Parcours de graphes

- Parcours d'un graphe

10.1 Parcours de graphe

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.*

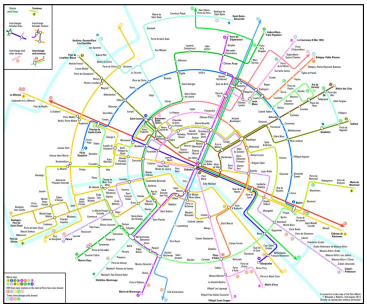


FIGURE 10.1 – Représentation ciculaire du métro parisien

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.

10.1.1 Parcours en largeur

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
9      # Le premier sommet à visiter entre dans la file
10     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 ?

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

```

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

10.1.2 Parcours en profondeur

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.

```python
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.

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

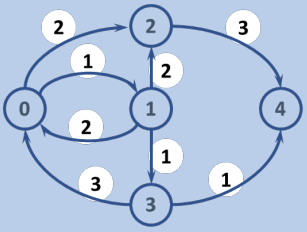
- Cours de Quentin Fortier https://fortierq.github.io/itc1/.
- Cours de JB Bianquis. Chapitre 5 : Parcours de graphes. Lycée du Parc. Lyon.
- Cours de T. Kovaltchouk. Graphes : parcours. Lycée polyvalent Franklin Roosevelt, Reims.
- https://perso.liris.cnrs.fr/vincent.nivoliers/lifap6/Supports/Cours/graph_traversal.html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

10.2 Algorithme de Dijkstra

L'algorithme de Dijkstra est un parcours en largeur d'un graphe **pondéré (poids positifs)** et orienté. Il permet de calculer l'ensemble des plus courts chemins entre un sommet vers tous les autres sommets du graphe.

Pour modéliser le graphe, on utilisera une matrice d'adjacence M pour laquelle $M_{ij} = w(i,j)$ et $w(i,j)$ représente le poids de l'arête de i vers j. Lorsqu'il n'y a pas d'arc entre deux sommets, on aura $M_{ij} = \infty$.

Exemple –



Colonne j, sommet d'arrivée

Ligne i, sommet de départ

$$\begin{bmatrix} 0 & 1 & 2 & \infty & \infty \\ 2 & 0 & 2 & 1 & \infty \\ \infty & \infty & 0 & \infty & 3 \\ 3 & \infty & \infty & 0 & 1 \\ \infty & \infty & \infty & \infty & 0 \end{bmatrix}$$

Définition – Poids d'un chemin

Soit un un graphe pondéré $G = (V, E, w)$ où V désigne l'ensemble des sommets, E l'ensemble des arêtes et w, la fonction poids définie par $w : E \longrightarrow \mathbb{R}$ ($w(u,v)$ est le poids de l'arête de u vers v).
On appelle poids du chemin C et on note $w(C)$ la somme des poids des arêtes du chemin.
Un chemin de $u \in V$ à $v \in V$ est un plus court chemin s'il n'existe pas de chemin de poids plus petit.

Exemple –

Pour le chemin $C = 0, 1, 2, 4$, on a $w(C) = 6$.
Pour le chemin $C' = 0, 1, 3, 4$, on a $w(C') = 3$.
C' est un plus court chemin.

Définition – Distance

La distance $d(u,v)$ est le poids d'un plus court chemin de u à v. On peut alors noter $d(u,v) = \inf\{w(C) | C \text{ est un chemin de } u \text{ à } v\}$.
Si v n'est pas atteignable depuis u on poser $d(u,v) = \infty$.

Exemple –

Dans le cas précédent, $d(0,4) = 3$, $d(2,4) = 3$ et $d(4,1) = \infty$

Propriété – Sous-optimalité

Soit C un plus court chemin de u à v ainsi que u' et v' deux sommets de C. Alors le sous-chemin de C de u' à v' est aussi un plus court chemin.

Exemple –

$C' = 0,1,3,4$ est un plus court chemin ; donc $C' = 1,3,4$ ou $C' = 0,1,3$ aussi.

Objectif

Soit un graphe pondéré $G = (V, E, w)$ où V désigne l'ensemble des sommets, E l'ensemble des arêtes et w, la fonction poids.
Soit s un sommet de V. L'objectif est de déterminer la liste de l'ensemble des distances entre s et l'ensemble des sommets de V.

Pour répondre à l'objectif, on peut formuler l'algorithme de Dijkstra ainsi.

Entrées : un graphe pondéré donné par liste ou marrice d'adjacence, un sommet `s` du graphe
Sortie : `D` liste des distances entre `s` et chacun des sommets
Initialisation de `D` : `D = n`$\times[\infty]$
Initialisation de `D` : `D[s] =`$[0]$
Initialisation de `T` : `T = n x [False]` liste des sommets traités
Initialisation d'une file de priorité avec le sommet de départ `F = {s}`
Tant que `F` n'est pas vide :
    Recherche du sommet `u` tel que `d[u]` minimal parmi les sommets de `F`
    Pour tout voisin v de u **faire** :
        Si v n'est ni dans T ni dans F **alors**
            Ajouter v à F
        `D`=min $(d[v], d[u] + w(u,v))$
    `T[u] = True`
Renvoyer `D`

Une des étapes qui diffère avec le parcours en largeur notamment, est l'utilisation d'une file de priorité et la recherche du sommet vérifiant $d[u]$ minimal. Cela signifie que lorsqu'on partira d'un sommet `s`, on déterminera alors l'ensemble des distances permettant d'atteindre les voisins de `s`. À l'itération suivante, on visitera alors le sommet ayant la distance la plus faible.

