



## Chapitre 9

---

### L'approche gloutonne (vorace)



# Le problème de rendre la monnaie

---

- Étant donné un montant  $n$  et une quantité suffisante de pièces pour chacune des dénominations  $d_1, d_2, \dots, d_m$ , trouvez le plus petit nombre de pièces dont la valeur totale donne  $n$ .
  - Exemple: au Canada nous avons:
    - $d_1 = 25$  cents
    - $d_2 = 10$  cents
    - $d_3 = 5$  cents
    - $d_4 = 1$  cents
  - Pour rendre la monnaie d'un montant  $n$ , les caissiers utilisent un algorithme glouton (vorace):
    - On utilise les pièces de la plus grande dénomination possible tant que leur valeur totale n'excède pas  $n$
    - Ex: lorsque  $n = 32$ . Nous utilisons  $1 \times 25 + 0 \times 10 + 1 \times 5 + 2 \times 1$  (donc 4 pièces).



## Le problème de rendre la monnaie (suite)

- En fait, l'algorithme glouton trouve la solution optimale pour certaines dénominations (comme celle utilisée au Canada).
- Cependant, il existe des dénominations où la solution trouvée par l'algorithme glouton n'est pas optimale.
  - Ex: supposons que nous utilisons une dénomination sans pièces de 5 cents:  $d_1 = 25$ ,  $d_2 = 10$  et  $d_3 = 1$ .
  - Pour  $n = 32$ , l'algorithme glouton trouve:  $1 \times 25 + 7 \times 1$  (8 pièces)
  - Or, la solution optimale est:  $3 \times 10 + 2 \times 1$  (5 pièces)
- Il existe un algorithme de **programmation dynamique** (problèmes série 7) qui trouve toujours la solution optimale en un temps  $\Theta(mn)$ .
- Or le temps requis par **l'algorithme glouton** est  $\in O(n)$  car, en pire cas, l'algorithme utilisera uniquement les pièces de dénomination  $d_1 = 1$ .
- Un algorithme glouton est souvent l'algorithme de choix lorsqu'il arrive à trouver la solution optimale ou lorsque l'on est satisfait de sa solution



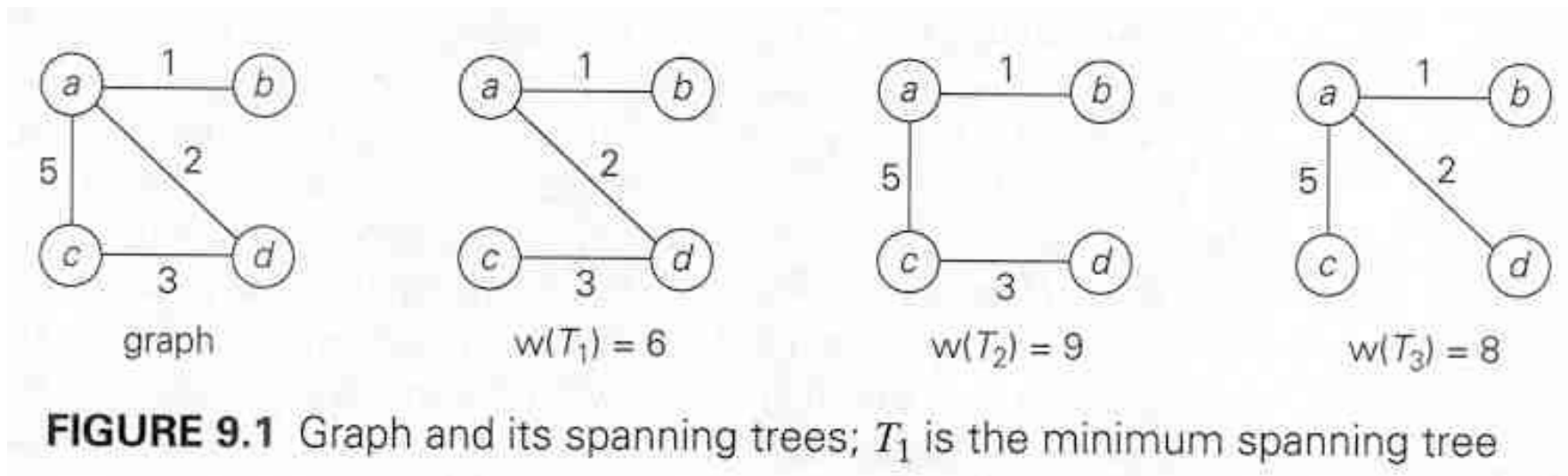
# Caractéristiques des algorithmes gloutons

