ITC – cours n°14

Parcours de graphes

I. Position du problème

Un parcours de graphe est un algorithme qui permet d'explorer un graphe en suivant les sommets de proche en proche à travers les arêtes, à partir d'un sommet initial. Un parcours peut servir

- à identifier les composantes connexes/fortement connexes ;
- à repérer les cycles ;
- à construire un arbre couvrant l'ensemble des sommets connexes du graphe ;
- trouver un plus court chemin d'un sommet à un autre ;
- ...et encore d'autres applications.

L'idée générale est de visiter les sommets adjacents à ceux déjà visités : la question est comment choisir, à partir des sommets visités, le sommet suivant à visiter. Les arcs ou arêtes effectivement empruntés par le parcours forment alors un arbre, ou une forêt dans le cas d'un graphe à plusieurs composantes connexes.

Dans tout le chapitre, on note u ou v des sommets, (u, v) un arc, $\{u, v\}$ une arête, et on suppose que le graphe G est décrit sous forme de listes d'adjacence $G[u]$.

Dans nos codes en Python, on gardera trace de la date/distance de découverte, du sommet précédent et de l'état d'un sommet (représenté par une couleur verte, bleue ou rouge) à l'aide de dictionnaires `distance`, `précédent` et `état` ayant pour clés les noms des sommets. On se dote d'une fonction d'initialisation de ces dictionnaires :

```
def init_parcours(g):
    précédent = {u: '' for u in g.keys()}
    état = {u: 'V' for u in g.keys()}
    distance = {u: float('inf') for u in g.keys()}
    return précédent, état, distance
```

II. Parcours en largeur

2.1. Principe

Le parcours en largeur consiste à explorer d'abord tous les sommets adjacents au sommet de départ, puis les sommets adjacents de ces sommets adjacents, etc.... On fait donc avancer un « front de parcours » de façon régulière. On peut voir un exemple d'un tel parcours sur la figure 1. En conséquence, l'étape du parcours à laquelle un sommet est découvert est la distance entre ce sommet et celui de départ.

