Chapter 1

Parcours de Graphe

Contenus	Capacités attendues	Commentaires
Arbres : structures hiérarchiques. Arbres binaires : nœuds, racines, feuilles, sous-arbres gauches, sous-arbres droits.	Identifier des situations nécessitant une structure de données arborescente. Évaluer quelques mesures des arbres binaires (taille, encadrement de la hauteur, etc.).	On fait le lien avec la rubrique « algorithmique ».
Algorithmes sur les graphes.	Parcourir un graphe en profondeur d'abord, en largeur d'abord. Repérer la présence d'un cycle dans un graphe. Chercher un chemin dans un graphe.	Le parcours d'un labyrinthe et le routage dans internet sont des exemples d'algorithme sur les graphes. L'exemple des graphes permet d'illustrer l'utilisation des classes en programmation.

Un parcours de graphe est un ordre d'énumération de ses sommets. Ce parcours est un algorithme consistant à explorer les sommets de proche en proche à partir d'un sommet initial. Le type de parcours étudié ici consiste donc à **explorer le graphe en passant par tous les sommets** : c'est un parcours dit *hamiltonien.*

On peut concevoir cet examen comme une promenade le long des arcs ou arêtes au cours de laquelle on visite les sommets. Le plus souvent, un parcours de graphe est un outil pour étudier une propriété globale du graphe, par exemple sa connexité.

1.1 Principe et algorithme général

Le parcours d'un graphe est une liste de tous ses sommets :

- Le premier élément est un sommet de départ choisi arbitrairement.
- Chaque sommet du graphe est visité une seule fois exactement.
- Chaque sommet, sauf le départ, est adjacent à au moins un sommet déjà visité.

Nous utiliserons le principe qui consiste à marquer les sommets au fur et à mesure de la visite du graphe. On commence par marquer le sommet initial s. Puis, tant qu'il existe un arc (x, y) non exploré avec x marqué et y non marqué, on marque y . A la fin de l'algorithme, les sommets marqués sont les descendants de s.

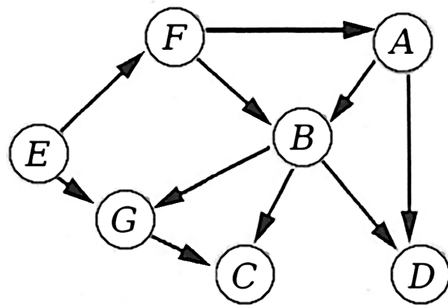
Il existe deux principales stratégies de parcours pour un graphe:

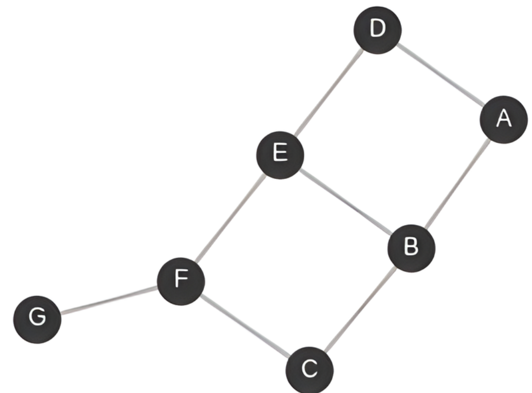
- **Parcours en largeur d'abord** Il consiste à explorer les sommets du graphe niveau par niveau, à partir d'un sommet donné. On explore donc tous les sommets successeurs d'un sommet avant de descendre plus loin dans le graphe.

- **Parcours en profondeur d'abord** Il ressemble au parcours d'un labyrinthe. Il consiste, à partir d'un sommet donné, à suivre un chemin le plus loin possible, puis à faire des retours en arrière pour reprendre tous les chemins ignorés précédemment. Une version itérative et récursive seront données.

Les arêtes/arcs sont choisies arbitrairement, il faut s'attendre à des visites dans des ordres différents !

Nous utiliserons le graphe 4 et le graphe 0 pour illustrer chaque parcours.





