GESTION D'ENSEMBLES DISJOINTS

APPLICATION À L'ALGORITHME DE KRUSKAL

Ce problème est aussi connu sous le nom de **UNION - APPARTENANCE**, ou **UNION - FIND**, ou encore de **MERGE - FIND**, il s'agit en gros de gérer des ensembles disjoints à partir de n éléments indépendants

I - LE PROBLÈME

Étant donnés n éléments distincts notés 1, 2, …, n, le problème consiste à gérer le regroupement de certains d'entre eux en sous-ensembles de plus en plus gros, que nous appellerons **CLASSES**

Initialement, il y a n classes : {1}, {2}, …, {n}

Chaque étape correspond au regroupement de deux éléments, x et y par exemple, ce que l'on notera **x ≡ y**, il s'agit alors de déterminer les classes C_x et C_y de x et y, puis d'en faire l'union $C_x \cup C_y$, si elles sont différentes. La suite des classes évolue donc au cours de regroupements, mais les sous-ensembles correspondants restent disjoints, on peut donc les assimiler à des classes d'équivalence

Par exemple, si n = 7, on a au départ {1}, {2}, {3}, {4}, {5}, {6} et {7}. Les regroupements successifs de 1 et 2, puis de 5 et 6, puis de 3 et 4 et enfin de 1 et 4 donneront finalement les classes {1, 2, 3, 4}, {5, 6} et {7}

Le problème est donc tout d'abord d'initialiser les classes avec les n éléments, nous appellerons cela **INITIALISATION**, il faut créer la classe $C_x = \{x\}$ pour chaque x. Puis à chaque étape de réaliser les opérations **APP(x)** et **APP(y)**, qu'on appellera **APPARTENANCE**, qui donnent les noms des classes auxquelles appartiennent x et y et **UNION** qui réalise l'union (ou la fusion) des classes C_x et C_y et place le résultat dans C_x ou dans C_y, il faut choisir, soit arbitrairement, soit suivant un certain critère, voir suite

Et le schéma général de l'algorithme, pour une suite de regroupements x ≡ y, est donc :

pour chaque x ≡ y **faire**
    C_x = APP(x)
    C_y = APP(y)
    UNION(C_x, C_y)

Plusieurs implémentations sont possibles pour réaliser ces opérations, nous en étudions quelques-unes

II - UNE PREMÈRE SOLUTION

Une première solution consiste à utiliser un unique tableau **Classe(1)**, **Classe(2)**, …, **Classe(n)** où Classe(x) est un entier de {1, 2, …n} représentant la classe de x.

Initialement, on prendra Classe(x) = x pour tout élément, **INITIALISATION** est donc réalisée par :

pour x = 1 **à** n **faire** Classe(x) = x

Pour chaque x ≡ y, **APPARTENANCE** consiste à consulter simplement Classe(x) et Classe(y), donc **UNION**, qui réalise $C_x = C_x \cup C_y$, peut s'écrire :

si Classe(x) ≠ Classe(y) **alors**
  pour i = 1 **à** n **faire**
    si Classe(i) = Classe(y) **alors** Classe (i) = Classe(x)

En effet on passe en revue tous les éléments et tout élément de $\mathbf{C_y}$ devient un élément de $\mathbf{C_x}$

L'**INITIALISATION** nécessite n étapes élémentaires, ainsi que chaque **UNION**
Une suite de m regroupements sera donc en O(m.n)

III - UNE SOLUTION PLUS EFFICACE

Dans la solution précédente, pour chaque union, on examine systématiquement tous les éléments, même ceux qui ne sont pas concernés. Pour éviter ces consultations inutiles, il suffit, pour chaque classe, de stocker l'ensemble de ses éléments, et uniquement ceux là

Au tableau Classe(1), …, Classe(n), on ajoutera un tableau de listes indicé par le nom des classes : **Liste(1)**, **Liste(2)**, …, **Liste(n)** où Liste(i) est un pointeur sur une liste contenant les éléments de la classe i

La classe d'un élément sera l'élément qui est choisi comme représentant de la classe. Initialement, on a bien sûr Liste (i) = {i}, et à chaque étape on pourra simplement prendre l'élément situé en tête de liste

Les opérations pour **INITIALISATION** et **UNION** sont alors respectivement :

pour i = 1 **à** n **faire**
  Classe(i) = i
  Liste(i) = {i}

si Classe(x) = C_x ≠ Classe(y) = C_y **alors**
  Liste(C_x) = Liste(C_x) ∪ Liste(C_y)
  Liste(C_y) = ∅
  pour chaque élément z de Liste(C_y) **faire** Classe(z) = Classe(x)

APPARTENANCE, comme précédemment, consiste tout simplement à consulter Classe(x) et Classe(y)