---

- Un algorithme glouton construit une solution en effectuant une séquence de décisions. Pour chaque décision:
  - Le choix effectué **satisfait les contraintes** du problème
    - Ex: chaque pièce que l'on ajoute (aux pièces déjà choisies) nous donne un montant total  $\leq n$ .
  - Le choix effectué est **localement optimal**
    - Ex: on choisi la pièce de la plus grande dénomination possible nous donnant un montant total (avec les autres pièces)  $\leq n$
  - Le choix effectué est **irrévocable**
    - Ex: La pièce choisie ne pourra pas être retirée ultérieurement
- Ainsi, chacune des décisions est gloutonne (ou vorace)
- C'est une bonne stratégie lorsque la séquence de décisions, localement optimales, donne une solution globalement optimale ou lorsque l'on est satisfait de la « non optimalité » de la solution globale

# Arbres de recouvrement minimaux

- Considérons les **graphes connexes et non orientés** où chaque arête possède un poids (ou une distance)
- Un **arbre de recouvrement** d'un graphe connexe et non orienté est un arbre (i.e. un graphe acyclique) contenant tous les nœuds du graphe
- Un **arbre de recouvrement minimal** est un arbre de recouvrement dont le poids (la somme des poids de chacune de ses arêtes) est minimal
- Nous allons étudier 2 algorithmes gloutons différents permettant de trouver un arbre de recouvrement minimal





# L'algorithme de Prim

---

- Cet algorithme débute avec un arbre constitué d'un seul nœud (choisi arbitrairement)
- Un arbre de recouvrement minimal est construit en ajoutant un nœud à la fois à cet arbre
- À chaque étape, le nœud est choisi de manière gloutonne
  - C'est le nœud, n'appartenant pas à l'arbre, qui est connecté à un nœud de l'arbre par l'arête de poids minimal
- L'algorithme termine lorsque l'arbre contient les  $n$  nœuds du graphe
  - L'algorithme effectue donc  $n - 1$  choix gloutons
- L'algorithme retourne l'ensemble des arêtes constituant cet arbre de recouvrement
- Nous allons démontrer, sous peu, que cet arbre est un arbre de recouvrement minimal



## Pseudo code de l'algorithme de Prim

### **ALGORITHM** *Prim*( $G$ )

//Prim's algorithm for constructing a minimum spanning tree

//Input: A weighted connected graph  $G = \langle V, E \rangle$

//Output:  $E_T$ , the set of edges composing a minimum spanning tree of  $G$

$V_T \leftarrow \{v_0\}$  //the set of tree vertices can be initialized with any vertex

$E_T \leftarrow \emptyset$

**for**  $i \leftarrow 1$  **to**  $|V| - 1$  **do**

    find a minimum-weight edge  $e^* = (v^*, u^*)$  among all the edges  $(v, u)$   
    such that  $v$  is in  $V_T$  and  $u$  is in  $V - V_T$

$V_T \leftarrow V_T \cup \{u^*\}$

$E_T \leftarrow E_T \cup \{e^*\}$

**return**  $E_T$

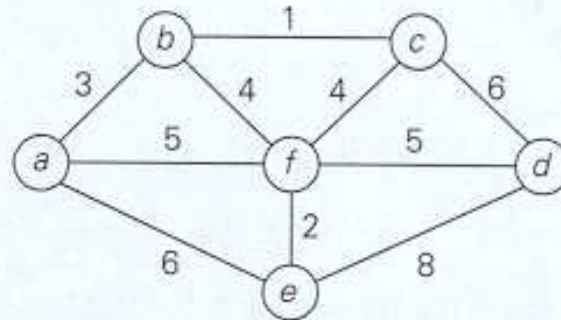


## Détail de chaque étape gloutonne de Prim

- À chaque étape gloutonne, nous devons déterminer, pour chaque nœud  $u$  à l'extérieur de l'arbre  $T$  (i.e.  $\forall u \in V - V_T$ ):
  - Le poids  $w_u$  de l'arête de poids minimal reliant  $u$  à l'arbre  $T$ 
    - (le poids est  $\infty$  lorsque  $u$  n'est pas relié à l'arbre par une arête)
  - et le nœud  $v_u \in V_T$  relié à  $u$  par cette arête de poids  $w_u$
- Pour cela, nous attachons (et maintenons) les 2 **étiquettes**  $w_u$  et  $v_u$  à chaque nœud  $u \in V - V_T$
- Après avoir identifié l'arête  $(v^*, u^*) : v^* \in V_T$  et  $u^* \in V - V_T$  et  $(v^*, u^*)$  est de poids minimal il faut:
  - Déplacer  $u^*$  de  $V - V_T$  vers  $V_T$
  - Pour chaque  $u \in V - V_T$  connecté à  $u^*$  par une arête de poids  $w$  inférieur à  $w_u$ , il faut faire les mises à jour des 2 étiquettes:
    - $w_u \leftarrow w$
    - $v_u \leftarrow u^*$