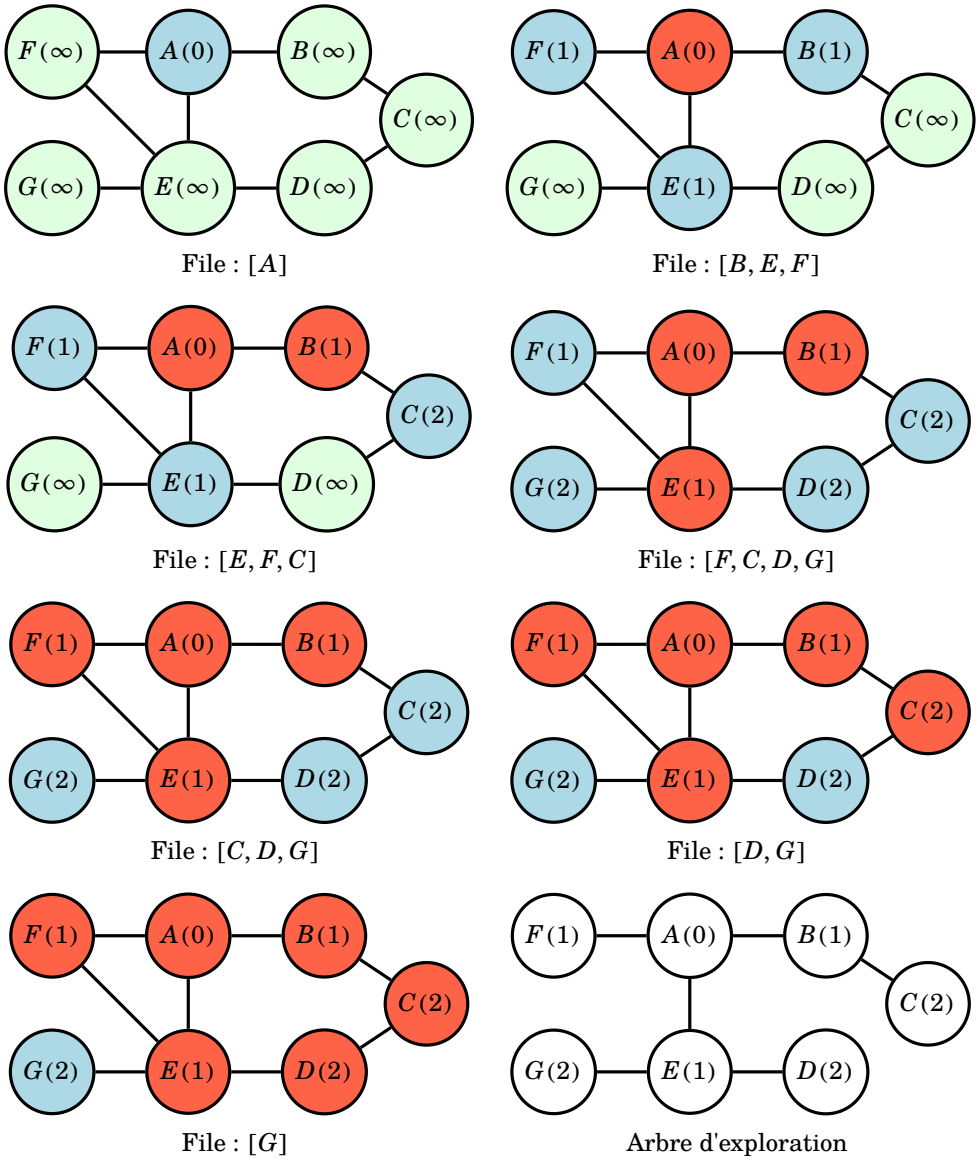

Figure 1 : Exemple de parcours en largeur

Afin de traiter les sommets dans le bon ordre, on utilise une file : lorsqu'on découvre un sommet adjacent, on le place dans la file, ce qui garantit qu'il sera exploré avant les sommets découverts plus loin que lui. Lorsqu'on visite un sommet u, on va donc parcourir sa liste d'ajacence et pour chaque sommet $v \in G[u]$ ainsi découvert :

- on regarde si v n'a pas déjà été visité (couleur rouge) ; si non, on effectue les étapes

suivantes ;
- on note que v a été atteint depuis u : `précédent[v] = u` ;
- on note à quelle distance v a été atteint ;
- on marque v comme découvert : couleur bleue ;
- on insère v dans la file pour visiter ses sommets adjacents.

Une fois la liste d'adjacence de u ainsi traitée, on colorie u en rouge pour noter qu'il a été exploré.

2.2. Proposition d'implémentation

La fonction de visite d'un sommet se présente donc ainsi :

```
def visiter_larg(g, file, préc, état, dist):
    u = file.get()  # récupération du sommet à visiter, bleu
    for v in g[u]:
        if état[v] == 'V':  # sommet v vert : non découvert
            préc[v] = u
            dist[v] = dist[u] + 1
            état[v] = 'B'  # on colorie v en bleu
            file.put(v)
    état[u] = 'R'  # sommet u traité
```

Le code du parcours depuis un sommet de départ u_0 s'écrit alors :

```
def parcours_largeur(g, u0):
    # initialisation
    file = queue.Queue()
    précédent, état, distances = init_parcours(g)
    état[u0], distances[u0] = 'B', 0
    file.put(u0)
    # début du parcours
    while not file.empty():
        visiter_larg(g, file, précédent, état, distances)
    # file vide : parcours fini
    return # à adapter selon ce que l'on souhaite
```

2.3. Propriétés

a) *Complexité*

Le système de marquage des sommets (visité ou non) permet de s'assurer que chaque sommet est mis dans la file (et donc enlevé) au plus une fois ; et pour chacun des sommets, on parcourt au plus une fois l'arc/arête le reliant à un autre sommet. On en déduit que la complexité du parcours est $\mathcal{O}(n_S + n_A)$.

b) *Plus court chemin*

On définit la **distance de plus court chemin** $\delta(u, v)$ comme le nombre minimum d'arcs/arêtes reliant un sommet u à un sommet v ; on nomme **plus court chemin** un chemin de u à v de longueur $\delta(u, v)$. L'algorithme de parcours en largeur va naturellement déterminer les distances de plus court chemin entre le sommet de départ u_0 et chaque autre sommet accessible depuis u_0 (appartenant à la même composante connexe).

Il peut y avoir plusieurs plus courts chemins pour atteindre un sommet donné ; le parcours en largeur en exhibe un en particulier, qui dépend de l'ordre dans lequel est rangée la liste d'adjacence des différents sommets parcours.

c) *Arbre de parcours en largeur*

On peut définir un arbre de parcours en largeur comme l'arbre composé des arcs visités uniquement ; il s'agit donc d'un sous-ensemble du graphe d'origine, qui conserve la propriété de plus court chemin entre le sommet de départ u_0 et tous les autres sommets accessibles.

