Parcours de graphes et applications

INFORMATIQUE COMMUNE - TP n° 2.5 - Olivier Reynet

À la fin de ce chapitre, je sais :

- ☞ parcourir un graphe en largeur pour détécter des boucles dans un graphe non orienté
- ☞ parcourir un graphe en largeur pour calculer les plus courts chemins
- ☞ parcourir un graphe en profondeur pour détécter des boucles dans un graphe orienté
- ☞ parcourir un graphe pour le colorer

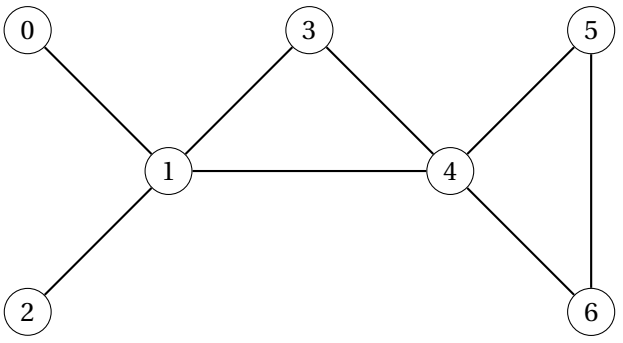
A Parcours en largeur d'un graphe

Soit le graphe non orienté dit du *passe-partout* défini par la liste d'adjacence :

```
passe_partout = [[1], [0, 2, 3, 4], [1], [1, 4], [1, 3, 5, 6], [4, 6], [4, 5]]
```

A1. Dessiner le graphe du passe-partout.

Solution :



A2. Proposer un parcours en largeur du passe-partout à partir du sommet 0.

Solution : `[0,1,2,3,4,5,6]`

Le code suivant implémente en partie le parcours en largeur d'abord dans un graphe. Il utilise la bibliothèque `Queue` qui implémente le type file d'attente (cf. figure 1). Il s'agit d'une structure de données de type First In First Out (FIFO) optimisée [1] pour que les opérations ENFILER (`put`) et DÉFILER (`get`) soient des opérations élémentaires en $O(1)$. Les opérations s'effectuent ainsi :

1. Il possible d'implémenter une file d'attente avec une liste Python. Néanmoins l'opération `pop(0)` qui permet de défiler n'est pas en $O(1)$ mais en $O(n)$ si la longueur de la file d'attente est n.

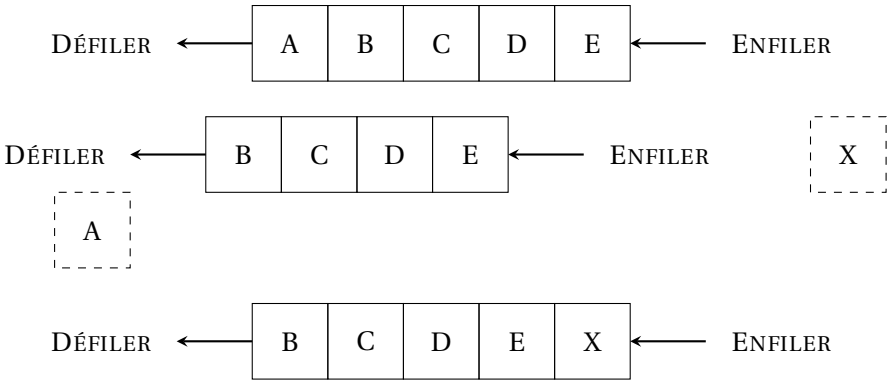


FIGURE 1 – Illustration des opérations défiler puis enfiler sur une file d'attente

```
file = queue.Queue() # création d'une file d'attente
file.put(s) # Enfiler s
u = file.get() # Défiler
file.empty() # Tester si la file est vide.
```

Le parcours en largeur peut alors s'écrire :

```
import queue

def parcours_largeur(g, depart):
    file = queue.Queue()
    decouverts = [False for _ in range(len(g))]
    parcours = []
    file.put(depart)
    decouverts[depart] = True
    while # CONDITION :
        u = file.get()  # Défiler en O(1)
        parcours.append(u)
        # POUR CHAQUE VOISIN x DE U:
            if not decouverts[x]:
                decouverts[x] = True
                file.put(x)  # Enfiler en O(1)
    return parcours
```

