11 Parcours de graphes

- Parcours d'un graphe

11.1 Introduction

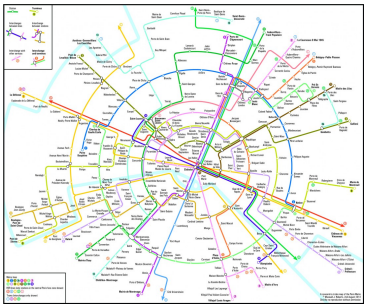

FIGURE 11.1 – Représentation ciculaire du métro parisien

Une fois que nous sommes en présence d'un graphe, il va falloir le parcourir pour répondre à différentes questions :

- est-il possible de joindre un sommet A et un sommet B ?
- est-il possible, depuis un sommet, de rejoindre tous les autres sommets du graphe ?
- peut-on détecter la présence de cycle ou de circuit dans un graphe ?
- quel est le plus court chemin pour joindre deux sommets ?
- *etc.*

Les deux algorithmes principaux sont les suivants :

- le parcours en largeur – *Breadth-First Search* (BFS) – pour lequel on va commencer par visiter les sommets les plus proches du sommet initial (sommets de niveau 1), puis les plus proches des sommets de niveau 1 *etc.* ;
- le parcours en profondeur – *Depth-First Search* (DFS) – pour lequel on part d'un sommet initial jusqu'au sommet le plus loin. On remonte alors la pile pour explorer les ramifications.

Une des difficultés du parcours de graphe est d'éviter de tourner en rond. C'est pour cela qu'on mémorisera l'information d'avoir visité ou non un sommet. On parle aussi de marquage.

11.2 Parcours en largeur

11.2.1 Un premier algorithme

On propose ci-dessous un algorithme de parcours en largeur en utilisant un graphe implémenté sous forme de liste d'adjacence ainsi qu'un sommet `s` de départ.

```
1 def bfs(G:dict, s:str) -> None:
2     """
3     G : graphe sous forme de dictionnaire d'adjacence
4     s : sommet du graphe (Chaine de caractere du type "S1").
5     """
6     visited = {}
7     for sommet,voisins in G.items():
8         visited[sommet] = False
```

```
    # Le premier sommet à visiter entre dans la file
    file = deque([s])
    while len(file) > 0:
        # On visite la tête de file
        tete = file.pop()
        # On vérifier qu'elle n'a pas été visitée
        if not visited[tete]:
            # Si on l'avait pas visité, maintenant c'est le cas :)
            visited[tete] = True
            # On met les voisins de tete dans la file
            for v in G[tete]:
                file.appendleft(v)
```

Dans cet algorithme :

- on commence par créer une liste ayant pour taille le nombre de sommets. Cette liste va permettre de savoir si un sommet a été visité ou non ;
- dans la file, on va commencer par ajouter le sommet initial ;
- on commence alors à traiter la file en extrayant l'indice du sommet initial ;
- si ce sommet n'a pas été visité, il devient visité ;
- on ajoute alors dans la file l'ensemble des voisins du sommet initial ;
- on continue alors de traiter la file.

Remarque

En l'état, à quoi sert cet algorithme ?

11.2.2 Applications

Exemple –

Comment connaître la distance d'un sommet `s` aux autres ?

```
def distances(G, s):
    dist = [-1]*len(G)
    q = deque([(s, 0)])
    while len(q) > 0:
        u, d = q.pop()
        if dist[u] == -1:
            dist[u] = d
            for v in G[u]:
                q.appendleft((v, d + 1))
    return dist
```

Exemple –

Comment connaître un plus court chemin d'un sommet s à un autre ?

```
def bfs(G, s):
    pred = [-1]*len(G)
    q = deque([(s, s)])
    while len(q) > 0:
        u, p = q.pop()
        if pred[u] == -1:
            pred[u] = p
            for v in G[u]:
                q.appendleft((v, u))
```

```
    return pred

def path(pred, s, v):
    L = []
    while v != s:
        L.append(v)
        v = pred[v]
    L.append(s)
    return L[::-1] # inverse le chemin
```

