Projet
Cours : Structures de Données

E. Soutil

ENSIIE

2024-2025

Plan

Plan

Date de remise et modalités

- Le projet est à réaliser seul ou en binôme
- Langage imposé : Java
- Attendus :
  - Un rapport de quelques pages précisant les points du projet qui ont été traités ainsi que les comparaisons en temps de calcul portant sur les différentes structures de données utilisées
  - Les sources de votre programmes (projet Java ou à défaut les différentes classes utilisées)
- Date limite de remise du projet final : 04 juillet 2025 à 23 :59
- Envoyer une archive zip de votre projet (rapport + projet Java) à l'adresse eric.soutil@lecnam.net en précisant explicitement les prénoms et noms des deux étudiants du binôme dans le mail, faute de quoi seul l'expéditeur sera noté.

Plan

Description du projet (1/3)

- Le but du projet est de mesurer combien le choix d'une struture de données peut influer sur la résolution d'un problème donné.
- Nous nous intéressons dans ce projet à la résolution d'un problème classique de graphes : la recherche d'un arbre couvrant de poids minimal dans un graphe non orienté connexe pondéré.
- L'algorithme utilisé pour la résolution est l'algorithme de Kruskal.
- 3 structures de données différentes seront implémentées pour la résolution du problème. Les 2 premières utilisent des listes chaînées (et sont relativement similaires). La dernière utilise une structure arborescente.
- Il faudra résoudre le problème de 3 façons différentes (en utilisant les 3 structures de données) et comparer les temps de résolution obtenus avec chacune des méthodes sur différentes instances du problème.

Description du projet (2/3)

- Il faudra générer des instances du problème. Une instance est un graphe connexe pondéré de n sommets et m arêtes. On générera aléatoirement 2 graphes pour chacune des tailles de graphe suivante : $n = 10$, $n = 1000$, $n = 10000$. Pour une valeur donnée de n, le premier graphe devra contenir un nombre d'arêtes égal à $3n$ (graphe peu dense), le second graphe devra contenir un nombre d'arêtes égal à $n^2/3$ (graphe dense). Les poids des arêtes devront être générés aléatoirement entre 1 et 1000. Il y aura donc 6 instances à générer et à résoudre, chacune de 3 façons différentes.
- Pour être sûr de générer un graphe connexe, on imposera que les arêtes [1,2], [2,3], [3,4], ..., [n-1,n] soient systématiquement présentes.

Description du projet (3/3)

- Un rapport devra présenter les résultats obtenus en temps de calcul, en comparant pour chacune des instances les temps obtenus avec chacune des 3 structures de données. Il conviendra d'indiquer le temps de résolution global de l'instance, mais également le temps consacré spécifiquement au tri des arêtes et le temps de construction de la solution par union-fond (cf ce qui suit).

Plan

Plan

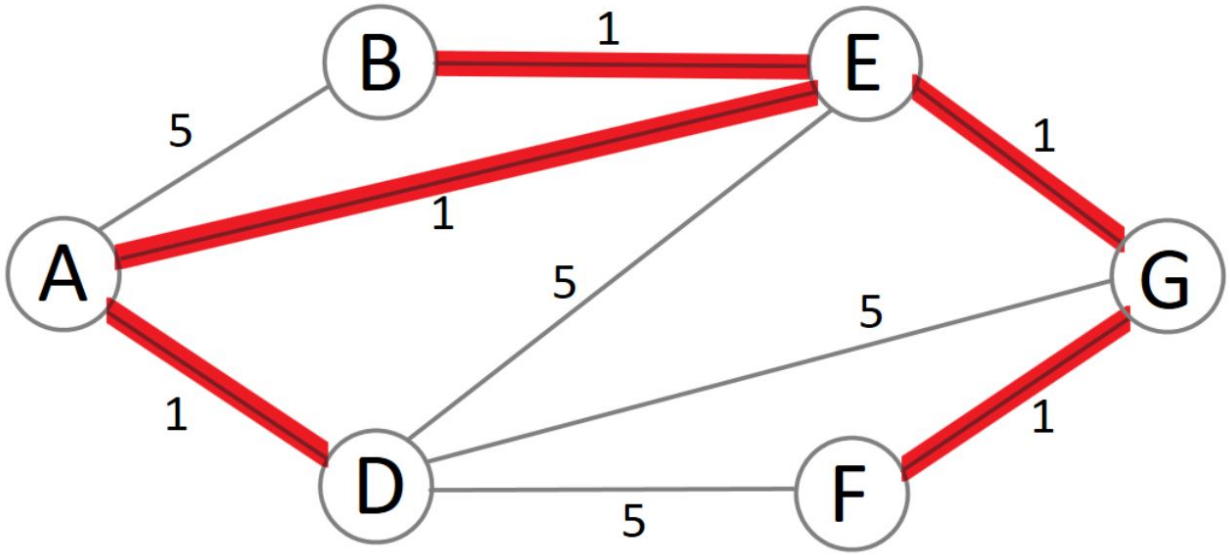
Problème de l'arbre couvrant de poids minimal