A3. Compléter le code du parcours en largeur au niveau de `# CONDITION :` et de `# POUR CHAQUE VOISIN x DE U:`

Solution :

```
def parcours_largeur(g, depart):
    file = queue.Queue()
    decouverts = [False for _ in range(len(g))]
    parcours = []
    file.put(depart)
    decouverts[depart] = True
    while not file.empty():
        u = file.get()  # O(1)
        parcours.append(u)
```

```
            for x in g[u]:
                if not decouverts[x]:
                    decouverts[x] = True
                    file.put(x)
        return parcours
```

A4. Quels sont les parcours en largeur que procure la fonction `parcours_largeur` à partir du sommet 0 et à partir du sommet 3 du passe-partout ?

Solution :

```
[0, 1, 2, 3, 4, 5, 6] # depuis 0
[3, 1, 4, 0, 2, 5, 6] # depuis 3
```

B Plus court chemin dans un graphe non pondéré

On cherche maintenant à connaître le chemin le plus court pour aller de 0 à 5 dans le passe-partout. Dans ce graphe non orienté et non pondéré, on peut considérer que la distance entre tous les sommets voisins vaut 1. Le parcours en largeur découvre les sommets dans l'ordre des distances depuis le sommet de départ : d'abord les voisins directs à une distance de 1, puis les voisins des voisins à une distance de 2 et ainsi de suite.

R En conséquence, la première fois qu'on découvre un sommet à l'aide du parcours en largeur, c'est forcément par le chemin le plus court en termes de nombre d'arêtes franchies.

■ **Définition 1 — Parent sur le chemin.** Sommet par lequel on a découvert un autre sommet lors d'un parcours en largeur d'un graphe.

On suppose qu'à l'issue d'un parcours en largeur d'un graphe, on dispose d'un tableau nommé `parents` qui comporte autant d'éléments que le graphe a de sommets et **qui mémorise, à chaque fois qu'on découvre un sommet, son parent sur le chemin**. Le sommet de départ est son propre parent.

Par exemple, si parents vaut `[0, 0, 1, 1, 1, 4, 4]`, cela signifie que pour aller de 0 à 5, il faut emprunter le chemin `[0,1,4,5]`. **En partant du sommet d'arrivée, on remonte ainsi le chemin suivi de proche en proche.**

B5. Écrire une fonction de signature `parents_vers_chemin(parents, s)` qui renvoie le chemin à suivre dans le graphe depuis le sommet de départ pour arriver au sommet `s`. On rappelle que le sommet de départ est son propre parent.

Solution :

```
def parents_vers_chemin(parents, s):
    chemin = []
    while parents[s] != s:
```

```
            chemin.append(s)
            s = parents[s]
        chemin.reverse()
        return chemin
```

B6. Modifier le code du parcours en largeur pour créer une fonction de signature `pcc(g,start, stop)` qui renvoie le plus court chemin de `start` à `stop` sous la forme d'une liste de sommets traversés. Si le chemin n'existe pas, la fonction renvoie `None`. On utilisera un tableau `parents` et la fonction précédente.

Solution :

```
def ppc(g,start, stop):
    file = queue.Queue()
    decouverts = [False for _ in range(len(g))]
    file.put(start)
    decouverts[start] = True
    parents = [-1 for _ in range(len(g))]
    parents[start] = start
    while not file.empty():
        u = file.get()  # O(1)
        if u == stop: # Early Exit
            return parents_vers_chemin(parents, u)
        for x in g[u]:
            if not decouverts[x]:
                decouverts[x] = True
                file.put(x)
                parents[x] = u
    return None # On n'a jamais rencontré le sommet d'arrivée...
```

C Détecter des cycles dans un graphe non orienté

On chercher maintenant à détecter des cycles dans le graphe du passe-partout.

