Projet de théorie des graphes : Orientation d'un graphe non orienté en un graphe fortement connexe

Coline Trehout et Alice Gydé

17 décembre 2021

Table des matières

Table des figures

1 Introduction

L'objectif de ce projet est d'implémenter un algorithme qui permet de trouver, si une telle orientation existe, une orientation du graphe en un graphe orienté fortement connexe. Le but de ce projet consiste donc à soit trouver une arête déconnectante ou à trouver une orientation fortement connexe. Pour ce faire, nous allons effectuer le calcul des composantes 2-connexes et des composantes 2-arêtes connexes. Un graphe est dit 2-connexe s'il est connexe et n'admet pas de sommet déconnectant. Un graphe est 2-arête connexe s'il n'admet pas d'arête déconnectante.

2 Exercice 1

Dans un premier temps il est demandé d'implémenter les algorithmes de calcul de 2-connexité dus à Schmidt.

2.1 Composantes 2-arêtes connexes

Calculer les composantes 2-arêtes connexes d'un graphe.

Construction du graphe orienté

Il faut tout d'abord construire le graphe orienté H à partir du graphe de départ non orienté G. Une erreur est levée si le graphe de départ n'est pas connexe ou possède moins de 3 arêtes.

Un parcours DFS est effectué sur le graphe G à partir d'un sommet quelconque. Dans le graphe H, les arcs de la relation de parenté sont tous dirigés vers le sommet de départ. Puis les arcs arrières sont ajoutés dans l'autre sens. On obtient donc H le graphe dirigé construit à partir de G. Le graphe peut ainsi être décomposé en chaînes qui sont soit des cycles soit des chemins.

Test de 2-arête connexité

Un graphe est 2-arête connexe si et seulement si toutes ses arêtes sont contenues dans au moins une chaîne du graphe. Les chaînes sont donc parcourues une par une (dans l'ordre des sommets de date de DFS les plus petites).

La fonction `deux_aretes_connexe` prend en paramètre le graphe g de départ (non orienté), h son graphe orienté associé et D la liste des dates de départ et de fin obtenues grâce au DFS. Elle renvoie deux paramètres : `True` si le graphe est 2-arête connexe, `False` sinon et le graphe hbis qui est une copie du graphe h dans lequel on supprime au fur et à mesure les arêtes des circuits parcourus. Donc à la fin hbis ne contient plus que les arcs déconnectants.

Dans cette fonction, pour chaque sommet du graphe, on va parcourir tous les circuits dont il est le sommet de départ (s'il a des arcs sortants). S'il n'a qu'un seul voisin sortant, le parcours du cycle se fait vers ce voisin. S'il en a plusieurs, on trie les voisins dans l'ordre de date de début de DFS croissante et on parcourt les cycles dans cet ordre.

Le parcours des cycles se fait grâce à la fonction `parcours_cycle`. Cette fonction prend en argument le sommet de départ du circuit sommet_depart, le graphe orienté h, le graphe orienté modifié hbis, un voisin sortant de sommet départ d et la liste des dates de DFS D. Elle renvoie le graphe hbis dont on a enlevé les arcs appartenant à des circuits. Dans cette fonction, on parcourt les voisins jusqu'à ce qu'on retourne sur le sommet de départ, dans ce cas le circuit a été parcouru et les arcs correspondant supprimés. hbis est renvoyé à la fonction `deux_aretes_connexe`. On teste à chaque tour de boucle si le nombre d'itérations n'est pas supérieur au nombre d'arcs du graphe pour éviter les boucles infines.

La fonction renvoie `True` si à la fin du parcours de tous les circuits il ne reste plus aucun arc dans `hbis` et que le degré minimum du graphe g est supérieur ou égal à 2.

Calcul des composantes 2-arêtes connexes

Le graphe hbis renvoyé par la fonction `deux_aretes_connexe` est composé des arcs déconnectants du graphe h. Il ne comporte donc aucun arc si g est 2-arête connexe. Dans ce cas, on ne calcule pas les composantes 2-arêtes connexes puisqu'il n'y a qu'une seule composante connexe composée de tous les sommets du graphe.