Définition du problème

- **Définition du problème** : relier des objets avec une longueur totale minimale des liens
- **Graphe valué** associé :
  - sommets = objets
  - arête = lien possible
  - poids d'une arête = longueur du lien

Arbre couvrant de poids minimal

Un exemple et sa solution



- En rouge, la solution optimale :
  - sans cycle et connexe → ARBRE
  - passant par tous les sommets → COUVRANT
  - de longueur totale min → MINIMAL

Plan

Arbre couvrant de poids minimal

Algorithme de Kruskal

- Rappel : arbre $\rightarrow$ $m = n - 1$ arêtes
- **Entrée** : $G = (X, U, P)$ (n sommets, m arêtes)
- **Sortie** : A arbre couvrant de poids min de G

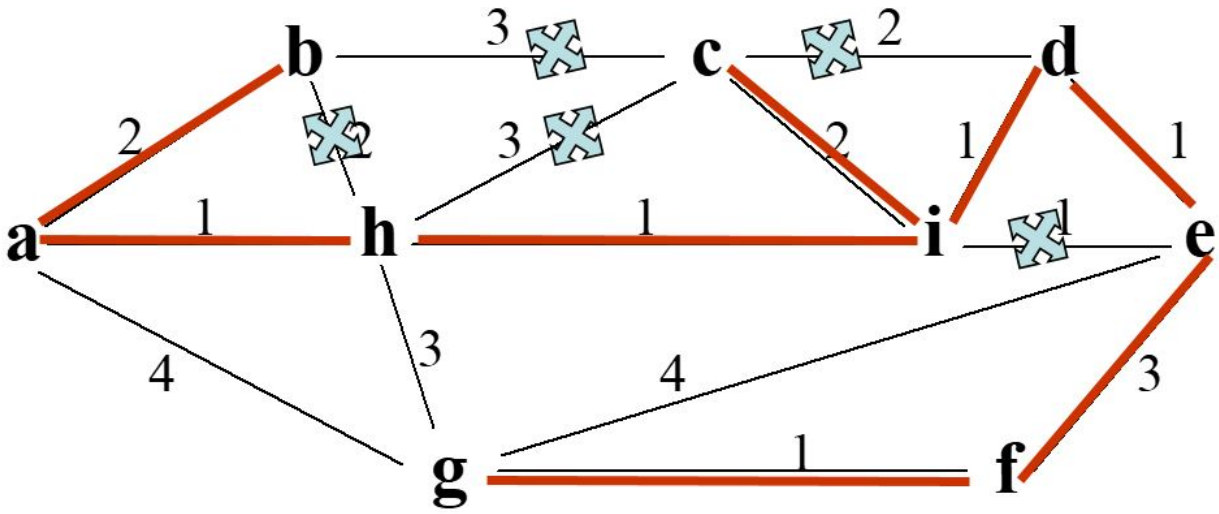
Arbre couvrant de poids minimal

Algorithme de Kruskal

```
algorithme kruskal (données : G, résultat : A : arbre)
début
    entier k ← 0;
    ensemble d'arêtes V ← ∅;
    trier les arêtes de G par ordre de poids croissant;
    tant que k < n − 1 faire
        parcourir la liste triée;
        sélectionner la première arête w qui ne forme
        pas de cycle avec les précédentes;
        k ← k + 1;
        V ← V ∪ {w};
    fait
    A ← graphe(X, V);
fin
```

Arbre couvrant de poids minimal

Exemple d'applciation de l'algorithme



- Liste triée :

ah	de	di	ei	hi	fg	ab	bh	ci	cd	bc	ch	ef	gh	ag	ge
1	1	1	1	1	1	2	2	2	2	3	3	3	3	4	4
x	x	x	-	x	x	x	-	x	-	-	-	x	STOP		

- $n = 9 \rightarrow$ stop après 8 sélections
- $P(A) = 1 + 1 + 1 + 1 + 1 + 2 + 2 + 3 = 12$

Arbre couvrant de poids minimal

Algorithme de Kruskal

- Implémentation peu facile (cf. exemple)
- Complexité : $O(m \log m)$ (tri)

Plan

Détection des cycles dans l'algorithme de Kruskal

La structure de données union-find

- Comment peut-on détecter qu'une arête sélectionnée forme un cycle avec les précédentes ?
- Pour ce faire, on utilise la notion de **connexité**.
- On peut considérer qu'un sous-ensemble d'arêtes choisies induit une **partition** de l'ensemble des sommets du graphe :
  - au départ : aucune arête n'est choisie : partition en singletons (on a n sous-ensembles, chaque sous-ensemble correspond à un sommet)
  - puis, à chaque fois qu'une arête est choisie, on fusionne les sous-ensembles auxquels appartiennent les deux extrémités de l'arête choisie.
- Au moment du choix d'une arête, on vérifie qu'elle relie des sommets appartenant à des ensembles différents, on est ainsi sûr que la nouvelle arête choisie ne formera pas de cycle
- Cf. Exemple