C7. Modifier le code du parcours en largeur pour créer une fonction de signature `cycle(g, depart)->bool` qui renvoie `True` si le graphe possède un cycle et `False` sinon. On utlisera un tableau `parents` pour bien mémoriser le sommet par lequel on a découvert un sommet. **Si on redécouvre un sommet depuis un autre parent que celui mémorisé, c'est qu'il existe un cycle dans le graphe.**

Solution :

```
def cycle(g, depart):
    file = queue.Queue()
    decouverts = [False for _ in range(len(g))]
    parcours = []
    file.put(depart)
    decouverts[depart] = True
    parents = [-1 for _ in range(len(g))]
```

```
        parents[depart] = depart
        while not file.empty():
            u = file.get()  # O(1)
            parcours.append(u)
            for x in g[u]:
                if not decouverts[x]:
                    decouverts[x] = True
                    parents[x] = u
                    file.put(x)
                else:
                    if x != parents[u] and parents[x] != u:
                        print("Présence d'un cycle !", parcours + [x], "Sommet "
                            , x, " découvert par ", parents[x],
                              " et redécouvert par ", u)
                        return True
        return False
```

D Graphe biparti et bicolorable

On peut montrer qu'un graphe biparti est un graphe bicolorable.

D8. En modifiant le parcours en largeur, écrire une fonction de signature `bicolorable(g)-> bool` qui vérfie si un graphe est biparti.

Solution : Il suffit de tester sa bicolorabilité.

```
    def bicolorable(g):
        decouverts = [False for _ in range(len(g))]
        couleurs = [ -1 for _ in range(len(g))]
        while not all(decouverts):
            i = 0
            while i < len(decouverts) and decouverts[i]:
                i += 1
            file = queue.Queue()
            file.put(i)
            decouverts[i] = True
            couleurs[i] = ORANGE
            while not file.empty():
                u = file.get()  # O(1)
                c = BLACK if couleurs[u] == ORANGE else BLACK
                for x in g[u]:
                    if not decouverts[x]:
                        decouverts[x] = True
                        couleurs[x] = c
                        file.put(x)
                    elif couleurs[x] == c:
                        return False
        return True
```

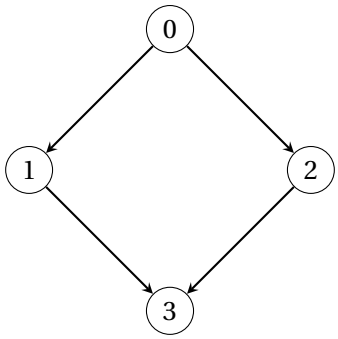


FIGURE 2 – Dans un graphe orienté, on peut très bien parcourir les sommets en largeur et redécouvrir un sommet déjà découvert sans pour autant qu'il exite un cycle. C'est le cas pour le sommet 3 sur ce graphe : depuis 0, il peut être d'abord découvert par 1 puis par 2. Pourtant il n'existe pas de cycle entre ces sommets.

E Détecter des boucles dans un graphe orienté

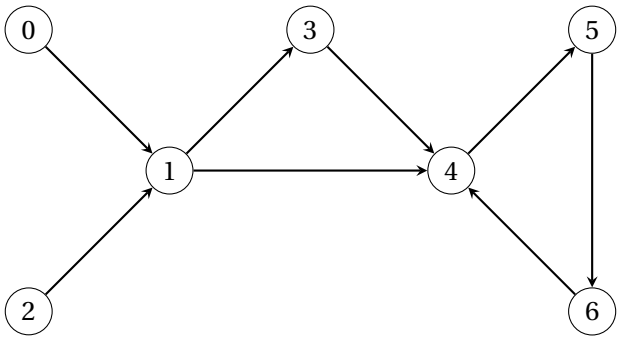
Dans le cas d'un graphe orienté, le parcours en largeur et la notion de parent d'un sommet est peu adaptée à la détection de boucles. En effet, on peut très bien redécouvrir un sommet par un autre parent sans pour autant qu'il existe un cycle, comme le montre la figure 2. C'est pourquoi on utilise dans ce cas un parcours en profondeur et un système de marquage des sommets.

Soit le graphe orienté dit du *passe-partout orienté* défini par la liste d'adjacence :

```
passe_partout = [[1], [3, 4], [1], [4], [5], [6], [4]]
```

E9. Dessiner le graphe du passe-partout orienté.

Solution :



E10. Proposer un parcours en profondeur du passe-partout à partir du sommet 0. Que remarquez-vous ?

Solution : `[0,1,4,5,6,3]` Certains sommets peuvent ne pas être découverts, comme le 2 par exemple. Il faudrait donc, pour parcourir totalement le graphe, lancer le parcours en profondeur à partir de **tous** les sommets.