Dans le cas où g n'est pas 2-arête connexe, on crée le graphe arcs_dec qui est une copie du graphe hbis. La fonction `graphe_c2ac` renvoie le graphe g_cac qui est une copie du graphe h (orienté) auquel on a retiré les arcs déconnectants. Puis la fonction `cfc` renvoie les composantes connexes du graphe g_cac sous forme d'une liste de listes des sommets appartenant à la même composante connexe. Les résultats et graphes obtenus sont affichés.

2.2 Composantes 2-connexes

Calculer les composantes 2-connexes d'un graphe.

Test de 2-connexité

Pour qu'un graphe soit 2-connexe, il doit être 2-arête connexe. Donc on teste d'abord si le graphe est 2-arête connexe grâce aux fonctions vues précédemment.

Soit C une décomposition en chaînes d'un graphe 2-arête connexe G. **Le graphe G est 2-connexe si et seulement si la chaîne C1 est le seul circuit dans le graphe.** Pour vérifier cette condition, il faut parcourir toutes les chaînes (dans l'ordre croissant) et supprimer les arêtes au fur et à mesure. La première chaîne est forcément un circuit car on a pas encore supprimé de sommet. Si toutes les autres chaînes sont des chemins, alors le graphe est 2-connexe. C'est le rôle des fonctions `deux_connexe` et `parcours_cycle2`. La fonction `deux_connexe` appelle la fonction `parcours_cycle2` pour chaque sommet et renvoie un booléen : `True` si le graphe est connexe et `False` sinon.

Calcul des composantes 2-connexes

Si le graphe est 2-connexe, il n'y a qu'une composante 2-connexe : le graphe entier. Sinon, il faut calculer ses composantes 2-connexes, c'est le rôle de la fonction `cdeuxc`.

Pour les obtenir, la fonction supprime tout d'abord les arêtes déconnectantes du graphe initial g à l'aide du graphe hbis, obtenu via la fonction `deux_aretes_connexes`. Elle cherche ensuite les sommets d'articulation grâce à la propriété suivante : le nombre de composantes connexes du graphe sans ce sommet est supérieur au nombre de composantes connexes du graphe avec ce sommet.

Une fois un sommet d'articulation trouvé, celui-ci est divisé en autant de sommets annexes qu'il y a de composantes. Finalement, la fonction réassemble les sommets des différentes composantes 2 connexes et les renvoie.

3 Exercice 2

Trouvez un orientation en un graphe fortement connexe ou trouver une arête déconnectante.

Un graphe G non orienté admet une orientation en un graphe fortement connexe si et seulement si G ne possède aucune arête déconnectante.

Soit H un graphe orienté. On suppose que le graphe sous-jacent de H, c'est-à-dire le graphe non orienté que l'on nomme G est simple et connexe. Si le graphe sous-jacent G est 2-arête connexe (donc ne possède pas d'arête déconnectante) alors il existe une décomposition en un graphe fortement connexe. On sait qu'un graphe est fortement connexe si et seulement si chaque arc est présent dans au moins un circuit de H (voir démonstration de l'exercice 3). Or cette condition est la condition testée pour vérifier qu'un graphe soit 2-arête connexe. Donc si un graphe G est 2-arête-connexe, H est fortement connexe par construction du graphe orienté telle que faite dans notre implémentation. Si le graphe admet une orientation en un graphe fortement connexe, celle-ci est affiché, sinon la listes des arêtes déconnectantes est affichée.

4 Exercice 3

Montrez qu'un graphe est fortement connexe si et seulement si chaque arc est présent dans au moins un circuit de G.

$\Longrightarrow$

On veut montrer que si un graphe est fortement connexe, alors chacun de ses arcs est présent dans au moins un circuit de G.

Par contraposée, on montre que s'il existe un arc qui n'appartient à aucun circuit de G alors G n'est pas fortement connexe. Soit (a,b) un arc orienté de a vers b qui n'appartient à aucun circuit de G. Alors par définition il n'existe aucun chemin allant de b vers a dans G. Donc G n'est pas fortement connexe.

