MPSI

Sem. 2

Informatique

Cours

Ch. 8
Parcours de graphes

E. Durif – X. Pessoles
J.-P. Berne
MPSI

1 Compléments sur le chapitre 6

1.1 Taille d'un graphe

Définition Taille d'un graphe Soit un graphe $G=(V,E)$ composé de sommets V et d'arêtes E. On appelle taille du graphe la quantité $|V|+|E|$.

R On retrouve dans des ouvrages la notation $|V|$ ou $|E|$ correpondant respectivement au nombre de sommets et d'arêtes. Ainsi, si un algorithme est linéaire en la taille du graphe, on pourra utilisé la notation $\mathcal{O}(|V|+|E|)$.

1.2 Formule des degrés

Définition Formule des degrés Soit un graphe $G=(V,E)$ un graphe, alors : $\Sigma_{v\in V} deg(v)=2|E|$.

1.3 Graphe complet

Définition Graphe complet Un graphe complet est un graphe possédant toutes les arêtes possibles.

R

- Le nombre maximal d'arêtes d'un graphe à n sommets est donné par : $\binom{n}{2}$.
- Chaque sommet v on a $d°(v)=n-1$.
- Le nombre d'arêtes est en $\mathcal{O}(|V|^2)$.

1.4 Matrices d'adjacences

Avantages

- Le test d'adjacence de deux sommets se fait en temps constant.
- Dans le cas d'un graphe orienté, obtenir les prédécesseurs n'est pas plus compliqué (ni plus coûteux) que d'obtenir les successeurs.
- Rajouter ou supprimer une arête (ou un arc) peut se faire en temps constant.

Inconvénients

- Ajouter ou supprimer un sommet nécessite un temps proportionnel à n^2.
- Pour récupérer les voisins/successeurs d'un noeud, il faut parcourir toute la ligne de la matrice : l'opération se fait donc en $\mathcal{O}(n)$ [a], ce qui est regrettable si le degré sortant du noeud est petit devant n.
- On consomme une mémoire proportionnelle à $|V|^2$, même quand la taille du graphe est de l'ordre de n (graphe creux).

a. $\Theta(n)$

1.5 Listes d'adjacences

Avantages

- La mémoire utilisée est de l'ordre de $|E|+|V|$, ce qui est nettement mieux que $|V|^2$ si le graphe est creux.
- On a directement accès à la liste des voisins d'un noeud : la parcourir prend un temps proportionnel au degré sortant du noeud.
- Ajouter un noeud peut normalement se faire en temps $|V|$ (si l'ajout n'impose pas de renuméroter les noeuds déjà présents).

Inconvénients

- Le test d'adjacence ne se fait plus en temps constant (mais en temps proportionnel au degré du noeud).
- Si le graphe est dense, on consommera plus de mémoire (d'un facteur constant) qu'avec une matrice d'adjacence.
- Ajouter ou supprimer une arête n'est pas aussi évident que dans une matrice d'adjacence : suivant l'opération précise que l'on souhaite faire, la complexité peut être unitaire ou proportionnelle aux degrés des noeuds impactés.
- Supprimer un noeud n'est pas pratique : le plus simple est de reconstruire entièrement le graphe (en un temps $\mathcal{O}(|E|+|V|)$).

2 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.*

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 procjes 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.

2.1 Parcours en largeur

2.1.1 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.

```
def bfs(G, s) -> None:
    """
    G : graphe sous forme de dictionnaire d'adjacence
    s : sommet du graphe (Chaine de caractere du type "S1").
    """
    visited = {}
    for sommet,voisins in G.items():
        visited[sommet]=False
    # Le premier sommet àvisiter entre dans la file
    q = deque([s])
    while len(q) > 0:
        # On visite la tête de file
        u = q.pop()
        # On vérifier qu'elle n'a pas été visitée
        if not visited[u]:
            # Si on l'avait pas visité, maintenant c'est le cas :)
            visited[u] = True
            # On met les voisins de u dans la file
            for v in G[u]:
                q.appendleft(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 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.

2.1.2 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
```

▪

2.2 Parcours en profondeur

2.2.1 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.

2.2.2 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