

Graphes : Parcours en largeur (BFS)

Quentin Fortier

May 16, 2022

Une **file** est une structure de donnée possédant les opérations :

- Ajout d'un élément à la fin de la file.
- Extraction (suppression et renvoi) de l'élément au début de la file.
Ainsi, c'est toujours l'élément le plus ancien qui est extrait.



On pourrait utiliser une liste `L` Python comme une file avec `L.append(e)` pour ajouter à la fin et `L.pop(0)` pour supprimer le 1er élément.

Mais `L.pop(0)` serait en $O(n)$ (quand on supprime le 1er élément il faut décaler tous les autres), ce qui n'est pas satisfaisant.

On pourrait utiliser une liste `L` Python comme une file avec `L.append(e)` pour ajouter à la fin et `L.pop(0)` pour supprimer le 1er élément.

Mais `L.pop(0)` serait en $O(n)$ (quand on supprime le 1er élément il faut décaler tous les autres), ce qui n'est pas satisfaisant.

Il est plus efficace d'utiliser la classe deque (pour *doubly-ended queue*) du module collections, qui permet d'ajouter à la fin avec append et de supprimer au début avec popleft :

```
from collections import deque

q = deque() # file vide
q.append(4)
q.append(7)
q.popleft() # renvoie 4
q.append(-5)
q.popleft() # renvoie 7
```

Parcours en largeur (BFS) : Avec file

Le parcours en largeur utilise une file q (et ressemble au DFS avec une pile) :

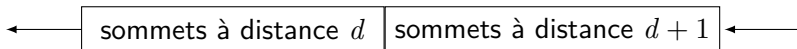
```
def bfs(G, s):  
    visited = [False]*len(G)  
    q = deque([s])  
    while len(q) > 0:  
        u = q.popleft()  
        if not visited[u]:  
            visited[u] = True  
            for v in G[u]:  
                q.append(v)
```

Parcours en largeur (BFS) : Avec file

Le parcours en largeur utilise une file q (et ressemble au DFS avec une pile) :

```
def bfs(G, s):  
    visited = [False]*len(G)  
    q = deque([s])  
    while len(q) > 0:  
        u = q.popleft()  
        if not visited[u]:  
            visited[u] = True  
            for v in G[u]:  
                q.append(v)
```

La file est toujours de la forme :

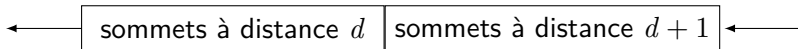


Parcours en largeur (BFS) : Avec file

Le parcours en largeur utilise une file q (et ressemble au DFS avec une pile) :

```
def bfs(G, s):  
    visited = [False]*len(G)  
    q = deque([s])  
    while len(q) > 0:  
        u = q.popleft()  
        if not visited[u]:  
            visited[u] = True  
            for v in G[u]:  
                q.append(v)
```

La file est toujours de la forme :



Les sommets sont donc **traités par distance croissante** à s : d'abord s , puis les voisins de s , puis ceux à distance 2

Parcours en largeur (BFS) : Avec 2 couches



Une variante du BFS utilise deux listes : `cur` pour la couche courante, `next` pour la couche suivante.

```
def bfs(G, s):
    visited = [False]*len(G)
    cur, next = [s], []
    while len(cur) + len(next) > 0:
        if len(cur) == 0:
            cur, next = next, []
        u = cur.pop()
        if not visited[u]:
            for v in G[u]:
                next.append(v)
```

Parcours en largeur (BFS) : Application au calcul de distance

Question

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

Parcours en largeur (BFS) : Application au calcul de distance

Question

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

Stocker des couples (sommet, distance) dans la file et stocker les distances dans un tableau `dist` (`dist[v]` = distance de s à v) :

Parcours en largeur (BFS) : Application au calcul de distance

Question

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

Stocker des couples (sommet, distance) dans la file et stocker les distances dans un tableau `dist` (`dist[v]` = distance de s à v) :

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

Parcours en largeur (BFS) : Plus courts chemins

Question

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

Parcours en largeur (BFS) : Plus courts chemins

Question

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

Stocker $\text{pred}[v]$ = prédécesseur de v dans le parcours :

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

Parcours en largeur (BFS) : Plus courts chemins

Question

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

On peut ensuite remonter les prédécesseurs de v à s :

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

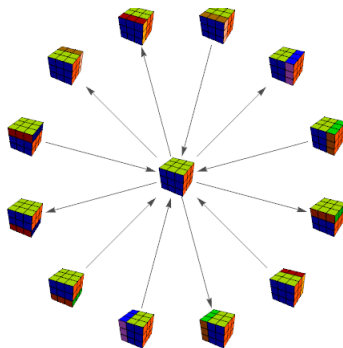
Parcours en largeur (BFS) : Plus courts chemins

Application : résoudre un Rubik's Cube avec le nombre minimum de coups.

Parcours en largeur (BFS) : Plus courts chemins

Application : résoudre un Rubik's Cube avec le nombre minimum de coups.

- 1 Sommets = configurations possibles du Rubik's Cube.
- 2 Arêtes = mouvements élémentaires.



Parcours en largeur (BFS) : Plus courts chemins

Application : résoudre un Rubik's Cube avec le nombre minimum de coups.

- ① Sommets = configurations possibles du Rubik's Cube.
- ② Arêtes = mouvements élémentaires.

Théorème (2010)

Le **diamètre** (distance max entre deux sommets) du graphe des configurations du Rubik's Cube est 20.

⇒ on peut résoudre n'importe quel Rubik's Cube en au plus 20 mouvements.