Graphes : parcours

Cours 12

Graphes - v1.0 Ⓟ

Lycée polyvalent Franklin Roosevelt, 10 Rue du Président Franklin Roosevelt, 51100 Reims

Table des matières

1 Parcours en largeur

1.1 Description et illustration

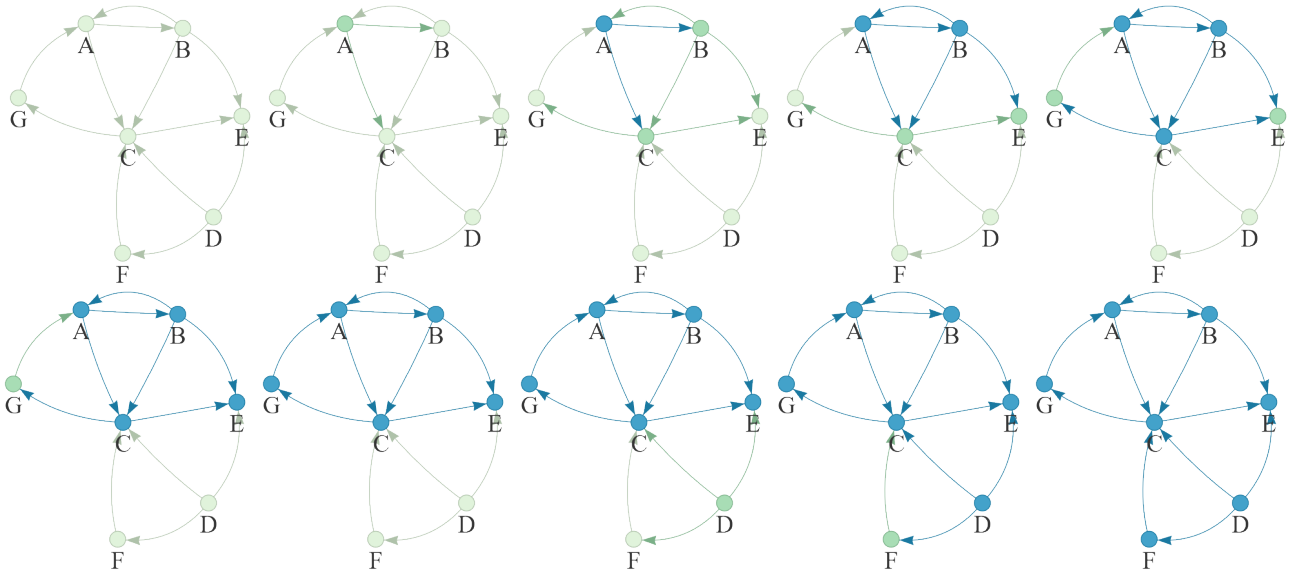

FIGURE 1 – Illustration du parcours en largeur

Le parcours en largeur (ou BFS, pour *Breadth First Search*) explore un graphe en considérant tous les successeurs d'un sommet exploré. On utilise une file pour stocker les différents sommets à explorer : quand on rencontre un nouveau sommet, on l'enfile, et on retire de la file pour savoir par où avancer. On part d'un sommet racine qui sera le premier à être mis en queue. On marque les sommets déjà explorés afin de ne pas effectuer un même parcours plusieurs fois. On a fini l'exploration à partir d'une racine donnée si la file est vide. Si, à partir d'une racine, tout le graphe n'a pas été parcouru, on recommence avec une nouvelle racine non-marquée.

1.2 État de la file

Les états successifs de la file sur les différentes étapes sont les suivantes dans l'illustration de la FIGURE 1.

Étape	1	2	3	4	5	6	7	8	9	10
File	[]	[A]	[B,C]	[C,E]	[E,G]	[G]	[]	[D]	[F]	[]

1.3 Pseudo-code

Le codage de cet algorithme peut se faire avec 2 sous-parties :

- un algorithme d'exploration en largeur à partir d'un sommet,
- le parcours en largeur qui appelle l'algorithme d'exploration pour chaque racine pas encore explorée.