Graphe 4 orienté

Graphe 0 non orienté

1.1.1 Algorithme : première approche

Méthode Voici une écriture en pseudo-code qui permet de formaliser les parcours en largeur d'abord et en profondeur d'abord avec un seul algorithme :

Algorithme 1 : Fonction Parcours

```
   Données : Graphe G et sommet s0
   Résultat : None, on affiche les sommets
 1 ajouter s0 à a_traiter
 2 ajouter s0 à deja_vu
 3 tant que a_traiter ≠ ∅: faire
 4 |  sortir s de a_traiter
 5 |  pour chaque voisin t de s faire
 6 |  |  si t ∉ deja_vu alors
 7 |  |  |  ajouter t à a_traiter
 8 |  |  |  ajouter t à deja_vu
 9 |  |  fin
10 |  fin
11 fin
```

Mise en oeuvre : pile ou file ? La différence fondamentale entre le parcours en largeur et le parcours en profondeur provient de la façon de gérer cette liste d'attente :

- le parcours en largeur utilise une `file` d'attente,
- le parcours en profondeur utilise une `pile`.

Le type de la variable `a_traiter` va déterminer si ce parcours est en largeur (file) ou en profondeur (pile). De façon pratique, la structure de données `a_traiter` est une *liste "d'attente"* dans laquelle tous les sommets seront stockés.

Les classes Pile et File déjà écrites cette année pourront être réutilisées.

1.1.2 Parcours en profondeur : méthode récursive

Comme nous l'avons vu dans le chapitre *récursivité*, Python gère la pile des appels récursifs et on peut réécrire l'algorithme pour le parcours en profondeur. Cette approche n'est donc valable que pour le parcours en profondeur.

Algorithme 2 : Fonction parcours

```
1 ajouter s à deja_vu
2 traiter s
3 pour chaque voisin t de s faire
4 |  si t ∉ deja_vu alors
5 |  |  parcours(s)
6 |  fin
7 fin
```

1.2 Implémentation : parcours en largeur d'abord

Méthode: Le parcours en largeur d'un graphe consiste à explorer les sommets du graphe niveau par niveau, à partir d'un sommet donné. Pour implémenter l'algorithme, une file est nécessaire pour y stocker les sommets successifs du sommet en cours de traitement (prévisite).

Entrée: On dispose d'un graphe G = (S, A) et un sommet s de G.

Sortie: On se contente ici d'un affichage mais la plupart du temps un tableau des pères de chaque sommet est indispensable. On peut également obtenir les distances entre s et chaque sommet de G en ajoutant une variable. On obtient un arbre couvrant si G est connexe.

Exercice : Implémentation possible Cette implémentation présente plusieurs erreurs : corrigez-les.

```python
def bfs(G:dict, s) :
    a_traiter = []
    deja_vu = []
    a_traiter.enfiler(s0)
    deja_vu.append(s0)
    while a_traiter :
        s = a_traiter.defiler()
        for v in G[s] :
            if t not in deja_vu :
                a_traiter.enfiler[t]
                deja_vu.append(t)
```

Exemple 1 Pour les *graphes* 4 et 0, à partir du sommet **A**, préciser un parcours en largeur.

1.3 Implémentation : parcours en profondeur d'abord

Méthode:Le parcours en profondeur consiste, à partir d'un sommet donné, à aller le plus loin possible. Quand ce n'est pas possible, on revient en arrière et on essaie de parcourir en profondeur en prenant un sommet qui n'a pas encore été visité.

1.3.1 Méthode itérative

Pour implémenter l'algorithme, une pile est nécessaire pour y stocker les sommets successifs du sommet en cours de traitement (prévisite).

Entrée: On dispose d'un graphe G = (S, A) et un sommet s de G.

Sortie: Tableau des pères.

