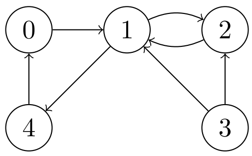
1 Parcours en profondeur

Exercice 1.
Pour chaque sommet du graphe ci-dessous, dérouler à la main le parcours en profondeur. Donner à chaque fois la valeur finale de la liste `vus`.



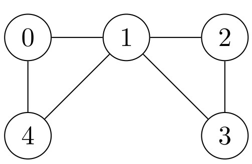
Exercice 2.
On considère la fonction `mystere` ci-dessous.

```python
def mystere(g, u, v):
  vus = []
  parcours_profondeur(g, vus, u)
  return v in vus
```

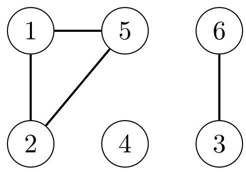
On suppose que `g` est le graphe de l'exercice précédent.

1. Quelle est le resultat de `mystere(g, 0, 4)` ?
2. Même question pour `mystere(g, 0, 3)`.
3. Décriver en une phrase le résultat de `mystere(g, u, v)` pour `u` et `v` deux sommets de `g`.

Exercice 3.
On peut se servir d'un parcours en profondeur pour déterminer si un graphe *non orienté* est *connexe*, c'est à dire si tous ses sommets sont reliés entre eux par des chemins.



Graphe connexe



Graphe non connexe

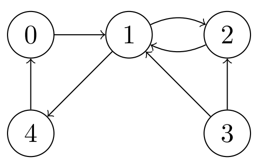
Pour cela, il suffit de faire un parcours en profondeur et de vérifier que tous les sommets ont bien été visités par ce parcours.
En utilisant de la méthode `sommets` de la classe `Graphe` et de la fonction `parcours_profondeur` , écrire écrire une fonction `est_connexe` qui réalise cet algorithme.

Exercice 4.
Dans cet exercice, on se propose d'utiliser un parcours en profondeur pour *construire* un chemin entre deux sommets lorsque c'est possible.
On le fait avec deux fonctions :

1. La fonction `parcours_chemin` est très semblable à la fonction `parcours_profondeur`. Elle prend un paramètre supplémentaire `org` (pour origine) qui est le sommet qui permis d'atteindre `s` et l'argument `vus` n'est plus une liste mais un *dictionnaire* qui associe à chaque sommet visité le sommet qui a permis de l'atteindre.

   Pour le graphe ci-contre, un tel parcours d'origine le sommet 1 donne le dictionnaire `vus` suivant :

   `{1 : None ,2 : 1 , 4 : 1, 0 : 4}`

   (a) Pour le graphe ci-dessus, déterminer le dictionnaire `vus` obtenu par `parcours_chemin` avec pour origine le sommet 2.

(b) Compléter le programme ci-dessous à fin qu'il réalise ce parcours.

```python
def parcours_chemin(g, vus, org, s):
    '''parcours depuis le sommet s, en venant de org'''
    if s not in vus:
        vus[s] = ???
        for v in g.voisins(s):
            ???
```

2. La deuxième fonction `chemin(g, u, v)` permet de construire le chemin entre `u` et `v` s'il existe. Pour cela on «remonte» le dictionnaire `vus` obtenu avec `parcours_chemin`.
   Par exemple, si l'on veut trouver un chemin du sommet `1` au sommet `0` et que notre dictionnaire obtenu à partir d'un parcours d'origine le sommet `1` est `{1 : None ,2 : 1 , 4 : 1, 0 : 4}`, «remonter» le dictionnaire donnera `0 4 1 None`, ce qui donne le chemin `1 -> 4 -> 0`.
   Compléter la fonction ci-dessous pour qu'elle réalise cet algorithme.

```python
def chemin(g, u, v):
    '''un chemin de u a v, le cas echeant, None sinon'''
    vus = {}
    parcours_chemin(g, vus, None, u)
    # s'il n'existe pas de chemin
    if ???
        return None
    # sinon on construit le chemin
    ch = []
    s = v
    while ???
        ch.append(s)
        s = ???
    ch.reverse()
    return ch
```

Exercice 5.
On considère que l'on travaille sur un graphe *non orienté*.

1. Expliquer comment un parcours en profondeur permet de détecter la présence d'un circuit dans ce graphe.
2. Ecrire, en modifiant le parcours en profondeur de la classe `Graphe`, une méthode `existe_circuit` qui renvoie un booléen indiquant s'il y a un cycle dans ce graphe.