Le pseudo-code correspondant à l'exploration en largeur est le suivant :

Entrées : Graphe G, Sommet r
f reçoit une file vide;
r enfilé dans f ;
r marqué comme parcouru ;
tant que *f est non vide* **faire**
    s retiré de la file;
    pour chaque *successeur ss de s dans G* **faire**
        si *ss marqué comme non-parcouru* **alors**
            ss enfilé dans f ;
            ss marqué comme parcouru ;

Algorithme 1 : Exploration en largeur à partir d'une racine r

Le pseudo-code correspondant au parcours complet est le suivant :

Entrées : Graphe G
pour chaque *sommet s dans les sommets de G* **faire**
    s marqué comme non-parcouru;
pour chaque *racine r dans les sommets de G* **faire**
    si *r marqué comme non-parcouru* **alors**
        explorationLargeur(G, r) ;

Algorithme 2 : Parcours en largeur complet

1.4 Implémentation en Python

On considère ici le graphe en entrée comme étant un dictionnaire d'adjacence. Pour consigner le marquage des sommets, on choisit d'utiliser un dictionnaire de booléens.

```
1  from collections import deque
2
3  def explorationLargeur(G, r, parcouru):
4      f = deque() # initialisation d'une deque vide
5      f.append(r)
6      parcouru[r] = True
7      while f:    # Vrai tant que f est non-vide
8          s = f.popleft()
9          for ss in G[s]: # G[s] liste des successeurs de s dans G
10             if not parcouru[ss]:
11                 f.append(ss)
12                 parcouru[ss] = True
13
```

```python
def parcoursLargeur(G):
    parcouru = {}   # dictionnaire de booléens pour le marquage
    for s in G.keys(): # Itération sur les clés de G (dictionnaire de listes)
        parcouru[s] = False
    for r in G.keys(): # Itération sur les clés de G (dictionnaire de listes)
        if not parcouru[r]:
            explorationLargeur(G, r)
```