Plan

Arbre couvrant de poids minimal

Preuve de l'algorithme de Kruskal

- $G = (X, U)$ le graphe
- $A = (X, V)$ l'arbre couvrant obtenu par Kruskal
- $p(u)$: poids de l'arête u

Arbre couvrant de poids minimal

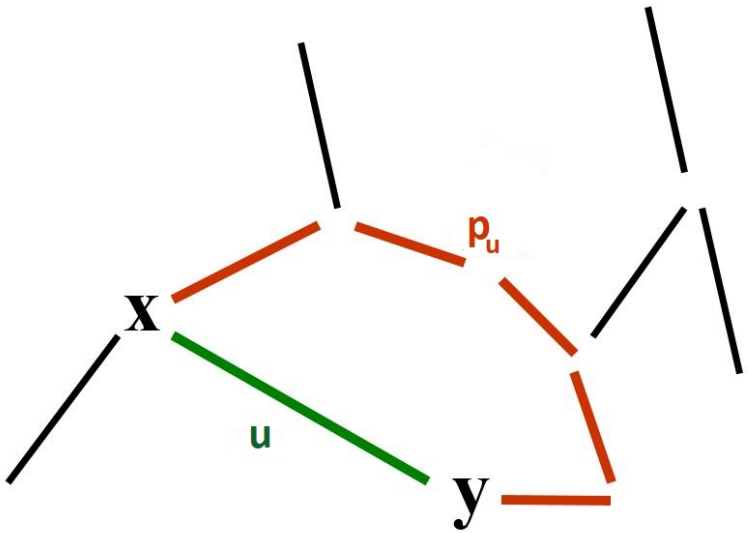
Preuve de l'algorithme de Kruskal

Property (1)

Soit :

- $w \in U - V, w = [x, y]$
- *et* c_w *: chaîne de* x *à* y *dans* A *;*

Alors, $p(w) \geq \max_{u \in c_w} p(u)$

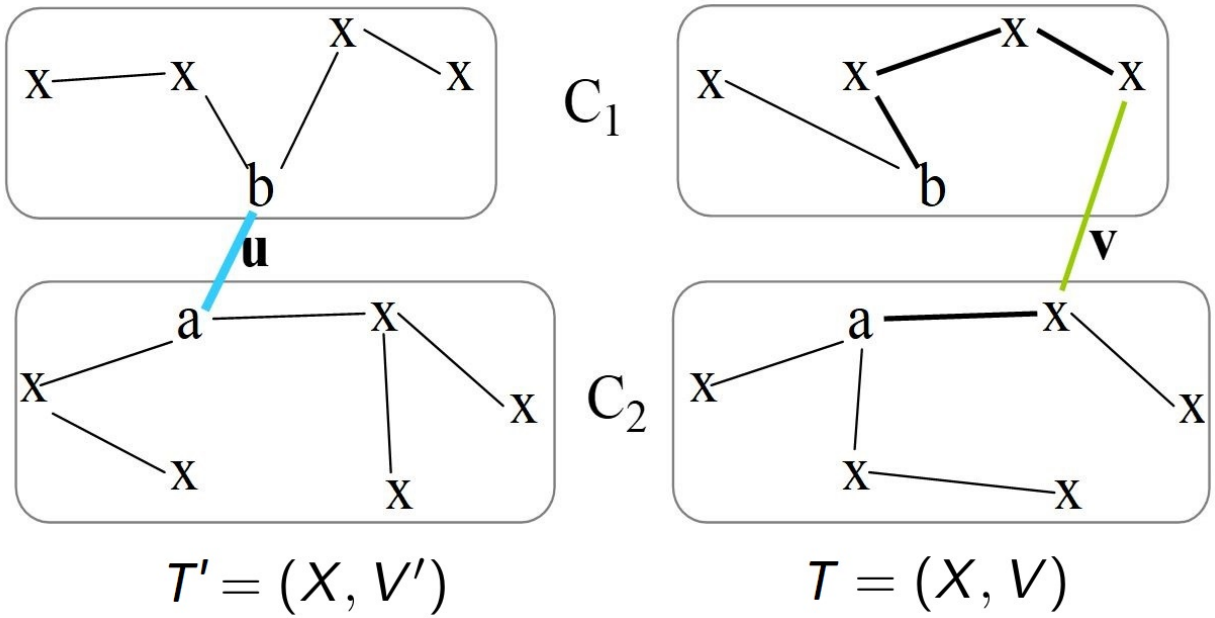


... car s'il existait une arête a de c_w de poids plus fort que $p(w)$, w aurait été sélectionnée avant a.