Définition – File de priorité

Une file de priorité est une structure de données sur laquelle on peut effectuer les opérations suivantes :

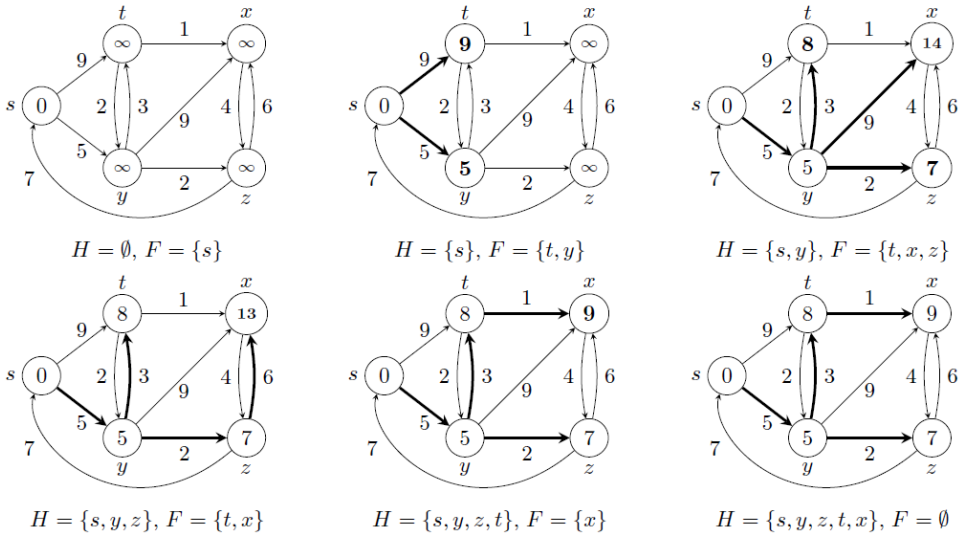
- insérer un élement ;
- extraire l'élément ayant, dans notre cas, la plus petite valeur ;
- tester si la file de priorité est vide ou pas.

Exemple –

Soit une file de priorité comprenant les éléments suivants : `file = [12,1,4,5]`. La file de priorité est dotée d'une méthode `pop` permettant d'extraire la plus petite valeur. Ainsi, `file.pop()` renvoie `1` et la file contient alors les éléments `[12,4,5]`. En réitérant `file.pop()` renverra la valeur `4` et la file contient désormais les éléments `[12,5]`.
Soit une file de priorité comprenant les éléments suivants : `file = [(1,2),(2,5),(0,1)]`. La méthode `pop` permetttra d'extraire le couple pour lequel la première valeur est la plus petite.
Ainsi, `file.pop()` renvoie `(0,1)` et la file contient alors les éléments `[(1,2),(2,5)]` etc.

Exemple –

[Jules Svartz] La figure suivante représente le déroulement de l'algorithme de Dijkstra sur un graphe à 5 sommets, depuis la source s. Pour chaque sommet u on a fait figurer la valeur $d[u]$ à l'intérieur du cercle. Les arcs en gras représentent l'évolution de la liste des prédecesseurs.



On peut donc commencer par implémenter une fonction `cherche_min` permettant de trouver le sommet `i` vérifiant `d[i]` minimal parmi les sommets n'ayant pas été traités.

```
def cherche_min(d, traites):
    """ Renvoie le sommet i vérifiant d[i] minimal et traites[i] faux, s'il
    existe un tel sommet
    tel que d[i] != inf. Sinon, renvoie -1 """
    n=len(d)
    x=-1
    for i in range(n):
        if not traites[i] and d[i] != float('inf') and (x==-1 or d[x]>d[i]):
            x=i
    return x
```

On donne alors l'algorithme de Dijkstra.

```
def dijkstra_mat(G,s):
```

```
    """
    G donné par matrice d'adjacence. Renvoie les poids chemins de plus petits
    poids depuis s.
    """
    n=len(G)
    d = [float('inf')]*n
    d[s]=0
    traites = [False]*n
    while True:
        x=cherche_min(d,traites)
        if x==-1:
            return d
        for i in range(n):
            d[i]=min(d[i], d[x]+G[x][i])
        traites[x]=True
```

Propriété –

Pour n sommets et a arcs, il existe un algorithme de Dijkstra telle de complexité $\mathcal{O}(a + n \log n)$.

Exemple –

Reprendre le graphe précédent et utiliser l'algorithme de Dijkstra en partant du sommet t.

Sources

- Cours de Quentin Fortier.
- Cours de Jules Svartz, Lycée Masséna.
- https://www.youtube.com/watch?v=GC-nBgi9r0U