Sur le graphe précédent, un parcours en profondeur possible est 1-2-4-5-7-8-6-3

```
def dfs(G:dict, s0) :
    a_traiter = Pile()
    deja_vu = []
    a_traiter.enpiler(s0)
    deja_vu.append(s0)
    while not a_traiter_est_vide() :
        s = a_traiter.depiler()
        for v in G[s] :
            if t not in deja_vu :
                a_traiter.enfiler(t)
                deja_vu.append(t)
```

Exemple Pour les graphes 4 et , à partir du sommet **A**, préciser un parcours en largeur. (Voir 1.5)

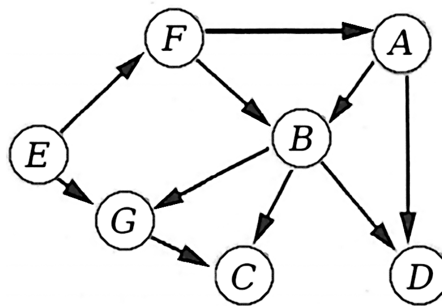


Figure 1.1: Graphe 4

1.3.2 Méthode récursive

Le parcours en profondeur s'implémente naturellement de façon récursive.

Implémentation possible

```
deja_vu = []
def dfs_rec(s, G):
    deja_vu.append(s)
    for t in G[s]:
        if t not in deja_vu:
            #couleur[v] = 'gris'
            dfs_rec(t, G)
```

1.3.3 Complément

Comparaison des deux algorithmes Les deux algorithmes itératifs côte à côte pour les comparer :

```
def dfs(G:dict, s0) :
    a_traiter = File()
    deja_vu = []
    a_traiter.enpiler(s0)
    deja_vu.append(s0)
    while not a_traiter_est_vide() :
        s = a_traiter.depiler()
        for v in G[s] :
            if t not in deja_vu :
                a_traiter.enfiler(t)
                deja_vu.append(t)
```

```
def dfs(G:dict, s0) :
    a_traiter = Pile()
    deja_vu = []
    a_traiter.empiler(s0)
    deja_vu.append(s0)
    while not a_traiter_est_vide :
        s = a_traiter.depiler()
        for v in G[s] :
            if t not in deja_vu :
                a_traiter.empiler(t)
                deja_vu.append(t)
```

Connexité Tous les sommets ne sont pas forcément atteignables à partir du sommet origine, on dit que le graphe n'est pas connexe. Il faudrait donc appeler notre procédure tant que tous les sommets ne sont pas visités.

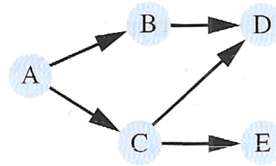
Coloriage On peut utiliser des couleurs pour visualiser le parcours :

- La liste d'attente `L` stocke les sommets en cours d'analyse (on visite leurs descendants).
- Blanc à l'initialisation, les sommets sont colorés en gris à leur entrée et en noir dès leur sortie de cette liste.
- Les sorties et entrées de cette liste dépendent donc du type de parcours, observer en particulier les lignes 4, 7 et 10. La liste se comporte soit comme une **file**, soit comme une **pile**.
- Deux couleurs suffisent pour effectuer le parcours, **voire même un seul ensemble où sont mémorisés les sommets visités**, on pourra dans ce cas utiliser un type `set`. Cependant l'utilisation de trois couleurs permet de bien comprendre le parcours et fournit plus d'informations qui seront utiles ensuite, par exemple pour la détection d'un cycle.

Figure 1.2: Illustration David Revoy⋆Licence CC-BY 4.0

1.4 Exercices/TP

Exercice 1



1. Donnez la matrice d'adjacence et le liste de successeurs pour ce graphe.
2. A chaque étape, recopier le graphe, colorier les sommets et indiquer:
   - L'état de la pile pour un parcours en profondeur. Donner ce parcours.
   - L'état de la file pour un parcours en largeur. Donner ce parcours.