$\Longleftarrow$

On veut montrer que si chaque arc est présent dans au moins un circuit de G, alors le graphe G est fortement connexe.

Donc il faut montrer que si chaque arc est présent dans au moins un circuit de G, alors il existe un chemin entre tout sommet x et y quelconques.

Premier cas : x et y sont dans le même circuit

Pour tout couple de sommets présents dans le même circuit, il existe un chemin de x vers y et de y vers x par définition.

Deuxième cas : x et y ne sont pas dans le même circuit

Comme chaque arc est présent dans au moins un circuit du graphe, les liens entre chaque circuit ne peuvent pas être des arcs. Deux circuits connectés entre eux sont donc liés par un sommet, ils ont donc au moins un sommet en commun. Pour relier x et y, le chemin va de x au sommet "lien" dans le premier circuit (on se ramène alors au premier cas), puis il passe par ce sommet "lien" avant de rejoindre y dans cet autre circuit (premier cas). Cette démarche est également valable pour aller de y à x.

Il existe donc bien un chemin entre tous sommets x et y de G. Donc G est fortement connexe.

5 Exercice 4

Prouvez ou infirmez l'affirmation suivante : Un graphe est fortement connexe si et seulement si son graphe sous-jacent est 2-arête connexe.

Contre exemple :

Le graphe non orienté sous-jacent représenté figure 1 est 2-arête connexe. En effet, si on enlève n'importe quelle arête du graphe il reste connexe. Pourtant il n'existe pas de chemin allant par exemple du sommet 1 au sommet 2 dans le graphe orienté représenté figure 2 donc ce graphe n'est pas fortement connexe. Donc l'affirmation est fausse.

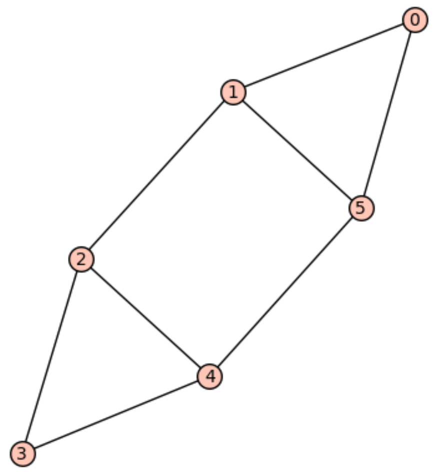

FIGURE 1 – Exemple d'un graphe 2-arête connexe

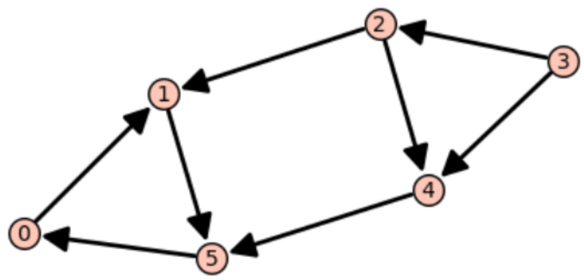

FIGURE 2 – Exemple d'un graphe orienté issu du graphe de la figure 1 non fortement connexe

6 Conclusion

Ce projet présente une implémentation en sagemath du calcul des composantes 2-arêtes connexes et 2-connexes d'un graphe non orienté. Il permet également de déterminer une orientation en un graphe fortement connexe à partir d'un graphe non orienté 2-arête connexe. Enfin, il démontre ou infirme certaines propriétés.