# Application de l'algorithme de Prim



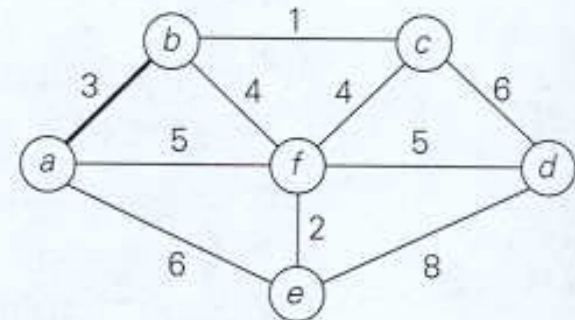
**Tree vertices**

**Remaining vertices<sup>a</sup>**

**Illustration**

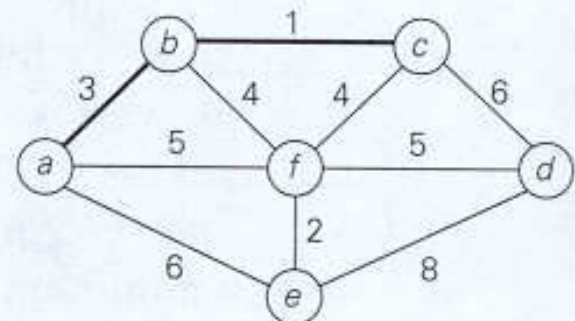
$a(-, -)$

$b(a, 3)$   $c(-, \infty)$   $d(-, \infty)$   
 $e(a, 6)$   $f(a, 5)$



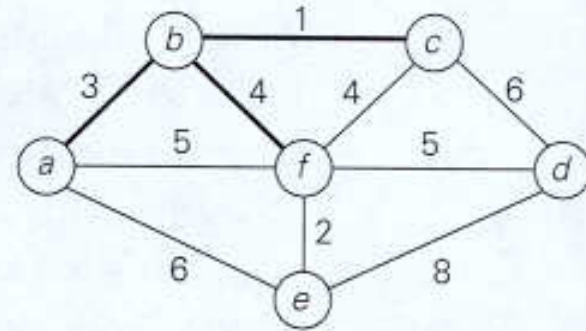
$b(a, 3)$

$c(b, 1)$   $d(-, \infty)$   $e(a, 6)$   
 $f(b, 4)$

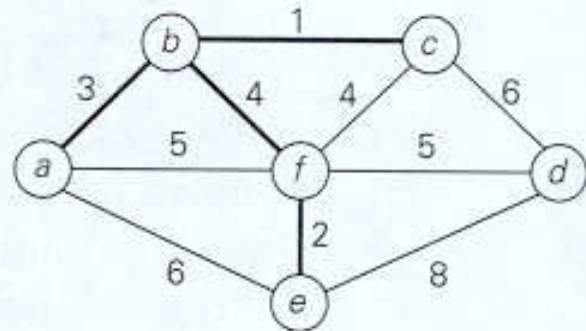


# Application de l'algorithme de Prim (suite)

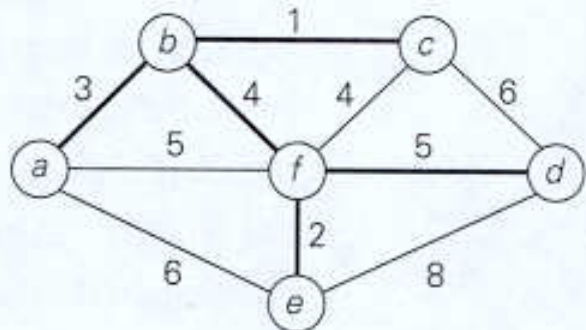
c(b, 1)      d(c, 6) e(a, 6) **f(b, 4)**



f(b, 4)      d(f, 5) **e(f, 2)**



e(f, 2)      **d(f, 5)**



d(f, 5)



