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 considére que (4, 3) $\in$ E)

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

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

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

Θ($|V|^3$ + |E|) = Θ($|V|^3$)

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| = Ω(|V|) si graphe connnexe.

Rappel $\sum$(v$\in$V)(deg(v)) = 2 * |E| = Θ(|E|)

DFS = depth-first search = parcours en profondeur

Liste d'adjacence

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

En python:

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

Algo:

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

```
  def rec(start):
    print(start)
    seen[start] = True
    for d in adj[start]:
      if not seen[d]:
        rec(d)
  for d in range(n):
    if not seen[d]:
      rec(d)

dfs(edges)
# 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])

```
def dfs_item(adj):
  n = len(adj)
  seen = [False] * n
  stack = []
  for start in range(n):
    if seen[start]:
      continue
    stack[(start, 0)]
    while stack:
      src, pos = stack.pop()
      if pos == 0:
        print(src)
        seen[src] = True
      if pos == len(adj[src]): # il ne reste pas des successeurs
        continue
      stack.append((src, pos + 1))
      if not seen[adj[src][pos]]:
        stack.append((adj[src][pos], 0))
```

Breadth-First Search (BFS) = parcours en largeur

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

```
    queue = [start] #O(|V|)
    seen[start] = True #a
    while queue: #Theta(|V|)
      src = queue.pop(0) #O(|V| * |V|) remplacer la liste ”queue” par
         collections.deque, et la complexité tombe en Theta(|V|)
      print(src) #Theta(|V|)
      for dst in adj[src]: #Theta(|E|)
        if not seen[dst]! #Theta(|E|)
          seen[dst] = True #b a + b = |V|
          queue.append(dst) #O(|E|)

bfs(edges)
# affiche 0, 1, 3, 2, 5, 4, 6
```

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

Calcul de distance depuis un sommet

```
from collections import deque

def distmap(adj, start):
  n = len(adj)
  dist = [None] * n #Theta(|V|)
  queue = deque([start])
  dist[start] = 0
  while queue: #O(|V|)
    src = queue.popleft()
    for dst in adj[src]:
    if dist[dst] is None:
      dist[dst] = d + 1
      queue.append(dst)
  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 ?

Djikstra

Plus court chemin des graphes pondérés avec poids ≥ 0)

ω(x, y) poids de x arc (x, y)

Djikstra (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 + ω(min, y)

old $\leftarrow$ D[y]

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

if old == ∞:

queue.insert(y)

else

queue.update(y)

return D