Annexe

1 Code sagemath

```
#PROJET DE GRAPHE ALICE GYDÉ ET COLINE TREHOUT
import sys
from sage.graphs.connectivity import connected_components

#couleur blanc = 0, gris = 1, noir = 2

#PARCOURS EN PROFONDEUR SUR GRAPHE NON ORIENTE
def parcours_prof(g):
    V = g.vertices()
    P = [] #parent
    D = [] #couple de dates de chaque sommet
    C = [] #couleur
    date = [] #valeur date qui va être incrémentée
    date.append(0) #initialisation à 0
    for i in V:
        D.insert(i,[0,0]) #initialisation à 0 pour toutes les dates
        P.insert(i,-1) #initialisation des parents à -1
        C.insert(i,0) #tous les sommets blancs
    for i in g.vertices():
        if C[i] == 0 : #si le sommet est blanc
            visiter_pp(g,i,P,D,C,date)
    return P,D

#VISITER PARCOURS EN PROFONDEUR SUR GRAPHE NON ORIENTÉ
def visiter_pp(g,u,P,D,C,date):
    C[u] = 1 #sommet gris
    date[0] = date[0]+1
    D[u][0] = date[0] #date de début
    for w in g.neighbors(u) :
            if C[w] == 0 :
                P[w] = u
                visiter_pp(g,w,P,D,C,date)
    C[u] = 2 #sommet noir
    date[0] = date[0]+1
    D[u][1] = date[0]

#PARCOURS EN PROFONDEUR SUR GRAPHE ORIENTÉ
def parcours_prof_oriente(g):
    V = g.vertices()
    P = [] #parent
    D = [] #couple de dates de chaque sommet
    C = [] #couleur
    ordre = []
    date = [] #valeur date qui va être incrémentée
    date.append(0) #initialisation à 0
    for i in V:
```

```
        D.insert(i,[0,0]) #initialisation à 0 pour toutes les dates
        P.insert(i,-1) #initialisation des parents à -1
        C.insert(i,0) #tous les sommets blancs
    for i in g.vertices():
        if C[i] == 0 : #si le sommet est blanc
            visiter_pp_oriente(g,i,P,D,C,date,ordre)
    return P,D,ordre

#VISITER PARCOURS EN PROFONDEUR SUR GRAPHE ORIENTÉ
def visiter_pp_oriente(g,u,P,D,C,date,ordre):
    C[u] = 1 #sommet gris
    date[0] = date[0]+1
    D[u][0] = date[0] #date de début
    for w in g.neighbors_out(u) :
            if C[w] == 0 :
                P[w] = u
                visiter_pp_oriente(g,w,P,D,C,date,ordre)
    C[u] = 2 #sommet noir
    date[0] = date[0]+1
    D[u][1] = date[0]
    ordre.append(u)

#PARCOURS EN PROFONDEUR DU GRAPHE TRANSPOSÉ
def parcours_proft(g,ordre):
    V = g.vertices()
    P = [] #parent
    D = [] #couple de dates de chaque sommet
    C = [] #couleur
    date = [] #valeur date qui va être incrémentée
    composante = [] #CFC
    liste_cfc = [] #liste contenant les cfc
    date.append(0) #initialisation à 0

    for i in V:
        D.insert(i,[0,0]) #initialisation à 0 pour toutes les dates
        P.insert(i,-1) #initialisation des parents à -1
        C.insert(i,0) #tous les sommets blancs

    #sommets pris dans l'ordre suffixe inverse
    ordre.reverse()
    for i in ordre:
        if C[i] == 0 : #si le sommet est blanc
            composante = visiter_ppt(g,i,P,D,C,date,composante)
            liste_cfc.append(composante.copy())
            composante.clear()
    return P,D,liste_cfc
```

```python

#VISITER PARCOURS EN PROFONDEUR DU GRAPHE TRANSPOSÉ
def visiter_ppt(g,u,P,D,C,date,composante):
    C[u] = 1 #sommet gris
    date[0] = date[0]+1
    D[u][0] = date[0] #date de début
    for w in g.neighbors_out(u) :
            if C[w] == 0 :
                P[w] = u
                visiter_ppt(g,w,P,D,C,date,composante)
    C[u] = 2 #sommet noir
    date[0] = date[0]+1
    D[u][1] = date[0]
    composante.append(u)
    return composante