## Exactitude de l'algorithme de Prim

---

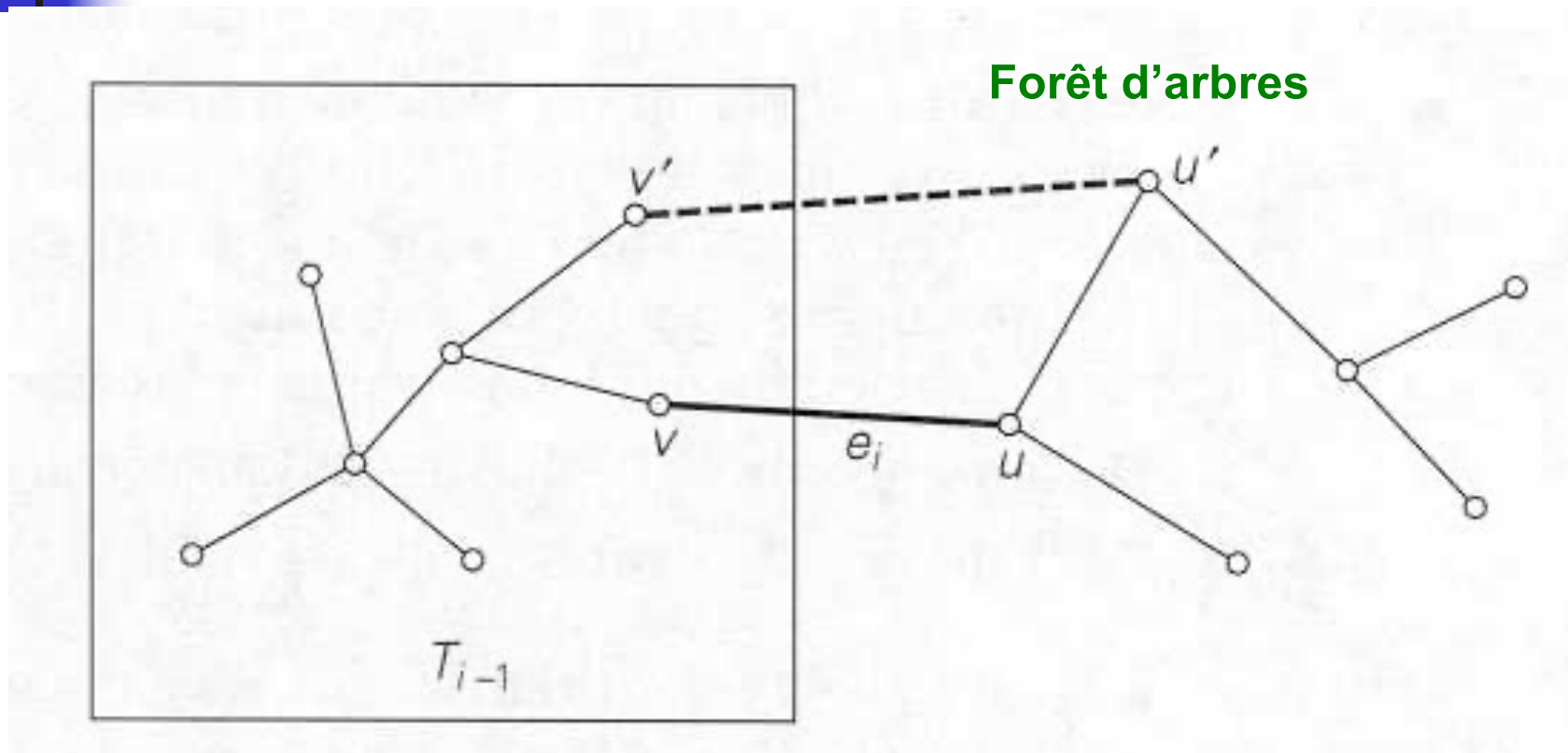
- Démontrons que l'arbre de recouvrement fourni par l'algorithme de Prim est un arbre de recouvrement minimal.
- Soit  $T_0$  l'arbre initial contenant un seul nœud
- Soit  $T_i$  l'arbre à la fin de la  $i$ -ième étape gloutonne de l'algorithme de Prim (contenant  $i + 1$  nœuds)
- $T_{n-1}$  est alors l'arbre final fourni par l'algorithme de Prim
- Démontrons, par induction, que chaque arbre  $T_i$  est un sous arbre d'un arbre de recouvrement minimal
  - Si c'est vrai, alors  $T_{n-1}$  est un arbre de recouvrement minimal
- $T_0$ , étant constitué d'un seul nœud, est forcément un sous arbre d'un arbre de recouvrement minimal
- Démontrons que pour tout  $i$  : si  $T_{i-1}$  est un sous arbre d'un arbre de recouvrement minimal, alors il en sera ainsi pour  $T_i$



## Exactitude de l'algorithme de Prim (suite)

- Prouvons ce dernier énoncé par contradiction.
- Supposons que  $T_i$  ne soit pas un sous arbre d'un arbre de recouvrement minimal
- Par contre,  $T_{i-1}$  est, par hypothèse, un sous arbre d'un arbre de recouvrement minimal  $T$
- Soit  $e_i = (v,u)$  = l'arête de poids minimal utilisé par l'algorithme de Prim pour passer de  $T_{i-1}$  à  $T_i$
- Par hypothèse,  $e_i$  ne peut pas être inclus dans  $T$  car sinon  $T_i$  serait un sous arbre d'un arbre de recouvrement minimal.
- L'ajout de  $e_i$  à  $T$  forme forcément un cycle contenant  $e_i$  ainsi qu'une autre arête  $(v',u')$  telle que  $v' \in T_{i-1}$  et  $u' \notin T_{i-1}$  (voir figure page suivante)
- (il est possible que  $v'$  coïncide avec  $v$  ou que  $u'$  coïncide avec  $u$  mais non les deux à la fois)

