]["vu"]=1
    F = deque([r])
    while len(F)!=0:
        v = F.popleft()
        for s in G[v]["adj"]:
            if G[s]["vu"]==0:
                G[s]["vu"]=1
                F.append(s)

def Nbcc(G):
    i,cc = 0,{}
    for s in G:
        if G[s]["vu"]==0:
            exploration_composante(G, s)
            i+=1
    return(i)
```

Pour retirer Paul Erdős de sa composante connexe, il est possible d'écrire une fonction qui supprime le sommet associé du graphe et toutes les arêtes correspondantes...ou plus simplement de marquer le sommet associé comme `"vu"` avant le parcours des composantes connexes.
Les résultats (1394 avec Paul Erdős, 1429 sans) sont obtenus à l'aide des commandes :

```
# Nb composantes connexes avec ERDOS, PAUL
PErd = init_parcours(GErd)
print(Nbcc(PErd))

# Nb composantes connexes sans ERDOS, PAUL
PErd = init_parcours(GErd)
PErd[6926]["vu"]=1
print(Nbcc(PErd))
```

Définition d'un ***point d'articulation*** : un point d'articulation est un sommet d'un graphe non orienté qui, si on le retire du graphe, augmente le nombre de composantes connexes. Si le graphe était connexe avant de retirer ce sommet, il devient donc non connexe.

6. La fonction `ec_dist` est analogue à la fonction `distance` vue au TP n°11. Ne pas oublier de compléter l'initialisation du parcours. Le code suivant convient :

```
def init_parcours(G):
    # copie « profonde » du dictionnaire
    P = {s:deepcopy(G[s]) for s in G}
    # initialisation des sommets comme non marqués
    for s in P:
        P[s]["vu"]=0
        P[s]["d"]=0
    return(P)

def ec_dist(G,sd):
    G[sd]["vu"]=1
    dist = 0
    F = deque([sd])
    while len(F)!=0:
        v = F.popleft()
        for s in G[v]["adj"]:
            if G[s]["vu"]==0:
                G[s]["vu"]=1
                G[s]["d"]=G[v]["d"]+1
                F.append(s)
    return {s:G[s]["d"] for s in G if G[s]["vu"]==1}
```

Le test peut s'effectuer grâce aux commandes :

```
# Distance de collaboration à partir de ERDOS, PAUL
PErd = init_parcours(GErd)
ec_ep = ec_dist(PErd,6926)
print(ec_ep)
print(len(ec_ep))
```

La composante connexe de Paul Erdős est formée de 5534 mathématiciens. Analyse plus complète des résultats obtenus :

```
def dd(G):
    dd = {}
    for s in G:
        dsd=G[s]["d"]
        if dsd in dd:
            dd[dsd]+=1
        else :
            dd[dsd]=1
    return dd

>>> dd(ec_ep)
{1: 507, 2: 5026, 0: 1}
```

La composante connexe est formé de 507 collaborateurs directs (distance de collaboration égale à 1) et 5026 collaborateurs indirects, dont la distance de collaboration est égale à 2 (autrement dit ces 5026 chercheurs ont collaboré directement avec les 507 chercheurs ayant directement collaboré avec Paul Erdős).

3 Parcours best-first (s'il reste du temps)

Exercice 12.5

1. Il s'agit cette fois-ci de la recherche d'un minimum mais la méthode habituelle convient :

```
def extrait_plus_proche(P,sa):
    def distance(x,y):
        return((x[0]-y[0])**2+(x[1]-y[1])**2)
    csa = P[sa]["XY"]
    dpp = float("inf")
    for s in P:
        if P[s]["vu"]==1:
            ds = distance(P[s]["XY"],csa)
            if ds<dpp:
                pp=s
                dpp=ds
    P[pp]["vu"]=2
    return(pp)
```

2. L'algorithme s'écrit par analogie avec celui du parcours en largeur :

```
def bf(G,sd,sa):
    G[sd]["vu"]=1
    E=[]
    while G[sa]["vu"]!=1:
        v = extrait_plus_proche(G,sa)
        E.append(v)
        if len(E)>=2:
            if E[-1]==E[-2]:
                E.pop()
                G[E[-1]]["vu"]=2
        for s in G[v]["adj"]:
            if G[s]["vu"]==0:
                G[s]["vu"]=1
    E.append(sa)
    return(E)
```

3. Le test est effectué à l'aide des instructions :

```
# Parcours et chemin dans G2 de A à L
P2 = init_parcours(G2)
ch = bf(P2,"H","F")
print(ch)
```

La sortie suivante est obtenue :

```
['H', 'I', 'J', 'L', 'F']
```

4. Instruction analogue à la question précédente :

```
# Parcours et chemin dans G2 de A à L
P2 = init_parcours(G2)
ch = bf(P2,"A","L")
print(ch)
```

La sortie suivante est obtenue :

```
['A', 'B', 'D', 'E', 'F', 'L']
```

Si la ligne `P[pp]["vu"]=2` est commentée, l'algorithme ne termine plus : en effet, le passage par le sommet `"E"` nécessite un détour, ce qui est impossible si `"D"` fait toujours partie des sommets parmi lesquels le sommet le plus proche du sommet d'arrivée est extrait.

* *
*