Exercice 6.
Un parcours en profondeur permet de déterminer s'il existe un cycle dans un graphe donné. Si, lors de ce parcours, le sommet qui vient d'être découvert a un voisin en cours de traitement (gris), c'est qu'un cycle existe.
Compléter le programme suivant de sorte qu'il renvoie un booléen indiquant s'il y a un cycle dans le graphe *orienté* `g`.

```python
def parcours_cycle(g, s, couleur):
    '''parcours en profondeur depuis le sommet s'''
    couleur[s] = 'gris'
    for v in g.voisins(s):
        if ???
            return True
        if couleur[v] == 'blanc':
```

```
            if ???
              return True
        couleur[s] = 'noir'
        return False

def existe_cycle(g):
    '''determine la precense d'un cycle dans le graphe g'''
    couleur = {}
    for s in g.sommets():
        couleur[s] = 'blanc'
    for s in g.sommets():
        if couleur[s] = 'blanc':
            #si le parcours depuis le sommet s detecte un cycle
            if ???
                return True
    return False
```

Exercice 7.
Reprendre les exercices 3, 4, 5 et 6 en donnant une solution itérative à chaque problème à l'aide d'une structure de pile.

2 Parcours en largeur

Exercice 8.
Reprendre l'exercice 1 mais avec le parcours en largeur.

Exercice 9.
Un arbre binaire peut être vu comme un graphe non orienté. Le parcours en profondeur correspond alors au parcours préfixe.
Ecire une fonction `largeur(arb)` qui prend un arbre binaire `arb` en argument et affiche les valeurs de ses noeuds dans un ordre donné par un parcours en largeur.
Pour cela adapter le programme de parcours en largeur du cours.

Exercice 10.
Cet exercice reprend l'exercice 4 mais on se propose cette fois d'utiliser un parcours en largeur pour *construire* un chemin entre deux sommets lorsque c'est possible.
On utilise encore une fois deux fonctions :

1. La fonction `parcours_largeur_ch` qui est très semblable à la fonction `parcours_largeur`, l'argument `vus` n'est plus une liste mais un *dictionnaire* qui associe à chaque sommet visité le sommet qui a permis de l'atteindre.
   Compléter le programme ci-dessous à fin qu'il réalise ce parcours.

   ```
   def parcours_largeur_ch(g, s):
       vus = {}
       vus[s] = None
       ...
   ```

2. La deuxième fonction `chemin(g, u, v)` permet de construire le chemin entre `u` et `v` s'il existe. Pour cela on «remonte» le dictionnaire `vus` obtenu avec `parcours_largeur_ch`.
   Compléter la fonction ci-dessous pour qu'elle réalise cet algorithme.

   ```
   def chemin(g, u, v):
           '''un chemin de u a v, le cas echeant, None sinon'''
       ...
   ```

Exercice 11.
Le parcours en largeur permet de déterminer la *distance* entre deux sommets s_1 et s_2 d'un graphe, c'est à dire le nombre minimal d'arcs à emprunter pour aller de s_1 à s_2.
La fonction `parcours_distance` est très semblable à la fonction `parcours_largeur`. La liste `vus` est remplacer par un dictionnaire `dist` des distances du sommet `s` aux autres sommets accessibles du graphe `g`. Ce dictionnaire contient initialement le sommet `s` avec une distance `0` et à chaque fois qu'un sommet `v` est découvert depuis le sommet `u`, la distance de `s` à `v` est égale à la distance de `s` à `u` plus 1 pour l'arc que l'on vient d'emprunter.
Compléter la fonction Python ci-dessous de manière à ce qu'elle réalise ce parcours.

```
def parcours_distance(g, s):
  '''dictionnaire des distances entre s et les autres sommets
  accessibles de g'''
    dist = {}
    ???
    file = File()
    file.ajouter(s)
    while not file.est_vide() :
        u = file.consulter()
        for v in g.voisins(u):
            if ???:
                ???
                file.ajouter(v)
        file.retirer()
    return dist
```

Compléter alors la fonction `distance(g, u, v)` qui donne la distance entre les sommets `u` et `v` si un chemin entre ces deux sommets existe et `None` sinon.

```
def distance(g, u, v):
  '''distance de u a v et None si pas de chemin'''
  dist = parcours_distance(g, u)
  if ???
    return None
  else:
    return ???
```