## Exactitude de l'algorithme de Prim (suite)



- En enlevant  $(v', u')$  nous obtenons alors un autre arbre de recouvrement  $T' \neq T$  qui inclut  $e_i$  et dont le poids total est inférieur ou égal à celui de  $T$  car le poids de  $e_i$  est inférieur ou égal à celui de  $(v', u')$ .
- Donc  $T'$  est forcément un arbre de recouvrement minimal.
- Alors  $T_i$  est un sous arbre d'un arbre de recouvrement minimal. **CQFD**

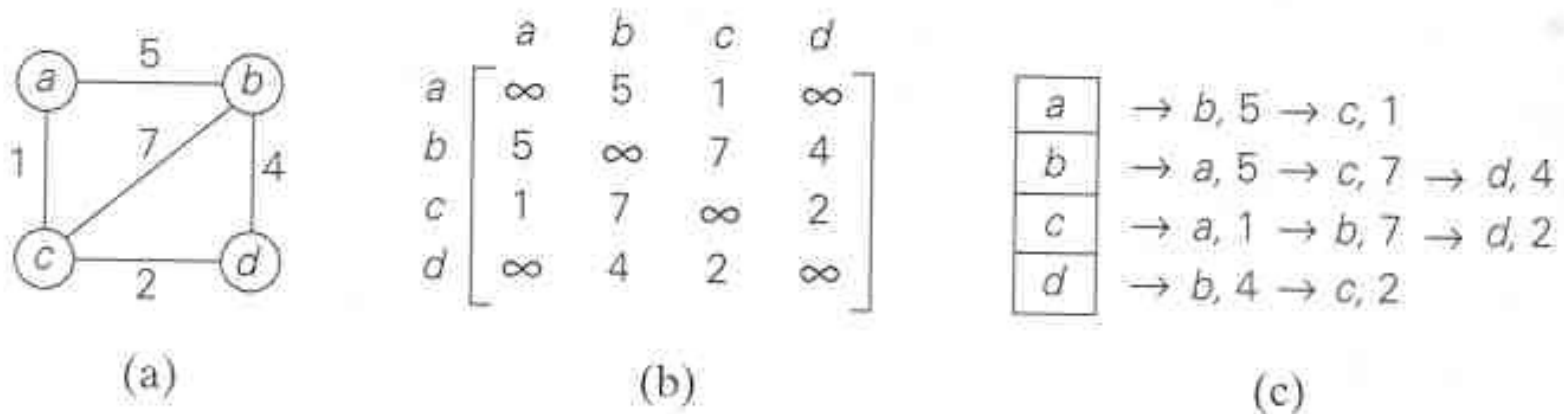


## Analyse de l'efficacité de l'algorithme de Prim

- Le temps d'exécution de l'algorithme de Prim dépend de la structure de données utilisée pour le graphe et de la structure de données utilisée pour la file d'attente des nœuds  $\in V - V_T$
- Utilisons un **tas-min** pour la file d'attente des nœuds  $\in V - V_T$ 
  - Un **tas-min** est un arbre binaire essentiellement complet dont la valeur de chaque nœud est **inférieure ou égale** à celle de ses enfants
    - Les tas que nous avons étudiés au chapitre 6 sont, en fait, des **tas-max**
  - La valeur de chaque nœud  $u$  dans la file d'attente est la valeur  $w_u$  du poids de l'arête de poids minimal reliant  $u$  à l'arbre
  - Le nœud  $u^* \in V - V_T$  ayant la plus faible valeur  $w_{u^*}$  sera donc toujours au sommet du tas-min et cela prendra un temps  $\in O(\log(|V - V_T|))$  pour reconstruire le tas-min après avoir enlevé  $u^*$ 
    - Cela aurait pris un temps  $\in O(|V - V_T|)$  si, au lieu d'un tas-min, nous utilisions un simple tableau (non-trié) pour cette file.

## Analyse de l'efficacité de l'algorithme de Prim (suite)

- Lorsque l'on ajoute un nœud  $u^*$  à l'arbre, il faut, pour chaque  $u \in V - V_T$  connecté à  $u^*$ , mettre possiblement à jour  $w_u$  et  $v_u$  et reconstruire le tas-min pour chaque  $u$  mis à jour
  - Si  $E_{u^*}$  désigne l'ensemble des nœuds connectés à  $u^*$ , cela nécessite un temps  $\in O(|E_{u^*}| \times \log(|V - V_T|)) \subseteq O(|E_{u^*}| \times \log(|V|))$  lorsque le graphe est représenté par des listes d'adjacences
    - Ça serait  $O(|V| \times \log(|V|))$  pour un graphe représenté par une matrice d'adjacence (nettement moins bon)