III. Parcours en profondeur

3.1. Principe

Le parcours en profondeur consiste à explorer, pour chaque sommet u, un sommet adjacent v_0, puis un sommet adjacent w_0 de ce sommet v_0...et de ne revenir aux autres sommets v_1, v_2...adjacents à u qu'une fois qu'il n'y a plus de sommets à explorer accesibles depuis v_0. On va ainsi « creuser » dans une direction possible du graphe au maximum, et revenir en arrière quand on atteint un « cul-de-sac ». Il s'agit donc d'un algorithme de backtracking.

3.2. Proposition d'implémentation

Une façon de facilement arriver à ce résultat est d'utiliser une approche récursive : pour tout sommet, on effectue un parcours en profondeur sur chacun de ses voisins.

```
def visiter_prof(g, u, prec, état):
    état[u] = 'B'  # on commence à visiter u
    for v in g[u]:
        if état[v] == 'V':  # si v non découvert
            prec[v] = u
            visiter_prof(g, v, prec, état)
    état[u] = 'R'  # u visité


def parcours_profondeur(g, u0):
    prec, état, _ = init_parcours(g)
    visiter_prof(g, u0, prec, état)
    return # à adapter selon ce que l'on veut
```

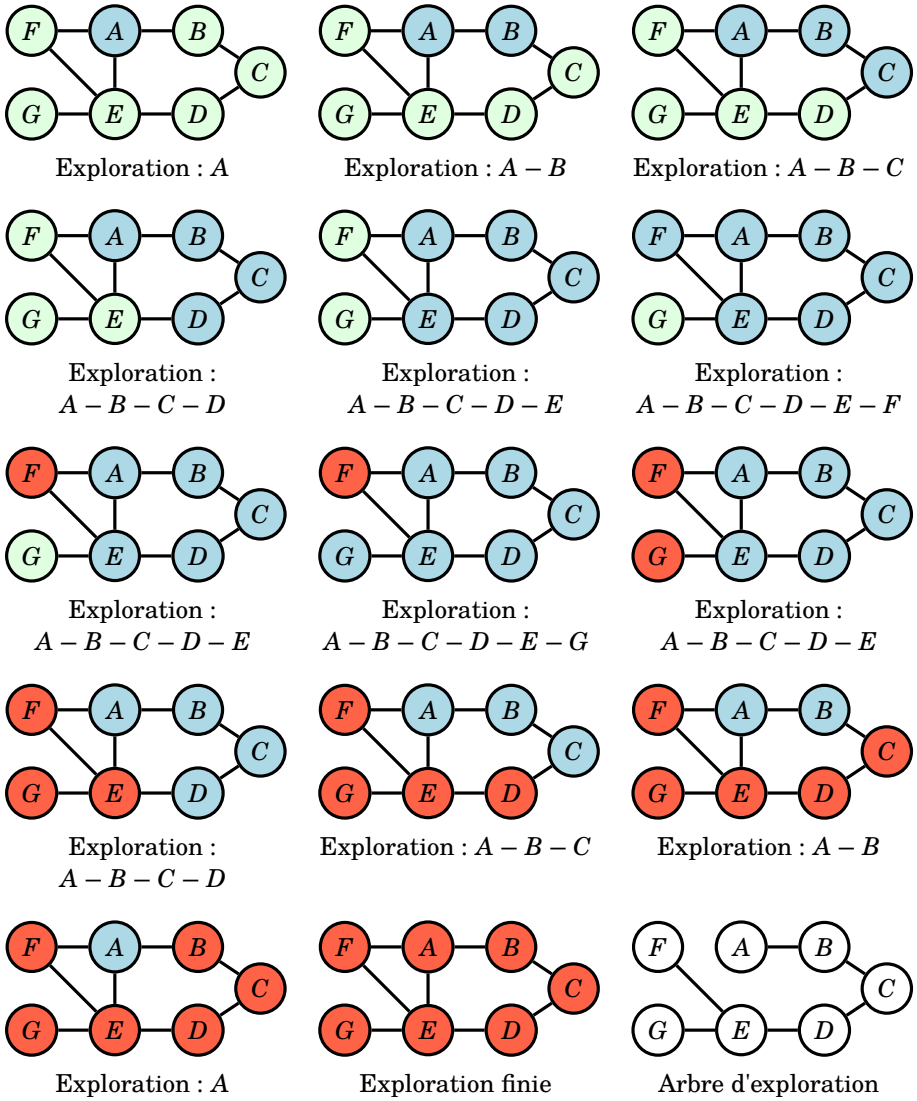

Figure 2 : Exemple de parcours en profondeur

Une implémentation sans récursivité peut être proposée en s'inspirant du parcours en largeur mais en remplaçant la file par une pile : ainsi, on s'assure qu'on explorera en dernier les sommets adjacents mis en attente, puisqu'ils sont au fond de la pile. L'inconvénient est que dans ce cas, un sommet peut être au fond de la pile, et découvert par un autre chemin avant de le dépiler ; il faut donc considérer que le sommet n'est pas « vraiment » visité quand on le met au fond de la pile (colorié en bleu), mais uniquement quand on a exploré ses voisins (auquel cas on le colorie en rouge). Il ne faut donc noter comme visité un sommet que lorsqu'on l'a effectivement dépilé pour la première fois, et ne pas le re-visiter si on le dépile à nouveau et qu'il est colorié en rouge.

```
def visiter_prof_pile(g, pile, préc, état):
    u = pile.get()  # récupération du sommet à visiter
    if état[u] == 'V':  # non visité
        état[u] = 'B'   # on commence à creuser
        pile.put(u)
        adj_u = [v for v in g[u]]  # (1) liste d'adjacence de u
        for v in adj_u[::-1]:      # (1) on empile dans l'ordre inverse
            if état[v] == 'V':     # sommet v non visité précédemment
                préc[v] = u
                pile.put(v)        # on le coloriera en le dépilant
    elif état[u] == 'B':  # on a déjà creusé à partir de u
        état[u] = 'R'     # fin


def parcours_profondeur_pile(g, u0):
    prec, état, _ = init_parcours(g)
    pile = queue.LifoQueue()
    pile.put(u0)
    while not pile.empty():
        visiter_prof_pile(g, pile, prec, état)
    return # à adapter selon ce que l'on veut
```