Arbre couvrant de poids minimal

Preuve de l'algorithme de Kruskal

- Soit A' un arbre optimal de poids $p(A')$
- Montrons que $p(A) = p(A')$
- Soit $u \in A' - A$



- $u \in V' - V$ relie C_1 et C_2 dans A'
- $v \in c_u$ relie C_1 et C_2 dans A

Arbre couvrant de poids minimal

Preuve de l'algorithme de Kruskal

- propriété (1) $\Rightarrow p(u) \geq p(v)$
- A' minimal $\Rightarrow p(v) \geq p(u)$
- $\Rightarrow p(u) = p(v)$

- Soit $A'' = A' + v - u$: $p(A'') = p(A')$
- $\rightarrow A''$ optimal et $v \in A''$

- Soit $u' \in A'' - A$, ... (idem... $\rightarrow A'''$
- ...
- fin : $A^{''\cdots''} = A$ et $p(A^{''\cdots''}) = p(A') = p(A)$ (fin preuve)

Plan

Union-find : Structures de données pour ensembles disjoints

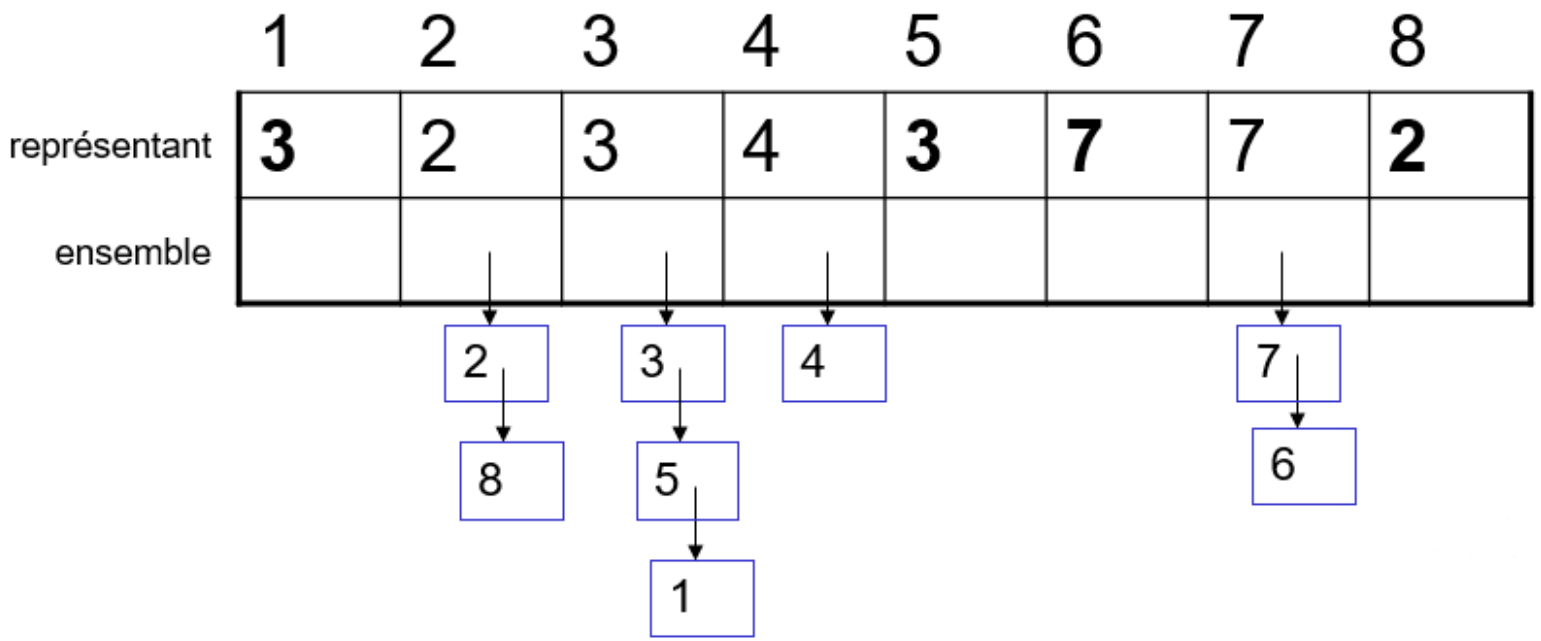
Principe

- *union-find* est une structure de données qui représente une partition d'un ensemble fini.
- Une telle structure peut se voir comme une collection $S = \{S_1, S_2, \ldots, S_k\}$ d'ensembles disjoints.
- Chaque ensemble est identifié par un ***représentant***, qui est un certain membre de l'ensemble.
- On souhaite disposer des opérations suivantes :
  - `makeSet(x)` : crée un nouvel ensemble dont le seul membre (et donc le représentant) est x. Comme les ensembles sont disjoints, il faut que x ne soit pas déjà membre d'un autre ensemble.
  - `union(x,y)` : réunit les ensembles qui contiennent x et y, disons S_x et S_y, dans un nouvel ensemble qui est l'union de ces deux ensembles. Les deux ensembles sont supposés être disjoints avant l'opération. Le représentant de l'ensemble résultant est un membre quelconque de $S_x \cup S_y$.
  - `find(x)` : retourne un pointeur vers le représentant de l'ensemble (unique) contenant x.