**FIGURE 1.8** (a) Weighted graph. (b) Its adjacency matrix. (c) Its adjacency linked lists.





## Analyse de l'efficacité de l'algorithme de Prim (suite)

- Donc pour chaque nœud  $u^*$  que l'on insère dans l'arbre cela coûte un temps  $O(\log|V|)$  pour enlever  $u^*$  du tas-min et cela coûte un temps  $O(|E_{u^*}| \times \log(|V|))$  pour les mis à jour des  $u \in V - V_T$  connectés à  $u^*$ 
  - La somme des temps requis pour les mis à jours effectuées pour tous les nœuds  $u^*$  insérés dans l'arbre sera de  $O(|E| \times \log(|V|))$  pour un graphe de  $|E|$  arêtes
  - La somme des temps requis pour enlever chacun des  $|V| - 1$  nœuds  $u^*$  du tas min sera de  $O((|V|-1) \times \log(|V|))$
- Le temps d'exécution total sera donc en  $O((|V|-1 + |E|) \times \log(|V|))$  et donc en  **$O(|E| \times \log(|V|))$**  puisque  $|E| \geq |V|-1$  pour un graphe connexe
- Examinons maintenant un autre algorithme glouton nous permettant, également, de trouver un arbre de recouvrement minimal.



# Pseudo-code de l'algorithme de Prim avec monceau

---

**Procédure** MST-Prim( $G, w, r$ )

---

**pour**  $u \in G.\text{noeuds}$  **faire**     $u.\text{distance} \leftarrow \infty$ ;     $u.\pi \leftarrow \text{nil}$ ;     $u.\text{dansQ} \leftarrow \text{vrai}$ ; $r.\text{distance} \leftarrow 0$ ; $Q \leftarrow G.\text{noeuds}$ ;

// Crée un monceau inverse en utilisant l'attribut distance pour comparer les éléments;

HeapBottomUp( $Q$ );**tant que**  $Q \neq \emptyset$  **faire**     $u \leftarrow \text{ExtraisLaRacine}(Q)$ ;     $u.\text{dansQ} \leftarrow \text{faux}$ ;    **pour**  $v$  adjacent à  $u$  dans  $G$  **faire**        **si**  $v.\text{dansQ} \wedge w(u, v) < v.\text{distance}$  **alors**             $v.\pi \leftarrow u$ ;             $v.\text{distance} \leftarrow w(u, v)$  // Percoler  $v$  dans le monceau  $Q$ ; $E_T \leftarrow \emptyset$ ;**pour**  $v \in G.\text{noeuds} \setminus \{r\}$  **faire**     $E_T \leftarrow E_T \cup \{(v, v.\pi)\}$ ;**retourner**  $E_T$ ;



# L'algorithme de Kruskal

---

- L'algorithme trie d'abord, en ordre croissant de leur poids, l'ensemble  $E$  des arêtes d'un graphe  $G = \langle V, E \rangle$  connexe
- Ensuite, en débutant avec  $E_T = \emptyset$ , l'algorithme ajoute à  $E_T$  l'arête  $e \in E$  de poids  $w_e$  minimal qui ne forme pas de cycle avec les arêtes déjà dans  $E_T$ 
  - Cette arête n'est pas nécessairement connectée à une autre arête de  $E_T$
  - L'ensemble des arêtes de  $E_T$  constitue alors une forêt (d'arbres)
    - C'est donc un graphe généralement non connexe
- Cette séquence de choix gloutons se termine lorsque  $|E_T| = |V| - 1$ 
  - $E_T$  devient alors un arbre (unique) qui recouvre  $G$
  - Nous démontrerons que c'est nécessairement un arbre de recouvrement minimal



## Pseudo code de l'algorithme de Kruskal

### ALGORITHM *Kruskal*( $G$ )

//Kruskal's algorithm for constructing a minimum spanning tree

//Input: A weighted connected graph  $G = \langle V, E \rangle$

//Output:  $E_T$ , the set of edges composing a minimum spanning tree of  $G$

Sort  $E$  in nondecreasing order of the edge weights  $w(e_{i_1}) \leq \dots \leq w(e_{i_{|E|}})$

$E_T \leftarrow \emptyset$ ;  $ecounter \leftarrow 0$  //initialize the set of tree edges and its size

