

Parcours de graphes

$G(V, E)$ avec V sommets et E arêtes

$V = \{0, 1, 2, 3, 4, 5, 6\}$

$E = \{(0, 1), (0, 3), (1, 2), (1, 5), (3, 4), (4, 5)\}$

avec la convention que si $(s, d) \in E$ alors $(s, d) = (d, s)$

(par exemple on considère que $(4, 3) \in E$)

On va calculer des complexités en fonction de $|V|$ et $|E|$.

$$|E| \leq (|V| - 1) = (|V| * (|V| - 1)) / 2 \leq |V|^2 / 2$$

$$|E| = O(|V|^2)$$

$$\Theta(|V|^3 + |E|) = \Theta(|V|^3)$$

Dans un graphe connexe (chaque sommet a un chemin vers n'importe quel autre), il y a au moins $|E| \geq |V| - 1$ (sinon des sommets ne sont pas connectés)

$|E| = \Omega(|V|)$ si graphe connexe.

Rappel $\sum_{v \in V} (\deg(v)) = 2 * |E| = \Theta(|E|)$

DFS = depth-first search = parcours en profondeur

Liste d'adjacence

```
1 0 -> 1 -> 3
2 1 -> 0 -> 2 -> 5
3 2 -> 1
4 3 -> 0 -> 4
5 4 -> 3 -> 5
6 5 -> 1 -> 4
7 6 /
```

En python:

```
1 edges = [[1, 3], [0, 2, 5], [1], [0, 4], [3, 5], [1, 4], []]
2 len(edges) # donne |V|
```

Algo:

```
1 def dfs(adj):
2     n = len(adj)
3     seen = [False] * n # a-t-on vu chaque sommet
```

```
4 def rec(start):
5     print(start)
6     seen[start] = True
7     for d in adj[start]:
8         if not seen[d]:
9             rec(d)
10    for d in range(n):
11        if not seen[d]:
12            rec(d)
13
14    dfs(edges)
15    # affiche 0, 1, 2, 5, 4, 3, 6
```

Version itérative du DFS

pile de paires (i, j) où i le sommet (edges[i]), j le successeur (edges[i][j])

```
1 def dfs_item(adj):
2     n = len(adj)
3     seen = [False] * n
4     stack = []
5     for start in range(n):
6         if seen[start]:
7             continue
8         stack[(start, 0)]
9         while stack:
10            src, pos = stack.pop()
11            if pos == 0:
12                print(src)
13                seen[src] = True
14            if pos == len(adj[src]): # il ne reste pas des successeurs
15                continue
16            stack.append((src, pos + 1))
17            if not seen[adj[src][pos]]:
18                stack.append((adj[src][pos], 0))
```

Breadth-First Search (BFS) = parcours en largeur

```
1 def bfs(adj):
2     n = len(adj) #Theta(1)
3     seen = [False] * n #Theta(|V|)
4     for start in range(n): #Theta(|V|)
5         if seen[start]: #Theta(|V|)
6             continue #O(|V|)
```

```

7     queue = [start] #O(|V|)
8     seen[start] = True #a
9     while queue: #Theta(|V|)
10        src = queue.pop(0) #O(|V| * |V|) remplacer la liste "queue" par
            collections.deque, et la complexité tombe en Theta(|V|)
11        print(src) #Theta(|V|)
12        for dst in adj[src]: #Theta(|E|)
13            if not seen[dst]: #Theta(|E|)
14                seen[dst] = True #b a + b = |V|
15                queue.append(dst) #O(|E|)
16
17 bfs(edges)
18 # affiche 0, 1, 3, 2, 5, 4, 6

```

BFS en $\Theta(|V| + |E|)$ = linéaire (pour un graphe)

Calcul de distance depuis un sommet

```

1 from collections import deque
2
3 def distmap(adj, start):
4     n = len(adj)
5     dist = [None] * n #Theta(|V|)
6     queue = deque([start])
7     dist[start] = 0
8     while queue: #O(|V|)
9         src = queue.popleft()
10        for dst in adj[src]:
11            if dist[dst] is None:
12                dist[dst] = d + 1
13                queue.append(dst)
14    return dist #Theta(|V|) + O(|E|)

```

Que faut-il changer pour travailler sur un graphe orienté ?

=> Rien dans l'algo. C'est juste "adj" qui change (et n'est plus symétrique)

Que faut-il changer si le graphe est pondéré par des distances ?

Dijkstra

Plus court chemin des graphes pondérés avec poids ≥ 0

$\omega(x, y)$ poids de x arc (x, y)

Dijkstra (G(V, E), start)

$\forall v \in V, D[v] \leftarrow \infty$

$D[start] \leftarrow 0$

queue $\leftarrow \{start\} \leftarrow$ file de priorité (tas) ordonné par rapport à D

while queue $\neq \emptyset$

min \leftarrow queue.popmin()

d $\leftarrow D[min]$

for y \in successor(min):

d' $\leftarrow d + \omega(min, y)$

old $\leftarrow D[y]$

$D[y] \leftarrow \min(old, d')$

if old == ∞ :

queue.insert(y)

else

queue.update(y)

return D