3 différentes implémentations d'une structure union-find

Implémentation 1 : listes chaînées simples

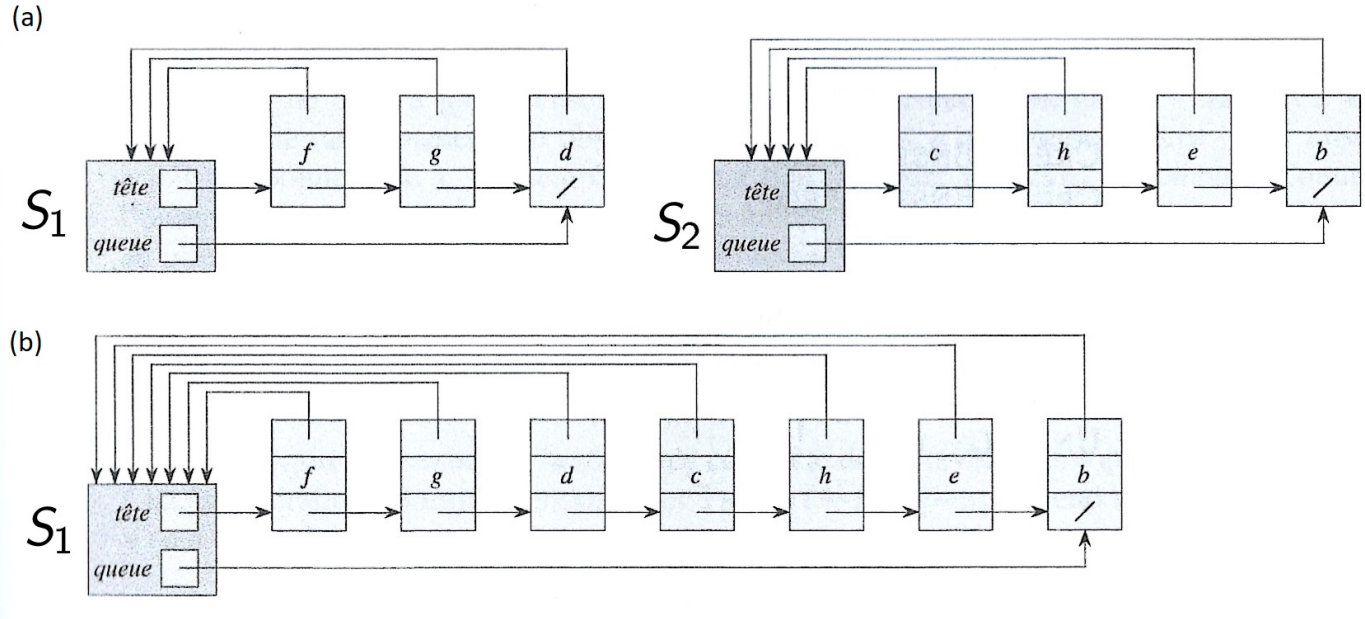
- Cf exemple précédent :



- Il s'agit là de la représentation des 4 ensembles disjoints $S_1 = \{2, 8\}$ (représentant : 2), $S_2 = \{3, 5, 1\}$ (représentant : 3), $S_3 = \{4\}$ (représentant : 4), $S_4 = \{7, 6\}$ (représentant : 7).