$k \leftarrow 0$  //initialize the number of processed edges

**while**  $ecounter < |V| - 1$

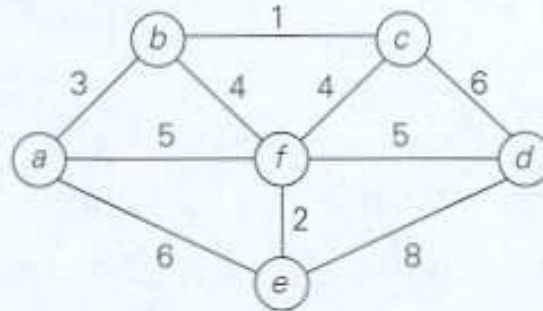
$k \leftarrow k + 1$

**if**  $E_T \cup \{e_{i_k}\}$  is acyclic

$E_T \leftarrow E_T \cup \{e_{i_k}\}$ ;  $ecounter \leftarrow ecounter + 1$

**return**  $E_T$

# Application de l'algorithme de Kruskal

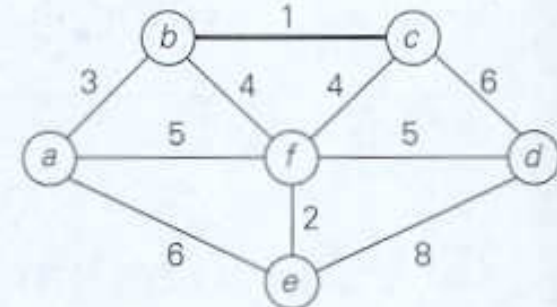


Tree edges

Sorted list of edges<sup>a</sup>

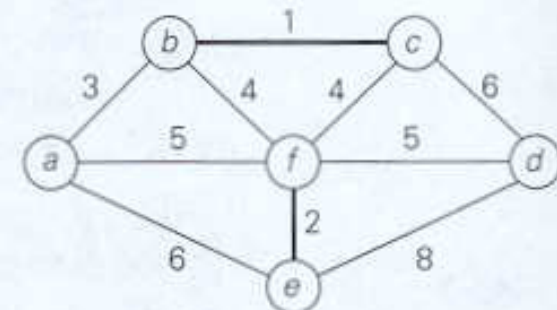
Illustration

**bc** 1   **ef** 2   **ab** 3   **bf** 4   **cf** 4   **af** 5   **df** 5   **ae** 6   **cd** 6   **de** 8



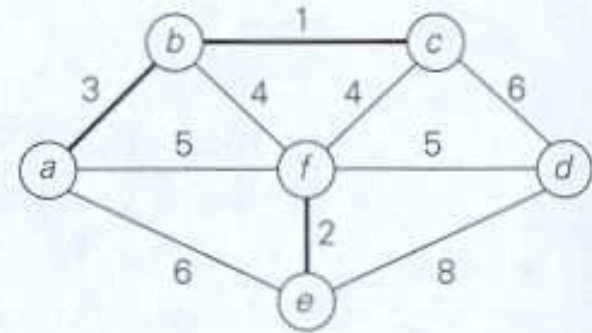
**bc**  
1

**bc** 1   **ef** 2   **ab** 3   **bf** 4   **cf** 4   **af** 5   **df** 5   **ae** 6   **cd** 6   **de** 8

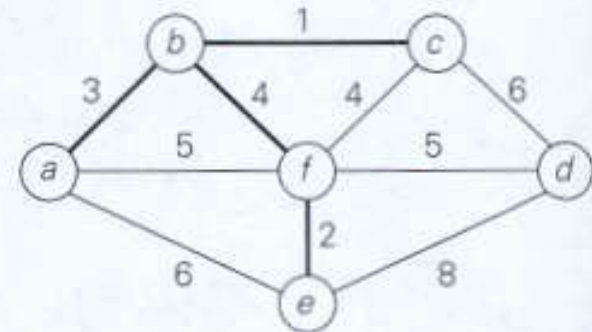


# Application de l'algorithme de Kruskal (suite)

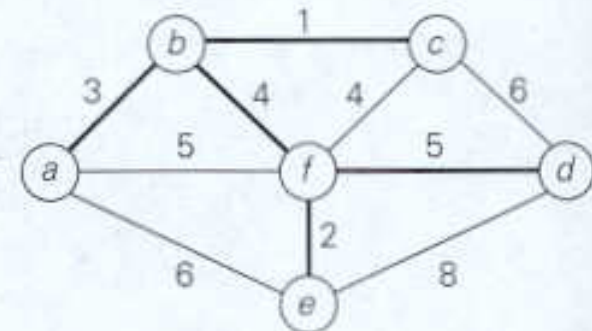
ef 2      bc 1   ef 2   **ab** 3   bf 4   cf 4   af 5   df 5   ae 6   cd 6   de 8



ab 3      bc 1   ef 2   ab 3   **bf** 4   cf 4   af 5   df 5   ae 6   cd 6   de 8



bf 4      bc 1   ef 2   ab 3   bf 4   cf 4   af 5   **df** 5   ae 6   cd 6   de 8



df 5



## Exactitude de l'algorithme de Kruskal

---

- La démonstration est presque identique à celle démontrant l'exactitude de l'algorithme de Prim
- Chaque étape gloutonne construit une forêt  $F_i$  à ajoutant une arête de poids minimal à la forêt précédente  $F_{i-1}$ 
  - La forêt initiale  $F_0$  est constituée de  $|V|$  arbres triviaux : chacun étant constitué d'un seul noeud
- Démontrons, par induction, que chaque forêt  $F_i$  est un sous graphe d'un arbre de recouvrement minimal
  - Cela impliquera alors que la forêt finale est un arbre de recouvrement minimal
- $F_0$  est trivialement un sous graphe d'un arbre de recouvrement minimal
- Supposons que  $F_{i-1}$  soit un sous graphe d'un arbre  $T$  de recouvrement minimal
- Démontrons, par contradiction, que  $F_i$  est nécessairement un sous graphe d'un arbre de recouvrement minimal