INITIALISATION se fait en Θ(n) et **UNION** en $\Theta(|C_y|)$, pour éviter de parcourir inutilement Liste(C_x), on rajoutera un pointeur sur la fin de liste. Cela améliore les performances en moyenne par rapport à la solution précédente, mais une suite de m regroupements pourra se faire en O(m.n) dans le plus mauvais cas

L'intérêt de cette solution est que l'on peut encore améliorer cette complexité en faisant une **UNION pondérée** ou **différenciée** qui consiste à rajouter C_y à C_x quand $|C_x| \geq |C_y|$, ou l'inverse sinon. Il faut alors rajouter dans le tableau Liste une case où l'on stockera la taille de la liste correspondante

Ainsi, **UNION** pourra se faire en $\mathbf{O(\min(|C_x|, |C_y|))}$

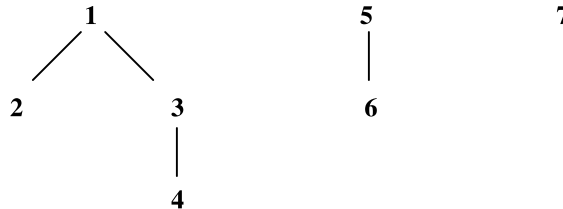
Tout élément changeant de classe se retrouve dans une classe de taille au moins double de celle où il se trouvait précédemment. Comme il y a initialement n classes, un élément ne peut changer de classe plus de $\log_2 n$ fois. Ainsi m regroupements pourront se faire en O(mlogn)

IV – FORÊT D'ARBORESCENCES

Cette méthode consiste à **représenter chaque classe par une arborescence dont la racine servira de représentant**, donc l'ensemble des classes sera une forêt

Dans cette solution, un seul tableau **Forêt(1)**, **Forêt(2)**, …, **Forêt(n)** suffit à tout représenter : pour tout élément x, non racine, Forêt(x) = y si y est le père de x dans une arborescence et Forêt(x) = - k si x est racine, où k est le nombre d'élément d'éléments de l'arborescence

Avec l'exemple donné en introduction, on a :



Le tableau **Forêt** correspondant, indicé de 1 à 7, sera alors :

- 4	**1**	**1**	**3**	**- 2**	**5**	**- 1**

INITIALISATION s'écrit alors :

pour i = 1 **à** n **faire** Forêt(i) = - 1

APP(x) consistera alors, partant de x, à « remonter » jusqu'à la racine :

C_x = x
tant que Forêt(C_x) > 0 **faire** C_x = Forêt(C_x)

UNION entre C_x = APP(x) et C_y = APP(y) peut se faire en rajoutant C_y comme fils de C_x, d'où :

Forêt (C_x) = Forêt(C_x) + Forêt (C_y)
Forêt(C_y) = C_x

Mais là aussi on peut obtenir une composante réduite à une simple chaîne, une suite de m regroupements peut donc dégénérer en O(m.n). Pour remédier à ce problème il suffit de tenir compte de la taille des arborescences et de rajouter la racine de la plus petite comme fils (fille ?) de la racine de la plus grande :

si Forêt(C_x) ≤ Forêt(C_y) **alors** Forêt (C_x) = Forêt(C_x) + Forêt (C_y) ; Forêt(C_y) = C_x
sinon Forêt (C_y) = Forêt(C_x) + Forêt (C_y) ; Forêt(C_x) = C_y

Dans l'exemple précédent, si le regroupement suivant est **4 ≡ 6**, le tableau **Forêt** devient, car le sommet 5 est désormais fils de 1 :

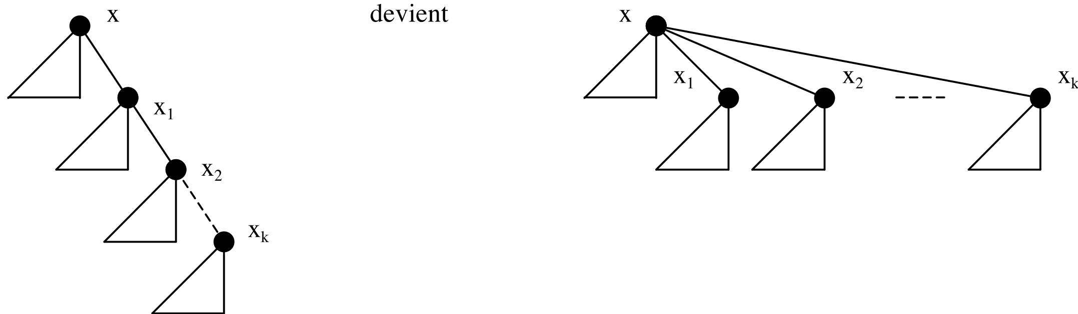
- 6	**1**	**1**	**3**	**1**	**5**	**- 1**