3 différentes implémentations d'une structure union-find

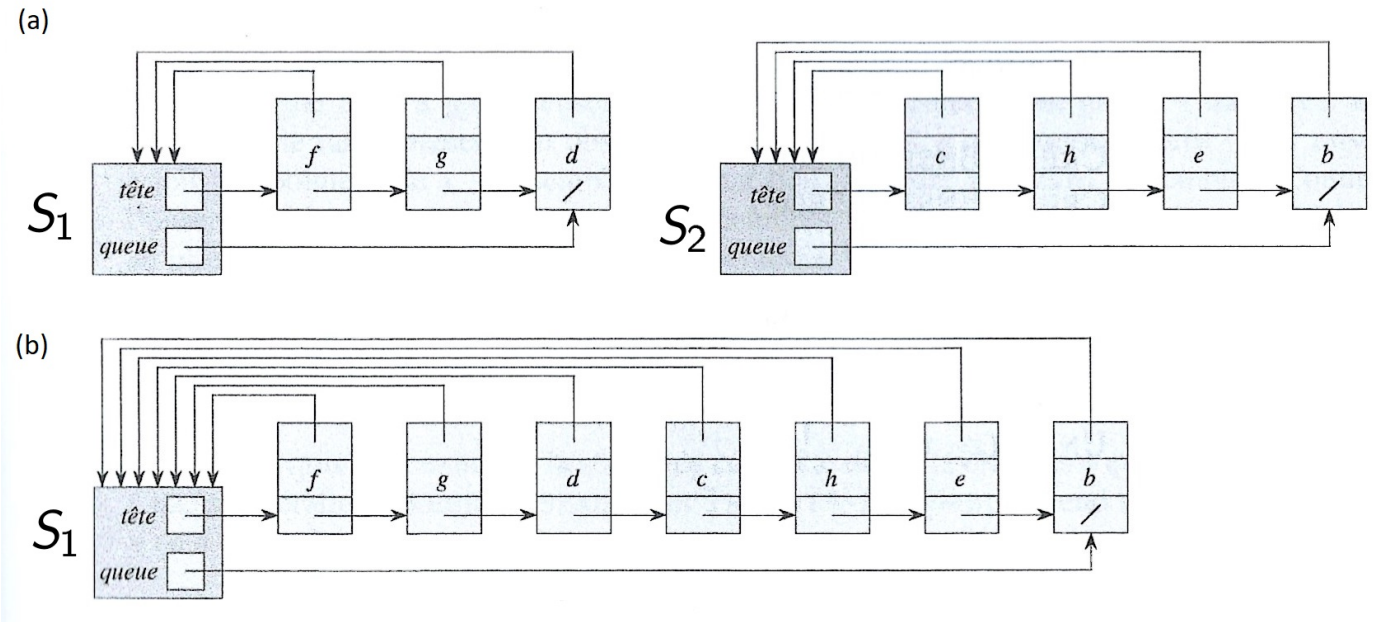
Implémentation 2 : listes chaînées avec tête et queue (1/2)



- (a) Représentation par listes chaînées avec tête et queue de deux ensembles : $S_1 = \{d, f, g\}$ (représentant : f) et $S_2 = \{c, h, e, b\}$ (représentant : c). Chaque noeud de la liste contient une valeur, un pointeur vers l'ensemble qui contient la valeur, un pointeur vers le noeud suivant dans la liste. Chaque objet associé à un ensemble a les pointeurs tête et queue pointant respectivement vers le premier et le dernier noeud.

3 différentes implémentations d'une structure union-find

Implémentation 2 : listes chaînées avec tête et queue (2/2)



- (b) Le résultat de `union(g,e)`, qui ajoute la liste chaînée contenant e à la liste chaînée contenant g. Le représentant de l'ensemble est f. L'objet ensemble pour la liste de e, S_2, est détruit (déréférencé en Java).
- Cette implémentation améliore la précédente : il conviendra de le vérifier en pratique et de l'expliquer. En particulier, il conviendra d'expliquer l'utilité pratique du pointeur de queue.

3 différentes implémentations d'une structure union-find

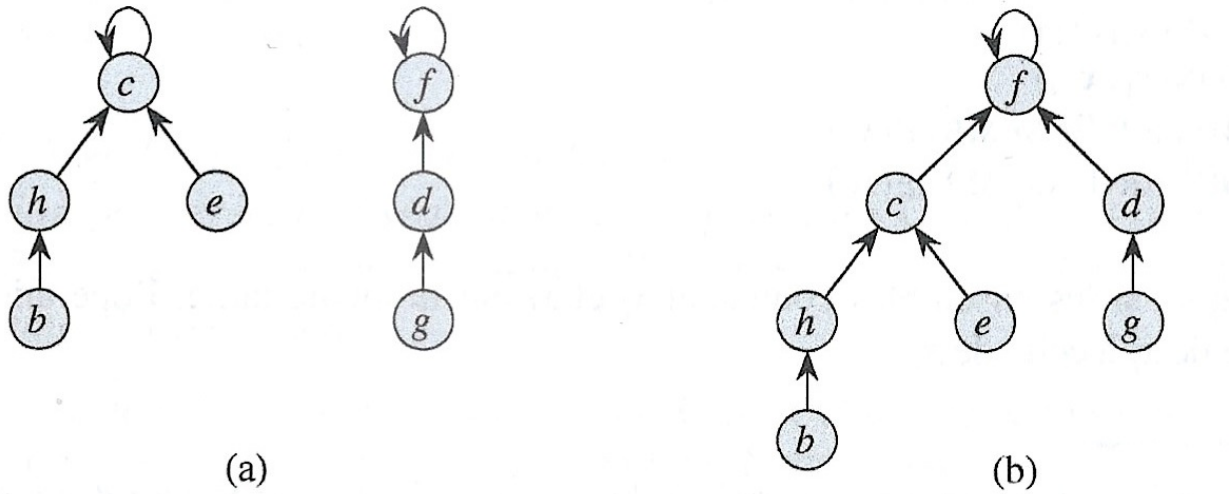
Implémentation 2 : l'union pondérée

- L'implémentation de la méthode `union` dans cette implémentation 2, dans le cas le plus défavorable, demande un temps moyen de $O(n)$ par appel. En effet, il se peut que l'on ajoute une liste longue à une liste courte, auquel cas il faut mettre à jour, pour chaque noeud de la liste la plus longue, le pointeur vers l'objet ensemble.
- Supposons maintenant que chaque liste contienne également la longueur de la liste (qui est facilement gérable) et que l'on concatène toujours la plus petite liste à la plus longue.
- Avec ce choix d'implémentation (appelée union pondérée simple) que vous mettrez en œuvre, la complexité algorithmique est améliorée.
- Là encore, il conviendra de mesurer ce gain sur les instances générées.