2 Parcours en profondeur

2.1 Description et illustration

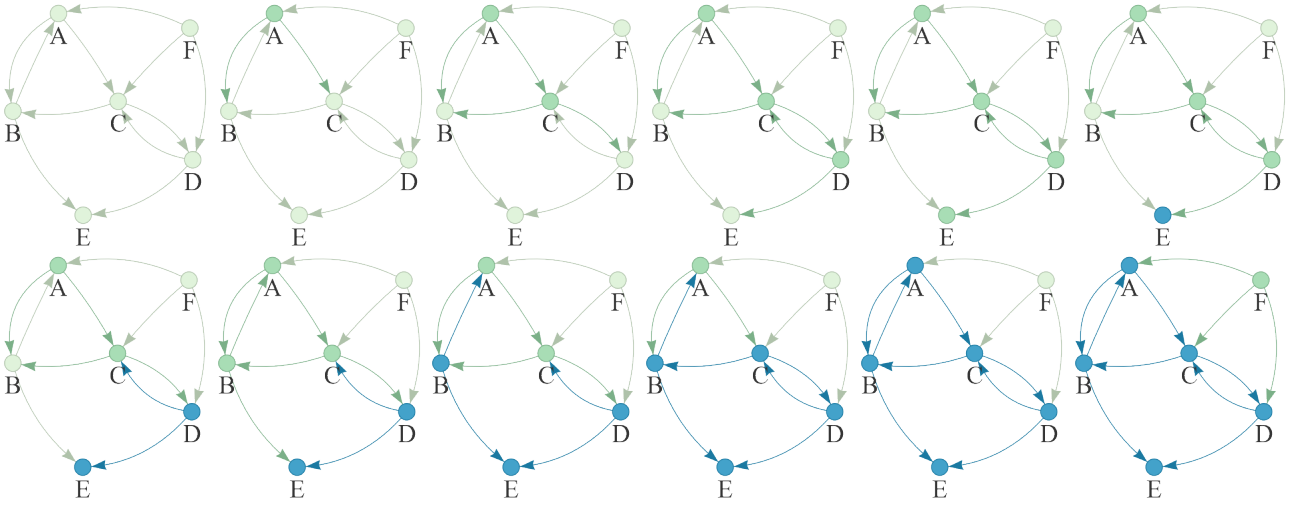

FIGURE 2 – Illustration du parcours en profondeur

Le parcours en profondeur (ou DFS, pour *Depth First Search*) explore un graphe en considérant un seul successeur d'un sommet exploré à la fois. On utilise une pile pour stocker les différents sommets à explorer : quand on rencontre un nouveau sommet, on l'empile. Quand un sommet n'a pas ou plus de voisin non-marqué, on peut le dépiler. On part d'un sommet racine qui sera le premier à être empiler. On marque les sommets déjà explorés afin de ne pas effectuer un même parcours plusieurs fois. On a fini l'exploration à partir d'une racine donnée si la pile est vide. Si, à partir d'une racine, tout le graphe n'a pas été parcouru, on recommence avec une nouvelle racine non-marquée.

2.2 État de la pile

Les états successifs de la pile sur les différentes étapes sont les suivantes dans l'illustration de la FIGURE 2.

Étape	1	2	3	4	5	6	7	8	9	10	11	12
Pile	[]	[A]	[A,C]	[A,C,D]	[A,C,D,E]	[A,C,D]	[A,C]	[A,C,B]	[A,C]	[A]	[]	[F]

2.3 Pseudo-code

Le codage de cet algorithme peut se faire avec 2 sous-parties :
- un algorithme d'exploration en profondeur à partir d'une pile de sommets, codé récursivement,
- le parcours en profondeur qui appelle l'algorithme d'exploration pour chaque racine pas encore explorée.

Le pseudo-code correspondant est donc le suivant :

Entrées : Graphe G, pile de sommets p
s valeur du haut de la pile p ;
s marqué comme parcouru ;
pour chaque *successeur ss de s dans G* **faire**
  si *ss n'est pas marqué* **alors**
    ss empilé dans p ;
    explorationProfondeur(G, p)
s enlevé de la pile p ;

Algorithme 3 : Exploration en profondeur avec pile

Entrées : Graphe G
pour chaque *sommet s dans les sommets de G* **faire**
  s marqué comme non-parcouru;
pour chaque *racine r dans les sommets de G* **faire**
  si *r marqué comme non-parcouru* **alors**
    p reçoit une pile vide ;
    r empilé dans p ;
    explorationPronfondeur(G, p) ;

Algorithme 4 : Parcours en profondeur complet

2.4 Implémentation en Python

On considère ici le graphe en entrée comme étant un dictionnaire d'adjacence. Pour consigner le marquage des sommets, on choisit d'utiliser un dictionnaire de booléens.

```
1  def explorationProfondeur(G, p, parcouru):
2      s = p[-1] # sommet provenant du haut de la pile
3      parcouru[s] = True
4      for ss in G[s]: # G[s] liste des successeurs de s dans G
5          if not parcouru[ss]:
6              p.append(ss) # ajout de ss dans la pile
7              explorationProfondeur(G, p, parcouru) # appel récursif
8      p.pop()   # haut de la pile retiré
9
10 def parcoursProfondeur(G):
11     parcouru = {}   # dictionnaire de booléens pour le marquage
12     for s in G.keys(): # Itération sur les clés de G (dictionnaire de listes)
13         parcouru[s] = False
14     for r in G.keys(): # Itération sur les clés de G (dictionnaire de listes)
15         if not parcouru[r]:
16             p = [r]    # pile contenant uniquement la racine
17             explorationProfondeur(G, p, parcouru)
```

La pile de sommets peut être interne et ne pas être explicité : en effet, cette pile de sommets correspond à la pile d'appels récursifs, et on peut donc ne pas expliciter cette pile.

```
def explorationProfondeur(G, s, parcouru):
    parcouru[s] = True
    for ss in G[s]: # G[s] liste des successeurs de s dans G
        if not parcouru[ss]:
            explorationProfondeur(G, ss, parcouru) # appel récursif

def parcoursProfondeur(G):
    parcouru = {}   # dictionnaire de booléens pour le marquage
    for s in G.keys(): # Itération sur les clés de G (dictionnaire de listes)
        parcouru[s] = False
    for r in G.keys(): # Itération sur les clés de G (dictionnaire de listes)
        if not parcouru[r]:
            explorationProfondeur(G, r, parcouru)
```

Attention

Les implémentations des algorithmes de parcours présentés ici n'ont rien d'officiel. De nombreuses variantes existent, et nous n'allons pas toutes les présenter.