Exercice 2

1. Implementer les trois algorithmes du cours.
2. En utilisant la classe `Pile_File` programmée au chapitre type abstrait de donnée, réécrire les algorithmes itératifs.
3. Vérifier les parcours obtenus.

Exercice 3 On donne une classe `Graphe`.

1. Implémentez le graphe suivant à l'aide de cette classe.
2. Écrivez une méthode `creer_arete`.

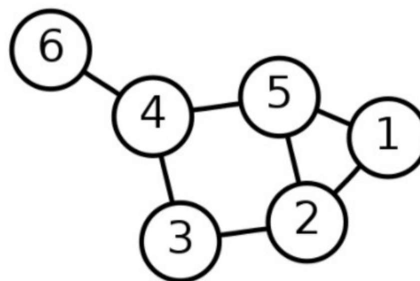


Figure 1.3: Graphe 0

3. Donnez les parcours en largeur et en profondeur pour ce graphe et vérifiez votre proposition à l'aide de votre code, adapté pour cette classe `Graphe`.

Note sur le type *python* `set` : Dans cette classe, les sommets adjacents (sortants) sont-stockés dans un *set*. On crée un `set` vide avec l'instruction `set()`.

On ne peut pas accéder à un élément avec `[]` mais il peut-être parcouru : c'est un *itérable*. Son intérêt est ici qu'il ignore les doublons. Les émthode `add` et `pop` permettent respectiveent d'ajouter ou supprimer un élément.

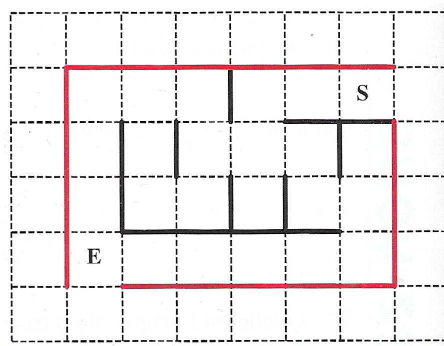
```
>>> v = set()
>>> v = {1, 3, 2, 5, 3}
{1, 3, 2, 5}
>>> v.add(8)
{1, 3, 2, 5, 8}
```

```python
class Graphe():
    """
    Le graphe est représneté par un dicitonanire d'adjacence
    """

    def __init__(self):
        self.adj = {}

    def ajouter_sommet(self, s):
        """
        L'ensemble des voisins du sommet s sont dans un 'set'
        """
        if s not in self.adj:
            self.adj[s] = set()

    def ajouter_arc(self, s1, s2):
        self.ajouter_sommet(s1)
        self.ajouter_sommet(s2)
        self.adj[s1].add(s2)

    def arc(self, s1, s2):
        """
        renvoie True si un arc existe entre s1 et s2
        """
        return s2 in self.adj[s1]

    def sommets(self):
        return list(self.adj)

    def voisins(self, s):
        """
        renvoie la liste des voisins de s
        """
        return self.adj[s]
```

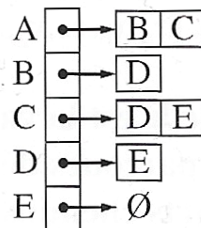
Exercice 4 Nous avons vu que tous les sommets ne sont pas nécessairement atteignables à partir du sommet choisi. Programmer une fonction `parcours_complet(G)` permettant de parcourir tout le graphe `G` dans ce cas.

Exercice 5 On représente un labyrinthe sous la forme d'une grille: Le déplacement est autorisé de case en case aux 4 points cardinaux. Les frontières du labyrinthe et les obstacles en trait plein noir. L'objectif est d'aller de l'entrée E à la sortie S en empruntant le moins de case possible.

1. Modélisez ce labyrinthe par un graphe.
2. Quel algorithme peut trouver une chaîne permettant d'atteindre la sortie le plus vite ?
3. Appliquez l'algorithme et donnez la chaîne obtenue.



Exercice 6 On représente un graphe G par une liste de successeur. Donnez un parcours en profondeur possible en précisant les différents états de la pile et l'arborescence obtenue.