3 différentes implémentations d'une structure union-find

Implémentation 3 : Forêts d'ensemble disjoints

- Pour une implémentation encore plus rapide, on représente les ensembles par des arbres.
- Dans une forêt d'ensembles disjoints, chaque élément pointe vers son parent. La racine de chaque arbre contient le représentant et elle est son propre parent.



- (a) Deux arbres représentant les deux ensembles S_1 et S_2 de l'implémentation 2. L'arbre de gauche représente l'ensemble $\{b, c, e, h\}$ (avec c comme représentant) et l'arbre de droite représente l'ensemble $\{d, f, g\}$ (avec f comme représentant).
- (b) Le résultat de `union(e,g)`.

3 différentes implémentations d'une structure union-find

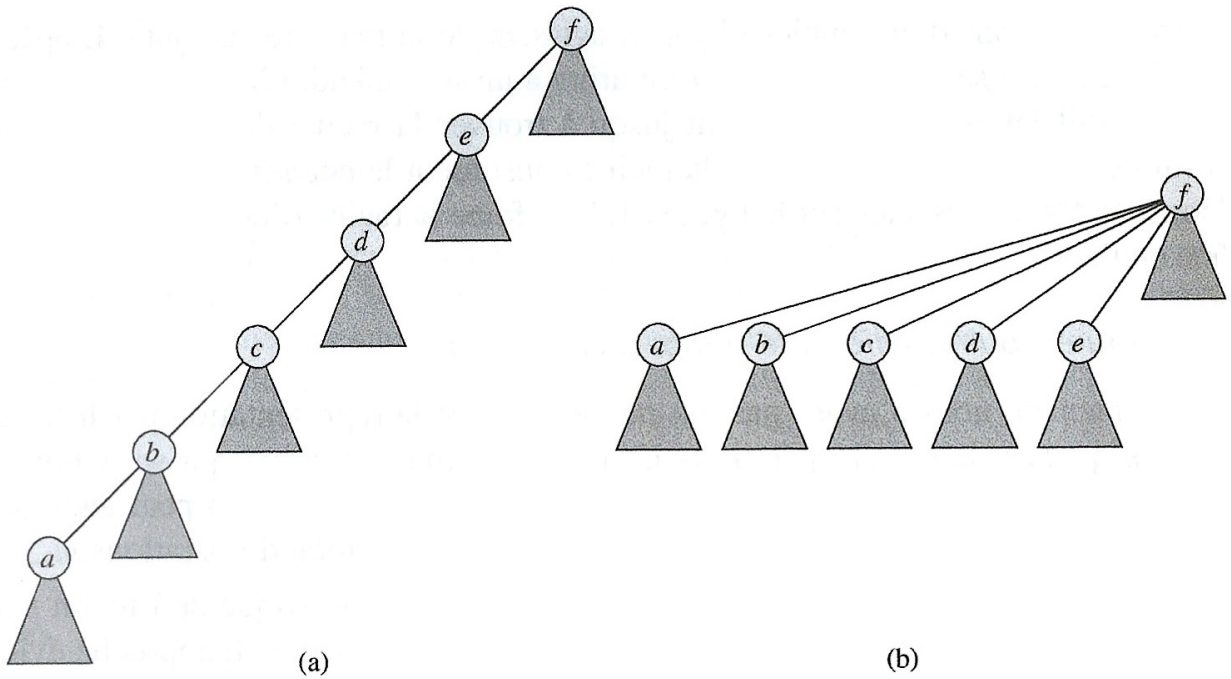
Implémentation 3 : Forêts d'ensemble disjoints

- Les trois opérations classiques agissent ainsi :
  - `makeSet(x)` se contente de créer un arbre à un seul noeud
  - `find(x)` suit les pointeurs de parent jusqu'à trouver la racine de l'arbre. les noeuds vidités sur ce chemin menant à la racine constituent le *chemin de découverte*.
  - L'opération `union` force la racine d'un arbre à pointer vers la racine de l'autre.
- Afin d'accélérer les traitements (et faire mieux que les listes chaînées), en cas d'union, on fera pointer l'arbre de hauteur minimale vers la racine de l'arbre de hauteur maximale. Pour chaque noeud, on gérera donc un ***rang*** qui correspondra à un majorant de la hauteur du noeud. Dans la méthode `union`, on fait pointer la racine de moindre rang vers celle de rang supérieur. (On parle d'union par rang.)