Remarque (1) dans le code : on choisit ici d'explorer la liste d'adjacence dans l'ordre inverse, afin de la dépiler dans l'ordre et de retrouver le même arbre de parcours que pour la version récursive ; mais si on choisit d'empiler dans l'ordre, ça ne change pas la nature de l'algorithme.

3.3. Complexité

Dans sa version récursive, on voit que la procédure `visiter_sommet` sera appelée exactement une fois par sommet de la composante connexe ; et un appel pour le sommet u parcours la liste d'adjacence de u, ce qui conduit à une complexité totale $\mathcal{O}(n_S + n_A)$.

> La version itérative proposée ici est un peu moins efficace, puisque les sommets peuvent être empilés plusieurs fois (ils ne restent visités qu'une fois).

IV. Exemples d'application (non exhaustif)

4.1. Détection des composantes connexes

Chaque algorithme de parcours sur un graphe non orienté ne peut qu'atteindre les sommets appartenant à une même composante connexe : on peut utiliser l'un ou l'autre des algorithmes de parcours pour extraire les composantes connexes en re-démarrant un parcours depuis chaque sommet encore non visité.

Une implémentation possible, en utilisant un parcours qui renvoie une liste des sommets visités :

```
def composantes_connexes(g):
    composantes = []
    visité = {u: False for u in g.keys()}
    for u in g.keys():
        if not visité[u]:
            nvelle_comp = parcours_largeur_liste_sommets(g, u)
            composantes.append(nvelle_comp)
            for v in nvelle_comp:
                visité[v] = True
    return composantes
```

4.2. Détection de cycles

On peut utiliser le parcours en profondeur pour détecter les cycles dans un graphe : en effet, si un sommet bleu v est découvert depuis u, c'est qu'il existe un chemin composé uniquement de sommets bleus allant de v à u ; l'arête (u, v) qui vient d'être découverte ferme donc le cycle.

On obtient le cycle en remontant les sommets précédents depuis u jusqu'à v (éventuellement en inversant pour retrouver le vrai sens du cycle si le graphe est orienté). On propose ci-après une implémentation simple dans laquelle on se contente de renvoyer `True` si un cycle est détecté et `False` sinon.

```
def visiter_prof_cycles(g, u, prec, état):
    état[u] = 'B'  # on commence à visiter u
    for v in g[u]:
        if état[v] == 'V':  # si v non découvert
            prec[v] = u
            if visiter_prof_cycles(g, v, prec, état):
                return True
        elif état[v] == 'B' and prec[u] != v:  # un cycle ici
            return True
        # si pas de cycle trouvé, on visite le sommet v suivant
    état[u] = 'R'  # u visité
    return False

def présence_cycles(g):
    """
    Détection de cycles en adaptant le parcours en profondeur.
    Args:
        g (dict[dict]): graphe d'entrée
    Returns:
        présence (bool): True si un cycle a été détecté, False sinon
    """
    préc, état, dist = init_parcours(g)
    for u in g.keys():
        if état[u] == 'V':  # ne pas ré-explorer
            if visiter_prof_cycles(g, u, préc, état):
                return True
    return False
```