11.2.3 D'autres formulations

```
def parcoursLargeur(G):
    parcouru = {}   # dictionnaire de booléens pour le marquage
    for s in G:     # Itération sur les clés de G (dictionnaire de listes)
        parcouru[s] = False
    f = deque()     # initialisation d'une deque vide
    for r in G:     # Itération sur les clés de G (dictionnaire de listes)
        if not parcouru[r]:
            parcouru[r] = True
            f.append(r)
            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]:
                        parcouru[ss] = True
                        f.append(ss)
```

```
def parcoursLargeur(G):
    def explorationLargeur(r):
        f = deque() # initialisation d'une deque vide
        f.append(r)
        parcouru[r] = True
        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
    parcouru = {}   # dictionnaire de booléens pour le marquage
    for s in G.keys(): # Itération sur les clés de G (dictionnaire de listes)
        parcouru[s] = False # initialisation du marquage
    for r in G.keys(): # Itération sur les clés de G (dictionnaire de listes)
        if not parcouru[r]:
            explorationLargeur(r)
```

11.3 Parcours en profondeur

11.3.1 Un premier algorithme

On propose ci-dessous un algorithme de parcours en profondeur en utilisant un graphe implémenté sous forme de liste d'adjacence ainsi qu'un sommet s de départ.

```
def dfs(G, s): #
    visited = [False]*len(G)
```

```
    pile = [s]
    while len(pile) > 0:
        u = pile.pop()
        if not visited[u]:
            visited[u] = True
            for v in G[u]:
                pile.append(v)
```

Dans cet algorithme :

- on commence par créer une liste ayant pour taile le nombre de sommets. Cette liste va permettre de savoir si un sommet a été visité ou non ;
- dans la pile, on va commencer par ajouter le sommet initial ;
- on commence alors à traiter le sommet initial après l'avoir extrait de la pile ;
- si ce sommet n'a pas été visité, il devient visité ;
- on ajoute alors dans la pile l'ensemble des voisins du sommet initial ;
- on continue alors de traiter la pile.

À la différence du parcours en largeur, lorsqu'on va traiter la pile, on va s'éloigner du sommet initial... avant d'y revenir quand toutes les voies auront été explorées.

11.3.2 Une autre formulation

La formulation précédente du parcours en profondeur a l'avantage d'être très proche de celle du parcours en largeur. Cependant, si on traçait l'arbre permettant de visualiser les sommets visités, on constate que l'algorithme crée des ramifications qui ne correspondent pas vraiment à un parcours en profondeur [1]. Il s'agit alors de

1: https://11011110.github.io/blog/2013/12/17/stack-based-graph-traversal.html

```
def dfs(G, s): #
    visited = [False]*len(G)
    pile = [s]
    while len(pile) > 0:
        u = pile.pop()
        if not visited[u]:
            visited[u] = True
            for v in G[u]:
                pile.append(v)
```

```
def dfs_2(G, s): # A VERIFIER :)
    visited = [False]*len(G)
    pile = [s]
    while len(pile) > 0 :
        u = pile.pop()
        if not visited[u]:
            visited[u] = True
            pile.append(u)
            for v in G[v] :
                pile.append(v)
```

11.3.3 Une autre formulation (récursive)

```
def dfs(G, s):
    visited = [False]*len(G)
    def aux(u):
        if not visited[u]:
            visited[u] = True
            for v in G[u]:
                aux(v)
    aux(s)
```

11.3.4 Applications

Exemple –

Lister les sommets dans l'ordre de leur visite.

Exemple –

Comment déterminer si un graphe non orienté est connexe ?

Exemple –

Comment déterminer si un graphe non orienté contient un cycle ?

Références

- Cours de Quentin Fortier https://fortierq.github.io/itc1/.
- Cours de JB Bianquis. Chapitre 5 : Parcours de graphes. Lycée du Parc. Lyon.
- Cours de T. Kovaltchouk. Graphes : parcours. Lycée polyvalent Franklin Roosevelt, Reims.
- https://perso.liris.cnrs.fr/vincent.nivoliers/lifap6/Supports/Cours/graph_traversal.html
- http://mpechaud.fr/scripts/parcours/index.html