#RETOURNE LA LISTE DES COMPOSANTES FORTEMENT CONNEXES
#entrée :
#g : graphe orienté
#sortie :
#retourne la liste des composantes fortement connexes de g
def cfc(g):
    P = []
    N = []
    ordre = [] #liste des sommets en ordre suffixe
    liste_cfc = [] #liste des cfc
    P,N,ordre = parcours_prof_oriente(g)
    #graphe transposé
    gt = g.reverse()
    #gt.show()
    P,N,liste_cfc = parcours_proft(gt,ordre)
    return liste_cfc


#PARCOURS D'UN CYCLE DU GRAPHE ORIENTÉ POUR 2-ARETE CONNEXE
#entrée :
#sommet_depart : sommet d'où on part pour le cycle
#h : graphe orienté
#hbis : graphe orienté sur lequel on supprime les arêtes du cyle au fur et à mesure
#d : voisin de sommet_depart
#D : dates de début et fin de tous les voisins
#sortie :
#retourne le graphe hbis modifié
def parcours_cycle(sommet_depart , h, hbis , d, D):
    #si sommet de degré 1 alors pas de cycle
    if len(h.neighbors_out(sommet_depart)) + len(h.neighbors_in(sommet_depart)) == 1:
        return hbis
```

```
    hsave = hbis.copy() #sauvegarde du graphe
    hbis.delete_edge(sommet_depart,d)
    voisins = h.neighbors_out(d)
    #print(f"voisins de {d} : {voisins}")
    iterations = 0
    while d != sommet_depart:
        #si il n'y a pas de cycle pour sommet_depart
        if len(voisins) == 0 or iterations > h.size():
            return hsave
        suivant = voisins[0]
        min = D[suivant][0]
        #recherche du voisin avec la date de départ min
        for i in voisins:
            if D[i][0] < min:
                min = D[i][0]
                suivant = i
        hbis.delete_edge(d,suivant)
        d = suivant
        voisins.clear()
        voisins = h.neighbors_out(d)
        iterations = iterations + 1
    return hbis

#TEST GRAPHE 2-ARETES CONNEXE
#entrée :
#g : le graphe non orienté
#h : le graphe orienté
#D : dates du DFS
#sortie : renvoie True si g est 2-connexe, False sinon
#ainsi que le graphe hbis constitué des arêtes qui n'appartiennent à aucun circuit
def deux_aretes_connexe(g,h,D):
    hbis = h.copy() #graphe dans lequel on enlève les arêtes quand on parcourt les cycles
    voisins = []
    s_date = [] #liste de couples [sommet voisin de sommet_depart, date de début], réinitialisé à chaque sommet_depart

    #PARCOURS DES CYCLES DE h POUR LA 2-ARETES CONNEXITÉ
    for i in range(len(h.vertices())):
        sommet_depart = hbis.vertices()[i]
        voisins = hbis.neighbors_out(sommet_depart)
        #print(f"i {i}, voisins de i : {voisins} ")
        if (len(voisins) == 1):
            #print(f"{i} n'a qu'un voisin qui est {voisins}")
            d = voisins[0]
            hbis = parcours_cycle(sommet_depart, h, hbis, d, D)
```

```
        elif (len(voisins) > 1): #plusieurs voisins, tri s_date par date croissante
            #print(f"{i} a comme voisins {voisins}")
            for j in voisins:
                if D[j][0] > D[i][0]:
                    s_date.append([j, D[j][0]])
            s_date = sorted(s_date, key = lambda x: x[1]) # tri par date de début croissante
            for d in s_date:
                hbis = parcours_cycle(sommet_depart, h, hbis, d[0], D)
            #print(f"voisins triés par date croissante : {s_date}")
        voisins.clear()
        s_date.clear()
    #si hbis n'as plus d'arêtes et que g a degré min >= 2
    if hbis.size() == 0 and min(g.degree()) >= 2:
        return True, hbis
    else:
        return False, hbis

#RENVOIE LE GRAPHE DES CC 2-ARETES CONNEXES
#entrée :
#h : graphe orienté
#arcs_dec : graphe des arcs déconnectants