À retenir

Ce qu'il est fondamental de retenir sur le parcours de graphe :

- il est primordial de laisser des marques de son passage par un graphe afin de ne pas être enfermé dans des boucles infinies causées par des circuits (dans un graphe orienté) ou des cycles (dans un graphe non-orienté) ;
- le parcours en largeur se base sur l'utilisation d'une file, forcément explicite ;
- le parcours en profondeur se base sur l'utilisation d'une pile, qui doit forcément être explicité dans un codage non-récursif, mais qui peut être implicite dans un codage récursif.

3 Utilisation des parcours

Les parcours sont au cœur de la résolution de nombreux problèmes dans les graphes :

- détermination des composantes connexes dans un graphe non orienté ;
- existence et obtention d'un chemin d'un sommet à un autre ;
- obtention du plus court chemin d'un sommet à un autre ;
- existence d'un cycle dans un graphe non orienté ou d'un circuit dans un graphe orienté ;
- etc.

Dans chaque cas, il faudra enrichir un des deux algorithmes de base pour obtenir les informations souhaitées. Si quelques cas vont être traité en cours, les Travaux Pratiques complèteront cet aperçu.

3.1 Détermination des composantes connexes dans un graphe non orienté

Dans un graphe non orienté, le parcours à partir d'une racine permet d'atteindre la totalités des sommets de la composante connexe qui contient cette racine : on peut donc, pour cet algorithme, enrichir soit le parcours en largeur, soit le parcours en profondeur.

On peut ainsi attribuer à chaque sommet un numéro caractérisant la composante connexe, numéro provenant d'un compteur. Ce compteur est incrémenté à chaque fois qu'une nouvelle composante connexe est visitée.

Proposer une implémentation à partir du parcours en largeur :

```
from collections import deque

def explorationLargeur(G, r, parcouru, composante, i):
    f = deque() # initialisation d'une deque vide
    f.append(r)
    parcouru[r] = True
    composante[r] = i
    while f:    # Vrai tant que f est non-vide
        s = f.popleft()
        for ss in G[s]: # G[s] liste des successeurs de s dans G
            if not parcouru[ss]:
                f.append(ss)
                parcouru[ss] = True
                composante[ss] = i

def parcoursLargeur(G):
    parcouru = {}   # dictionnaire de booléens pour le marquage
    composante = {} # dictionnaire d'entiers pour les composantes
    for s in G.keys(): # Itération sur les clés de G (dictionnaire de listes)
        parcouru[s] = False
    num_comp = 0 # compteur de composantes
    for r in G.keys(): # Itération sur les clés de G (dictionnaire de listes)
        if not parcouru[r]:
            explorationLargeur(G, r, parcouru, composante, num_comp)
            num_comp += 1 # incrémentation du compteur de composante
    return composante
```

3.2 Existence d'un circuit dans un graphe orienté

En modifiant légèrement un parcours en profondeur d'un graphe, on peut détecter la présence d'un circuit. En effet, si on se retrouve lors du parcours à rencontrer un sommet déjà dans la pile, alors le graphe comporte un circuit.

Proposer une implémentation :

```
def presenceCircuit(G):
    def explorationProfondeur(s):
        etat[s] = 'découvert'
        for ss in G[s]: # G[s] liste des successeurs de s dans G
            if etat[ss] == 'attente':
                return explorationProfondeur(ss) # appel récursif
            elif etat[ss] == 'découvert':
                return True
        etat[s] = 'exploré'
        return False
    etat = {}      # dictionnaire de chaine de caractères pour le marquage
    for s in G.keys(): # Itération sur les clés de G (dictionnaire de listes)
        etat[s] = 'attente'
    for r in G.keys(): # Itération sur les clés de G (dictionnaire de listes)
        if etat[r] == 'attente':
            if explorationProfondeur(r):
                return True
    return False
```