Alors une suite de m regroupements peut se faire en **O(mlogn)**, en effet, on peut montrer que la hauteur de toute arborescence est toujours au plus $\log_2 n$

V - COMPRESSION DE CHEMINS

On peut encore améliorer les performances en utilisant la « **Compression de Chemin** ». L'idée est la suivante : lors du parcours d'un élément x à la racine dans la fonction appartenance, chacun des sommets du chemin parcouru est rendu fils de la racine

Schématiquement :



Cette compression peut se faire lors de l'appel de la fonction **APP(x)**, de la façon suivante :

C_x = x
tant que Forêt(C_x) > 0 **faire** Empiler(C_x) ; C_x = Forêt(C_x)
pour tout élément z de la pile **faire** Forêt(z) = C_x

On utilise une pile pour stocker les éléments du chemin allant de x à la racine

Cette compression « écrase » l'arborescence en hauteur qui aura pour effet d'accélérer la fonction d'appartenance. Les calculs de complexité sont difficiles, mais l'on a le résultat suivant :

Théorème

La compression de chemins avec l'union différenciée pour m regroupements est en O(m.α(n))

La fonction **α(n)** est définie sur les entiers par : **$\alpha(n) = \min \{ x \text{ entier} \mid n \leq A(x, x) \}$** où A est la **Fonction d'Ackermann**, d'une croissance très rapide, qu'on peut définir pour tout couple d'entiers positifs par

A(0, y) = 1
A(1, 0) = 2
A(x, 0) = x + 2 pour x ≥ 2
A(x, y) = A(A(x – 1, y), y – 1), pour x, y ≥ 2

On peut montrer que :

- A(x, 1) = 2x, pour x ≥ 1
- A(x, 2) = 2^x
- A(x, 3) = « x élévations à la puissance de 2 » , pour x ≥ 1
- A(0, 4) = 1, A(1, 4) = 2, A(2, 4) = 2^2, A(3, 4) = 2^{16} = 65 536, A(4, 4) = « 65 536 élévations à la puissance de 2 », …

Nous avons : **α(1) = α(2) = 1**, car A(1, 1) = 2 ; **α(3) = α(4) = 2**, car A(2, 2) = 4 ; **α(5) = … = α(16) = 3**, car A(3, 3) = 16, et **α(n) ≤ 4** pour des valeurs « raisonnables » de n

α est donc une fonction de croissance extrêmement lente, qu'on peut considérer comme constante pour des tailles usuelles de données

VI – APPLICATION À L'ALGORITHME DE KRUSKAL

L'algorithme de **KRUSKAL** qui construit un **Arbre Recouvrant Minimal** d'un graphe simple non orienté valué **G = (X, E, v)** procède en deux temps :

- Trier les arêtes par ordre de coût croissant
- Considérer les arêtes xy une à une dans cet ordre : toute arête ne formant pas de cycle avec les arêtes déjà retenues est gardée ; l'algorithme stoppe dès que l'on a n – 1 arêtes

La première étape peut se faire en temps en O(m.logm) = O(m.logn), car $m \leq n^2$, tout le problème réside donc dans le test « d'**acyclicité** » de la deuxième étape : il faut tester si les deux sommets x et y de l'arête à tester sont dans le même sous arbre, autrement dit dans la même composante connexe de la forêt en construction. Cela peut coûter globalement O(m.n), soit $O(n^3)$, complexité moins bonne que celle de la première étape

Il est possible de faire bien mieux en remarquant que la deuxième étape est exactement un problème de type **UNION – FIND**, il suffit en effet de noter que l'on a à gérer des classes d'équivalence, chaque classe correspondant aux sommets des composantes connexes de la forêt. Ainsi les méthodes développées précédemment peuvent s'appliquer

Pour finir, remarquons qu'il faut aussi stocker l'arbre en construction, une simple liste d'arêtes peut suffire, ou, mieux, un tableau de listes d'adjacence, l'espace sera en $\Theta(n)$, dans les deux cas**.** Stocker l'arbre dans une matrice d'adjacence n'est pas judicieux, celle-ci nécessitant un temps et un espace en n^2, pour une structure d'arbre n'ayant que n – 1 arêtes

En conclusion, en prenant en compte le stockage du graphe G en tableau de listes d'adjacences :

- Le Tri des arêtes peut se faire en **Θ(m.logn)** en temps (Tri par Fusion, par exemple) et en **Θ(n + m)** en espace
- La construction de l'Arbre peut se faire en **O(m.α(n))** en temps en**Θ(n + m)** en espace

Si G est donné par sa matrice d'adjacence, la complexité en espace sera en **$\Theta(n^2)$**