#sortie : g_cac graphe des composantes 2-aretes connexes
def graphe_c2ac(h, arcs_dec):
    g_cac = h.copy() #copie graphe de départ orienté
    for edge in arcs_dec.edges():
        g_cac.delete_edge(edge)
    return g_cac

#PARCOURS D'UN CYCLE DU GRAPHE ORIENTÉ POUR 2-CONNEXE
#entrée :
#sommet_depart : sommet d'où on part pour le cycle
#hbis : graphe orienté sur lequel on supprime les arêtes du cyle au fur et à mesure
#d : premier voisin de sommet_depart
#D : dates de début et fin de tous les voisins
#*sortie : retourne le graphe hbis modifié et un booléen False si le graphe n'est pas 2 connexe,
#True si on continue de tester les autres cycles
def parcours_cycle2(sommet_depart, hbis, d, D):
    voisins = hbis.neighbors_out(d)
    hbis.delete_edge(sommet_depart,d)
    k = 0
    while d != sommet_depart:
        if (len(voisins) == 0):
            return True, hbis
```

```
        suivant = voisins[0]
        min = D[suivant][0]
        #recherche du voisin avec la date de départ min
        for i in voisins:
            if D[i][0] < min:
                min = D[i][0]
                suivant = i
        #si on est bloqué (fin du chemin)
        if D[suivant] > D[d] and k > 0:
            return True, hbis
        else :
            hbis.delete_edge(d,suivant)
            d = suivant
            voisins.clear()
            voisins = hbis.neighbors_out(d)
            k = k+1
    return False, hbis

#TEST GRAPHE 2-CONNEXE
#entrée :
#h : graphe orienté
#D : dates du DFS
#sortie :
#renvoie True si g est 2-connexe, False sinon
def deux_connexe(h, D):
    hbis = h.copy()
    voisins = []
    s_date = [] #liste de couples [sommet voisin de sommet_depart, date de début], réitialisé à chaque sommet_depart

    #parcours et suppression des aretes du premier cycle
    sommet_depart = hbis.vertices()[0]
    voisins = hbis.neighbors_out(sommet_depart)
    if (len(voisins) == 1):
        d = voisins[0]
    elif (len(voisins) > 1):
        for j in voisins:
            if D[j][0] > D[0][0]:
                s_date.append([j, D[j][0]])
        s_date = sorted(s_date, key = lambda x: x[1]) #tri par date de début croissante
        d = s_date[0][0] #d : voisin de 0 de date de départ min

    voisins = hbis.neighbors_out(d)
    hbis.delete_edge(sommet_depart,d)
    #print(f"voisins de {d} : {voisins}")
    while d != sommet_depart:
```

```
        suivant = voisins[0]
        min = D[suivant][0]
        #recherche du voisin avec la date de départ min
        for i in voisins:
            if D[i][0] < min:
                min = D[i][0]
                suivant = i
        hbis.delete_edge(d,suivant)
        d = suivant
        voisins.clear()
        voisins = hbis.neighbors_out(d)

    #s'il n'y avait qu'une chaîne dans le graphe, hbis est vide
    if len(hbis.edges()) == 0:
        return True
    else:
        connexite = True
        #parcours et suppression des aretes des autres cycles (ou chemins)
        for i in range(1, len(h.vertices())):
            sommet_depart = hbis.vertices()[i]
            voisins = hbis.neighbors_out(sommet_depart)
            if (len(voisins) == 1):
                #print(f"{i} n'a qu'un voisin qui est {voisins}")
                d = voisins[0]
                connexite, hbis = parcours_cycle2(sommet_depart, hbis, d, D)
            elif (len(voisins) > 1): #plusieurs voisins, tri s_date par date croissante
                #print(f"{i} a comme voisins {voisins}")
                for j in voisins:
                    if D[j][0] > D[i][0]:
                        s_date.append([j, D[j][0]])
                s_date = sorted(s_date, key = lambda x: x[1]) #tri par date de début croissante
                for d in s_date:
                    connexite, hbis = parcours_cycle2(sommet_depart, hbis, d[0], D)
            voisins.clear()
            s_date.clear()
            if not connexite:
                return False
        return True

#CALCUL COMPOSANTES 2-CONNEXES
#entrée :
#g : graphe d'origine
```

```
#hbis : le graphe des arêtes déconnectantes
#sortie :
#cc : liste des composantes 2-connexes
def cdeuxc(g, hbis):
    g_cc = g.copy() #graphe qui comportera les composantes connexes
    g_deco = g.copy() #graphe de base qui ne comportera plus d'arête déconnectante
    for edge in hbis.edges(): #suppression des arêtes déconnectantes
        g_deco.delete_edge(edge)
        g_cc.delete_edge(edge)

    divise = [] #chaque divise[i] est la liste des nouveaux sommets divisés correspondant au sommet d'articulation i
    cc_g_supp = []
    for i in range (g.num_verts()):
        divise.append([0])
        cc_g_supp.append(0)
    nbr_cc_base = g_cc.connected_components_number()
    for y in g_cc.vertices(): #recherche des sommets d'articulations
        g_supp = g_cc.copy() #copie temporaire du graphe
        g_supp.delete_vertex(y)
        cc_g_supp[y] = g_supp.connected_components_number()
    y = 0
    while (y <= max(g.vertices())):
        if (cc_g_supp[y] > nbr_cc_base): #si c'est un sommet d'articulation
            g_cc.delete_vertex(y) #supression du sommet
            for z in g_cc.connected_components() : #duplication du sommet
                sommet = g_cc.num_verts()+1
                g_cc.add_vertices([sommet])
                divise[y].append(sommet)
                for a in z : #re-création des arêtes
                    if g_deco.has_edge(y,a) == True:
                        g_cc.add_edges([[sommet,a]])
        y = y+1
    for y in g_cc.vertices(): #en cas de duplication inutile
        if (len(g_cc.neighbors(y)) == 0) & (y > max(g.vertices())):
            g_cc.delete_vertex(y)
    #g_cc.show()
    compo = g_cc.connected_components() #récupération des composantes connexes avec les sommets d'articulations divisés
    compo_finales = []
    for i in range (len(compo)):
```

```
        compo_finales.append([0])
    iter = 0
    for i in compo: #parcours de ces c.c.
        for y in i:
            if y > max(g.vertices()): #si le sommet choisi est un sommet divisé
                cpt = 0
                for s in divise:
                    if y in s:
                        compo_finales[iter].append(cpt) # remise au nom du sommet d'origine
                    cpt = cpt+1
            else : #si c'est un sommet d'origine
                compo_finales[iter].append(y) #ajout dans les cc
        iter = iter+1
    for i in range (len(compo_finales)): #suppression de l'initialisation à 0
        del compo_finales[i][0]
    return compo_finales


#-------------------------------------------------------
#PROGRAMME PRINCIPAL
g = Graph()

#EXEMPLES (graphes non orientés)

#exemple avec moins de 3 arêtes
#g.add_edges([[0,1],[1,2]])

#exemple non connexe
#g.add_edges([[0,1],[3,2]])

#exemples pas 2-arête connexe
#g.add_edges([[0,1],[0,2],[0,3],[1,4],[2,4],[4,5],[5,6],[5,7],[6,7],
[4,9],[4,8],[8,9],[1,2],[2,3]])
#g.add_edges([[0,1],[1,2],[2,3],[3,4]]) #chemin
#g.add_edges([[0,1],[0,2],[0,3],[0,4]]) #étoile