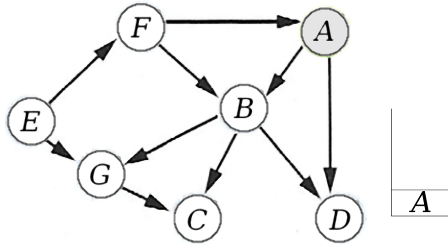


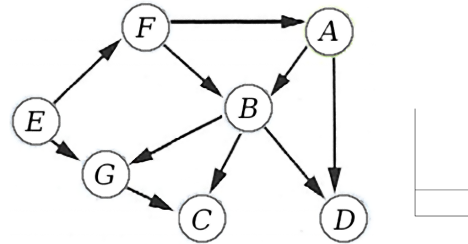
Exercice Même exercice à partir de la matrice d'adjacence et du point A pour déterminer s'il est connexe.

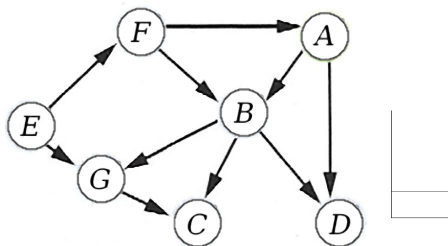
$$M' = \begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array} \begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

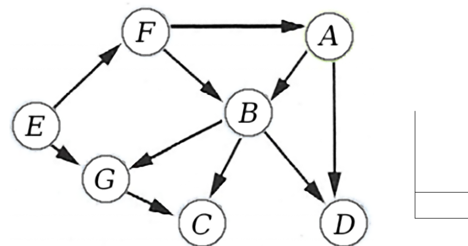
1.5 Annexes

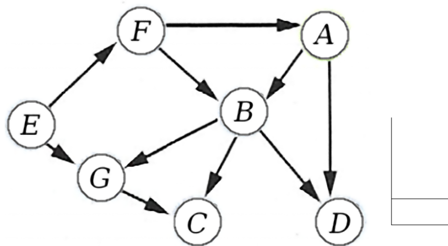
Parcours en profondeur

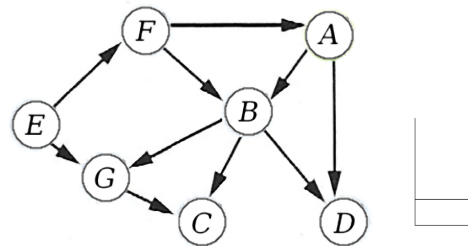


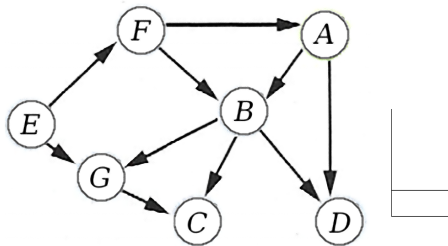


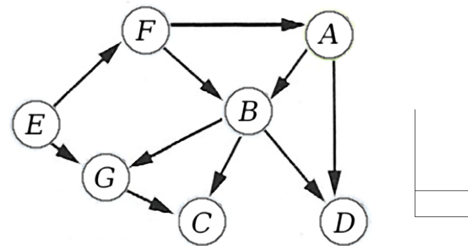


