Cours 2 Sophie Laplante 22/09/14

Pour manipuler `G` dans un programme, on peut stocker la relation d'adjacence (c-à-d pour chaque $x, y \in S$ est-ce que (x, y) est une arrête ou non ?) dans un tableau nxn, T

```
T[x][y] = Vrai si x -> y
          Faux sinon
```

Mathématiquement cela correspond à la **MATRICE D'ADJACENCE**

$$M[x,y] \begin{cases} 1 & \text{si } x(x,y) \in G \\ 0 & \text{sinon} \end{cases}$$

Differente notation pour la relation d'adjacence $(x, y) \in A$ même chose que $x \sim y$

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

$$1 \sim 3 \mathrm{OUI}$$
$$1 \sim 4 \mathrm{NON}$$

$$\mathrm{A} = \{(0,2),(1,3),(4,4)\}$$

Pour G $(1,3) \in A$
$(1,4) \notin A$

1. Definition

Dans un graph `G = (S, A)` un *chemin* de $s \in \mathrm{S}$ à $t \in \mathrm{S}$ est une seccession de sommet (s= s0, s1, s2, ... sk = t)

$tq, (s_i, s_{i+1}) \in \mathrm{A}, 0 \leq i < k$ ce chemin de longueur k

1.1. Un graph est *connexe*

Si $\forall s \in \mathrm{S}, \forall t \in \mathrm{S}$ Il existe un chemin de s à t

1.2. Une *composante connexe*

est un ensemble de sommets $\mathrm{T} \subseteq S tq \forall s, t \in \mathrm{T}$ Il existe un chemin de s à t et T n'est pas inclu dans une composante connexe plus grande

ex: dans G Les composante connexe sont {0,2} et {1, 3, 4} **mais** {1, 3} n'est pas une composante connexe

2. Algo UNION-FIND

C'est une structure de données qui permet d'implémenter deux opération sur un ensemble de point **S** muni d'une relation d'equivalence $u \sim v$ (pensez a l'accesibilité dans un graph)

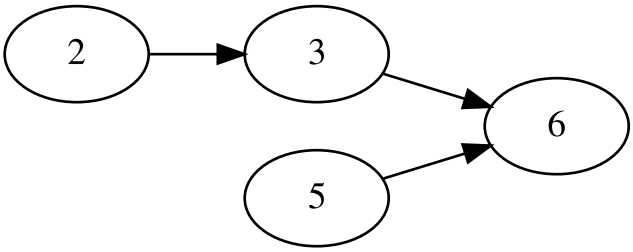
1. UNION (u,v) Met en relation u et v

2. FIND (u, v) Détermine si u, v sont en relation

2.1. Implémentation

1. Idée Chaque point note dans un tableau son *père* quand on fait **l'union** l'ancêtre de l'un devient le père de l'ancêtre de l'autre

Union (2,3)
Union (5,6)
Union (2,6) l'ancêtre de 2 vers l'ancêtre de 6



Find (2,3) Ancêtre(2) = Ancêtre(3) = 6

```
def find(u, v):
    return (root(u) == root(v))

def root(u):
    while u != Pere[u]:
        u = Pere[u]
    return u

def union(u, v):
    if find(u, v) == False:
        Pere[root(u)] = root(u)
```

ATTENTION Initialement le tableau père doit être initialisé Père[u] = u pour tout u in S

Pere

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

Union(2,3)

0	**1**	**3**	**3**	**4**	**5**	**6**

Union(0,5)

5	**1**	**3**	**3**	**4**	**5**	**6**

Union(3,6)

5	**1**	**3**	**6**	**4**	**5**	**6**

find 2,6 OUI

Union(3,0)

5	**1**	**3**	**6**	**4**	**5**	**5**

Quel est le coût de cet algo?
find: 2x le cout de root
root: pire scénario on doit parcourir le tableau cout(root) longueur du plus long chemin dans le *graph de parenté*
union: Cout(find)+2*cout(root)

2.2. Variante 1

On tente d'équilibrer les arbre. On maintient pour chaque point le nombre de point dont il est l'ancetre
Au moment de l'union, on raccroche le plus petit au plus grand

```
find ne change pas
root ne change pas
```

Initialisation:

Il faut ajouter un tableau taille[u] = 1

```
def union2(u,v):
    if find(u,v)==False:
        if taille[root(u)] > taille[root(v)]:
            Pere[root(u)] = root(u)
            taille[root(u)] += taille[root(v)]
        else:
            Pere[root(u)] = root(v)
            taille[root(v)] += taille[root(u)]
```