#g.add_edges([[0,1],[1,2],[2,3],[0,3],[1,4],[4,5]]) #cycle + chemin
g.add_edges([[0,1],[0,2],[0,3],[1,4],[2,4],[4,5],[5,6],[5,7],[6,7],
[4,9],[4,8],[8,9],[1,2],[2,3],[9,10]])

#exemples 2-arête connexe mais pas 2-connexe
#g.add_edges([[0,1],[1,2],[2,3],[2,4],[0,2],[3,4]]) #sablier
```

```
#g.add_edges([[0,1],[1,2],[2,0],[0,3],[4,5],[3,4],[5,0]]) #poisson
#g.add_edges([[0,1],[0,2],[0,3],[1,4],[2,4],[4,7],[7,8],[4,8],[4,5],
[5,6],[4,6],[1,2],[2,3]])

#exemples graphes 2-arête connexe et 2 connexe
#g.add_edges([[0,1],[1,2],[0,2],[2,3],[0,4],[3,4]]) #maison
#g.add_edges([[0,1],[1,2],[2,3],[0,3]]) #cycle C4
#g.add_edges([[0,1],[0,3],[0,2],[1,2],[1,3],[2,3]]) #clique K4
#g.add_edges([[0,1],[0,2],[1,2],[1,3],[1,4],[2,4],[2,5],[2,6],[3,4],
[4,5],[5,6],[3,7],[4,7]])
#--------------------------------------------------------

print("Graphe non orienté g :")
g.show()

if not g.is_connected():
    print("Erreur : le graphe g n'est pas connexe !")
    sys.exit(0)
if len(g.edges()) < 3:
    print("Erreur : le graphe g a moins de 3 arêtes !")
    sys.exit(0)

P = [] #parents dans l'arbre de DFS
D = [] #dates du DFS

P,D = parcours_prof(g)

#AFFICHAGE DFS
print("Parcours en profondeur")
print ("sommets :")
l = [x for x in range(len(g.vertices()))]
print (l)
print ("parents :")
print(P)
print("dates du DFS:")
print(D)

#CRÉATION DU GRAPHE ORIENTÉ h ISSU DE g
gbis = g.copy() #graphe auquel on retire les arêtes de la relation de parenté

#AJOUT DES ARCS DE LA RELATION DE PARENTÉ DANS h
h = DiGraph()
for i in range (len(P)-1):
    b = P[i+1]
```

```
    a = i+1
    h.add_edges([(a,b)])
    gbis.delete_edge([a,b])

#AJOUT DES ARCS ARRIÉRES DANS h
for j in gbis.edges():
    if D[j[0]] < D[j[1]]:
        h.add_edges([(j[0],j[1])])
    else:
        h.add_edges([(j[1],j[0])])

print("Graphe orienté h obtenu à partir de g :")
h.show()

#--------------------------------------------------------
#exercice 1 : composantes 2-aretes connexes
est_DAC, hbis = deux_aretes_connexe(g,h,D)

#SI LE GRAPHE EST 2-ARÊTES CONNEXE
if est_DAC:
    print("Le graphe g est 2-arête connexe.")

#SINON CALCUL DES COMPOSANTES 2-ARÊTES CONNEXES
else:
    print("Le graphe g n'est pas 2-arête connexe.")
    arcs_dec = hbis.copy() #arcs déconnectants

    g_cac = graphe_c2ac(h, arcs_dec)

    print("Graphe des composantes 2-arête connexes :")
    g_cac.show()
    liste_cfc = cfc(g_cac)
    print("Liste des composantes 2-arêtes connexes de g :")
    print(liste_cfc)
    print("Graphe équivalent au graphe des composantes 2-arêtes connexes non orienté :")
    g_cac_undirected = g_cac.to_undirected()
    g_cac_undirected.show()

#--------------------------------------------------------
#exercice 1 : composantes 2-connexes
print(f"Le degré min de g est {min(g.degree())}.")

#si g n'est pas 2-connexe
if min(g.degree()) < 2 or not est_DAC or not deux_connexe(h,D):
    print("Le graphe g n'est pas 2-connexe.")
    cc = cdeuxc(g, hbis)
```

```
    print("Les composantes 2-connexes du graphe g sont les
   suivantes :")
    print(cc)
else:
    print("Le graphe g est 2-connexe.")

#-------------------------------------------------------
#exercice 2 : orientation en un graphe fortement connexe
if est_DAC:
    print("Orientation en un graphe fortement connexe :")
    h.show()
else:
    print(f"arête(s) déconnectante(s) : {arcs_dec.edges()}")

#vérification avec les fonctions de sagemath
#print(f"Le graphe g est {g.edge_connectivity()}-arête
   connexe.")
#print(f"Le graphe g est 2-connexe ? {g.is_biconnected()}.")
```