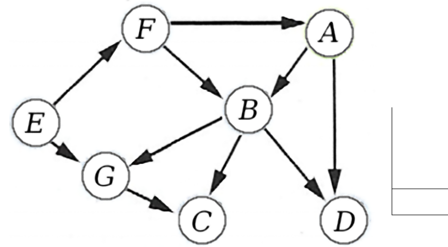


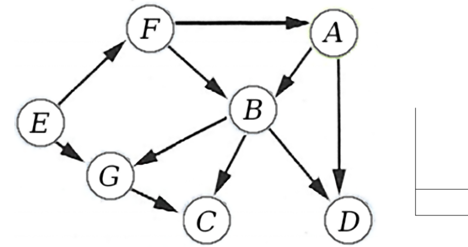




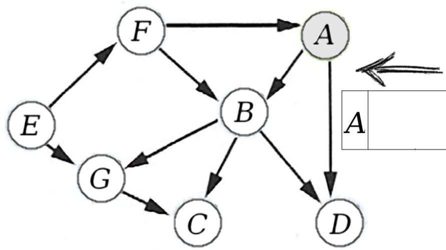


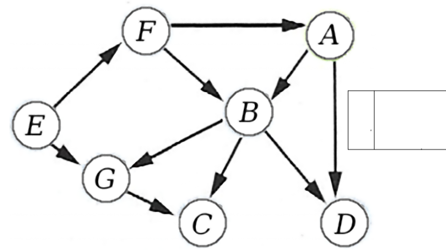


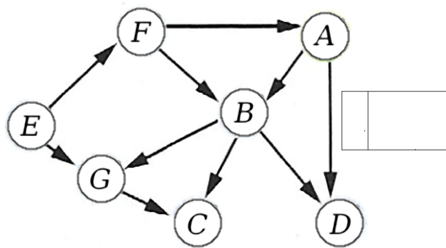


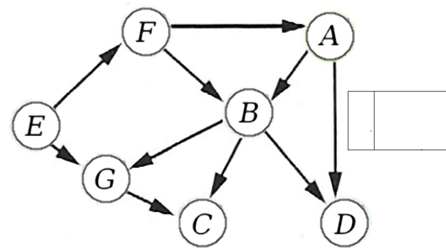


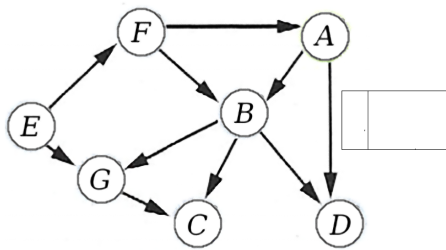
Parcours en largeur

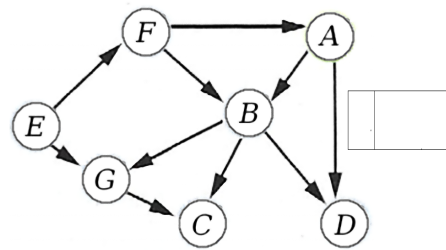


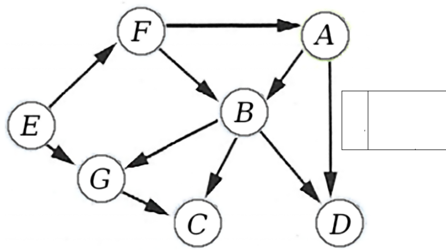


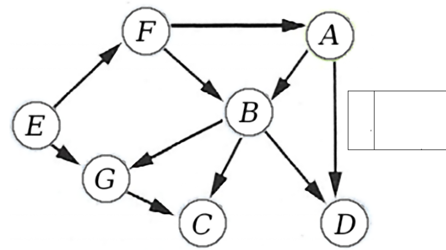


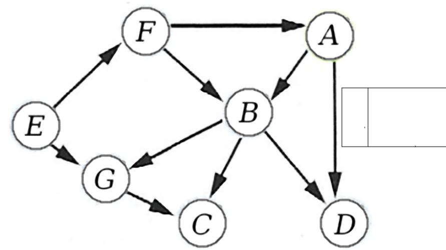


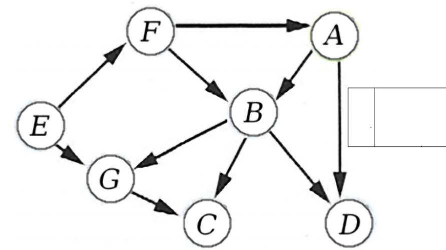












1.6 Références

Site du lycée Émile Duclaux d'Aurillac.