3 différentes implémentations d'une structure union-find

Implémentation 3 : Forêts d'ensemble disjoints

- Toujours afin d'accélerer les traitements, on utilise la *compression de chemin*, qui est très simple et très efficace. Comme le montre la figure, on s'en sert pendant les opérations `find` pour faire pointer chaque noeud du chemin de découverte directement vers la racine. La compression de chemin ne modifie aucun rang.



3 différentes implémentations d'une structure union-find

Implémentation 3 : Forêts d'ensemble disjoints

- Dans la figure précédente, les flèches et les boucles aux racines sont omises.
- (a) est un arbre représentant un ensemble avant l'exécution de `find(a)`. Mes triangles représentent des sous arbres dont les racines sont les noeuds montrés sur la figure. Chaque noeud possède un pointeur vers son parent.
- (b) Le même ensemble après exécution de `find(a)`. À présent, chaque noeud du chemin de découverte pointe directement vers la racine.

3 différentes implémentations d'une structure union-find

Implémentation 3 : pseudo-code des opérations

```
méthode makeSet(x)
        x.parent = x
        x.rang = 0


méthode union(x,y)
        link(find(x), find(y))


méthode link(x,y)
        si (x.rang > y.rang) alors y.parent = x sinon x.parent = y fin si
        si (x.rang == y.rang) alors y.rang = y.rang + 1 fin si


méthode find(x)
        si (x ≠ x.parent) alors
                x.parent = find(x.parent)
        fin si
        retourner x.parent
```

3 différentes implémentations d'une structure union-find

Implémentation 3 : explication de la méthode `find`

- la méthode `find(x)` est une procédure récursive à deux passes : quand elle lance un appel récursif, elle effectue une passe qui remonte le chemin de découverte pour trouver la racine ;
- et quand la récursivité revient en arrière (backtrack), elle effectue une seconde passe qui redescend le chemin de découverte pour mettre à jour chaque noeud de façon qu'il pointe directement vers la racine.
- Chaque appel de la méthode retourne `x.parent`. Si `x` est la racine, alors la méthode retourne `x.parent` qui est `x` : c'est le cas où la récursivité prend fin.
- Sinon, il y a exécution de la partie alors et l'appel récursif sur le paramètre x.parent retourne un pointeur vers la racine. Le noeud x est mis à jour pour le faire pointer directement vers la racine.

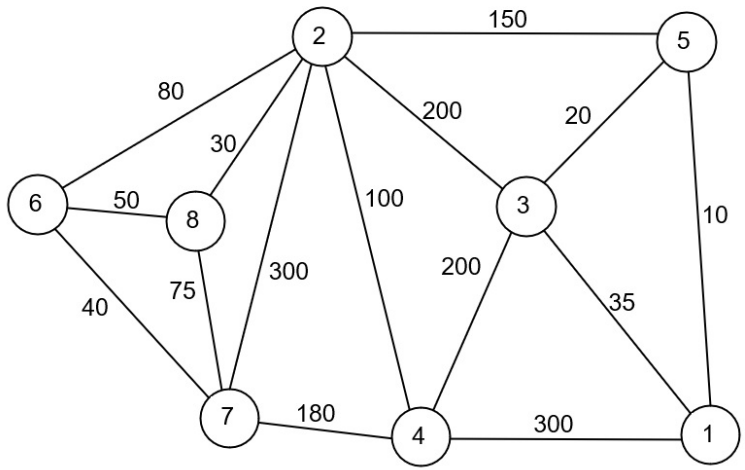
Plan

Entrées/Sorties de votre programme (1/2)

- Votre programme, en plus de générer les 6 instances mentionnées précédemment, devra permettre de lire un graphe contenu dans un fichier texte nommé graphe.txt, qui contiendra les poids des arêtes, ligne par ligne, en utilisant un espace comme séparateur :

```
0 0 35 300 10 0 0 0
0 0 200 100 150 80 300 30
35 200 0 200 20 0 0 0
300 100 200 0 0 0 180 0
10 150 20 0 0 0 0 0
0 80 0 0 0 0 40 50
0 300 0 180 0 40 0 75
0 30 0 0 0 50 75 0
```

fichier graphe.txt correspondant au graphe ci-contre



Entrées/Sorties de votre programme (2/2)

- Votre programme devra pouvoir afficher, pour une instance donnée, la solution trouvée en affichant à l'écran :
  - le poids de l'arbre couvrant trouvé
  - la liste des arêtes sélectionnées

Plan

Utilitaires

- Vous trouverez dans le sharepoint une classe `LireLigne.java` qui contient les indications pour :
  - générer un entier aléatoire selon une loi uniforme entre deux valeurs min et max (comprises)
  - lire un fichier texte ligne par ligne
  - "splitter" (éclater) la ligne en ses différents éléments séparés par des espaces