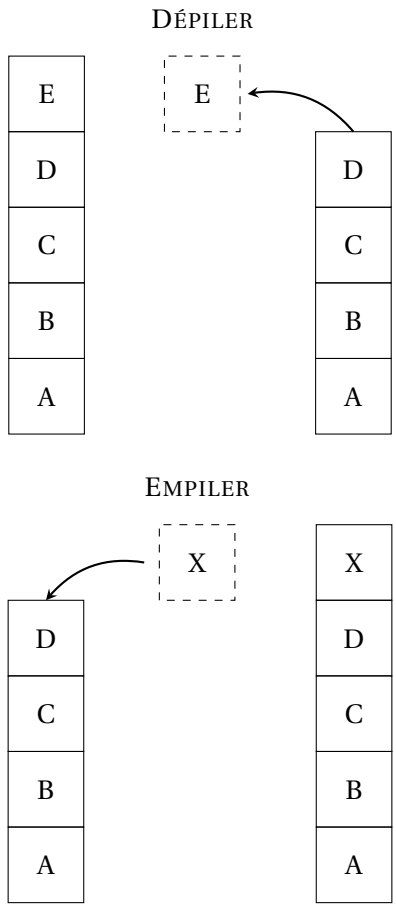


FIGURE 3 – Une pile avant et après les opérations de défilement et d'empilement.

Le code suivant implémente partiellement le parcours en profondeur d'un graphe donné sous la forme d'une liste d'adjacence. Ce code utilise un pile, une structure de données de type Last In First Out (LIFO) dont les deux opérations principales DÉPILER et EMPILER sont décrites sur la figure 3. Cette pile est réalisée par une liste Python à l'aide des méthodes `pop` et `append`.

```
def parcours_prof(g, s):
    pile = []  # liste utilisée comme une pile
    decouverts = [False for _ in range(len(g))]
    parcours = []
    pile.append(s)
    decouverts[s] = True
    while len(pile) > 0:
        u = # DEPILER
        parcours.append(u)
        # POUR CHAQUE VOISIN DE u :
            if not decouverts[x]:
                decouverts[x] = True
                pile.append(x)
    return parcours
```

E11. Compléter le code du parcours en profondeur.

Solution :

```
def parcours_prof(g, s):
    pile = []  # used as stack
    decouverts = [False for _ in range(len(g))]
    parcours = []
    pile.append(s)
    decouverts[s] = True
    while len(pile) > 0:
        u = pile.pop()
        parcours.append(u)
        for x in g[u]:
            if not decouverts[x]:
                decouverts[x] = True
                pile.append(x)
    return parcours
```

E12. Modifier le code du parcours en profondeur afin de détecter les cycles dans un graphe orienté. On procèdera comme suit :

1. On répète un parcours en profondeur à partir de chaque sommet du graphe.
2. On utilise trois marques pour mémorisé l'état des sommets du graphe `NON_VISITE`, `EN_COURS_DE_VISITE`, `VISITE`.
3. Lors d'un parcours en profondeur, on marque les sommets `EN_COURS_DE_VISITE` et lorsqu'on a terminé le parcours à partir d'un sommet, celui-ci est marqué `VISITE`.

On pourra se servir des constantes suivantes :

```
NON_VISITE = 0
EN_COURS_DE_VISITE = 1
VISITE = 2
```

Solution :

```
NON_VISITE = 0
EN_COURS_DE_VISITE = 1
VISITE = 2

def cycle(g):
    pile = []  # used as stack
    etats = [NON_VISITE for _ in range(len(g))]
    for sommet in range(len(g)-1, -1, -1):
        pile.append(sommet)
        etats[sommet] = EN_COURS_DE_VISITE
        while len(pile) > 0:
            u = pile.pop()
            for x in g[u]:
                if etats[x] == NON_VISITE:
                    etats[x] = EN_COURS_DE_VISITE
                    pile.append(x)
                elif etats[x] == EN_COURS_DE_VISITE:
                    print("Présence d'un cycle ! Sommet ", x, " en cours de
                        visite et redécouvert par ", u)
```

```
                    return True
            etats[sommet] = VISITE
            return False
```