Partie 6 : Algorithmique des Graphes

Génération de graphes de contrôle

$V = [0, n-1]$

```python
n = 7
E = {
    (0,4),
    (1,2),(1,3),(1,4),(1,6),
    (2,0),(2,3),
    (4,6),(5,2),(5,3),
    (6,1)
}
```

4. Représentation

Matrice d'adjacence

	0	1	2	3	4	5	6
0					1		
1			1	1	1		1
2	1			1			
3							
4							1
5			1	1			
6		1					

```c
// Le type serait int array array, en C ce serait :
(int n, int** g)
```

Liste d'adjacence

```python
[
    [4], # Voisins de 0
    [2,3,4,6], # Voisins de 1
    [0,3], # dots
    [],
    [6],
    [2,3],
    [1],
]
```

On gagne de la place en évitant de remplir de 0. Cependant chaque ligne n'a pas la même taille et on perd le retour des graphes orientés.

Pour l'implémenter en Caml pas de souci, pour le C il y a plusieurs façons de faire :

- ```c
  // Tableau de structs, on oublie c'est galère
  struct nadine = {int len; int* a;};
  ```
  ```

- Premier élément de chaque tableau indique la taille
- Sentinelle, savoir quand s'arrêter

La taille de la liste est $log(n)m$

Dictionnaire d'adjacence

Si $V \neq [0, n-1]$

$g[i] \rightarrow$ liste des voisins de i

$g[\text{Paris}] = [\text{Lille}, \text{Nantes}, ...]$

Caractérisation des parcours

Largeur

Un parcours en largeur visite les sommets par distance croissante à la source, c'est-à-dire que tous les sommets à distance d de u0 seront vus avant tous ceux à distance d+1.

Profondeur

Parcours de graphes

Parcours en profondeur

On travaille en liste d'adjacence

```python
# depth first search
def dfs(G:Graph):
    n = len(G)
    visited = [False]*n # Petits cailloux

    def visite(u:Sommet):
        if not visited[u]: # Si pas caillou
            visited[u] = True # On met caillou
            print(u)
            for v in g[u]:
                visite(v)
            foo()


    for u in range(n):
        visite(u)
```

Variantes / Extensions

1. Un tableau parent de longueur n tel que parent[n] = le sommet depuis lequel on a visité u. On peut obtenir un arbre de parcours.
2. Pour parcourir toutes les composantes connexes on appel visite sur tous les sommets et non pas juste sur un seul.

En impératif

Le début est celui du parcours en largeur (voir la suite) puis on remplace les files par des piles.

```python
    dots
    visited = [False]*n
    # on supprime cette ligne
    # visited[u0] = True

    while !is_empty(Q):
        u = Q.pop()
        if not visited[u]:
            print(u)
            visited[v] = True
            for v in G[u]
                Q.push(v)
```

Cependant certains éléments apparaîtrons plusieurs fois, c'est inévitable.

Complexité en $O(n+m)$ quand même.

Parcours en largeur

```python
from queue import crate_queue, is_empty


# breadth first search
def bfs(G:Graph):
    u0:Sommet # Un premier élément (la racine) à déterminer
    Q = create_queue()
    Q.enqueue(u0)
    visited = [False]*n
    visited[u0] = True
    # Invariant : u appartient à Q implique visited[u] is True

    while !is_empty(Q):
        u = Q.dequeue()
        print(u)
        for v in G[u]:
            if not visited[v]:
                visited[v] = True
                Q.enqueue(v)
```

Éventuellement, réitérer sur un autre u tel que visited[u0] = False.

Remarque : Chaque sommet de la composante connexe de u0 passe exactement une fois dans la file. Cependant ce n'est pas toujours le cas !

$\Rightarrow$ On a une complexité en 0(n+m) car la boucle while est itérée exactement n fois (si le graphe est connexe). Par conséquent, la complexité est $\sum_{u\in V} 1 + d_+(u) = O(n+m)$