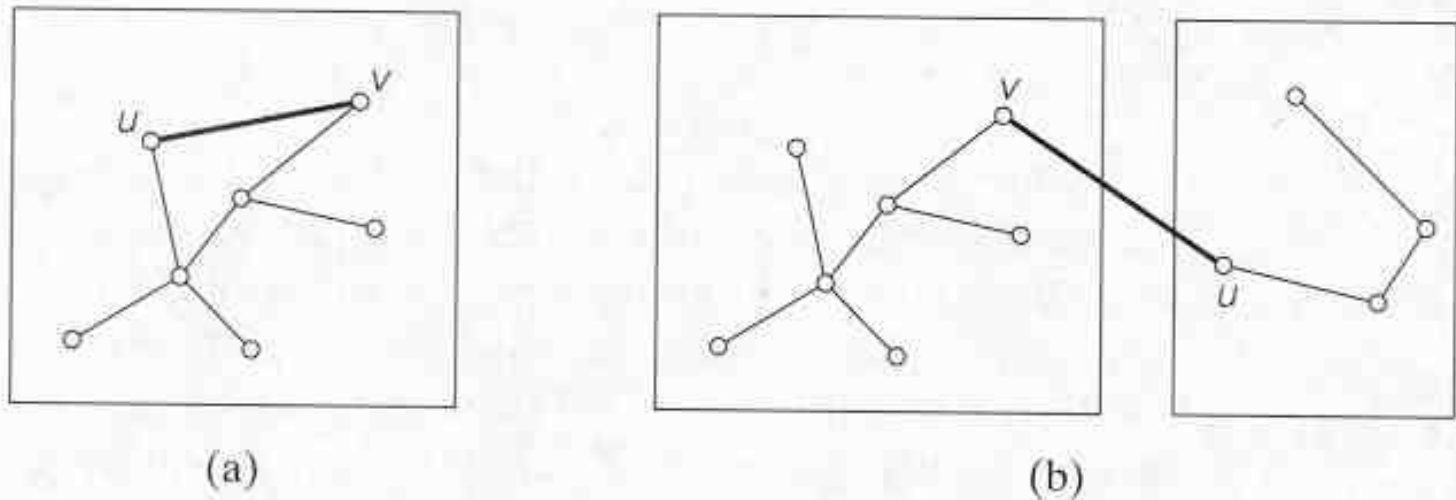
## Exactitude de l'algorithme de Kruskal (suite)

- Alors si  $F_{i-1}$  est un sous graphe de  $T$  et que  $F_i$  n'en est pas un, l'arête  $e_i = (v, u)$  choisie à l'étape  $i$  ne doit pas être incluse dans  $T$ .
- $e_i$  forme alors un cycle avec  $T$  constitué de  $e_i$  et d'une autre arête  $(v', u')$  telle que  $v' \in F_{i-1}$  et  $u' \notin F_{i-1}$
- En enlevant  $(v', u')$  nous obtenons alors un autre arbre de recouvrement  $T' \neq T$  qui inclut  $e_i$  et dont le poids total est inférieur ou égal à celui de  $T$  car le poids de  $e_i$  est inférieur ou égal à celui de  $(v', u')$ .
- Donc  $T'$  est forcément un arbre de recouvrement minimal et  $F_i$  est un sous graphe de  $T'$ . **CQFD.**



# Analyse de l'efficacité de l'algorithme de Kruskal

- Le tri de  $E$  nécessite un temps  $\in O(|E| \log |E|)$
- Pour chacun des choix gloutons, nous devons:
  - Choisir le prochain élément  $e$  de  $E$  (en un temps  $\Theta(1)$ )
  - Vérifier si  $e$  forme un cycle avec un arbre de la forêt  $F_{i-1}$ 
    - Pour cela, il faut vérifier si  $e_i = (v, u)$  est tel que  $v$  et  $u$  appartiennent au même arbre dans  $F_{i-1}$  (voir figure)
  - Si  $e$  ne forme pas de cycle: ajouter  $e_i$  à  $F_{i-1}$  pour obtenir  $F_i$



**FIGURE 9.5** New edge connecting two vertices may (a) or may not (b) create a cycle





## Analyse de l'efficacité de l'algorithme de Kruskal (suite)

- Les arbres de la forêt  $F_{i-1}$  forment une collection d'**ensembles disjoints**: chaque nœud dans  $F_{i-1}$  est élément d'un seul arbre
- Initialement, nous avons  $|V|$  ensembles disjoints  $S_1, S_2, \dots, S_{|V|}$  où chaque ensemble  $S_i$  contient un seul nœud de  $V$
- Ayant choisi l'arête  $(v,u)$  de poids minimal, il faut **trouver** l'ensemble  $S_i$  contenant  $v$  **et** l'ensemble  $S_j$  contenant  $u$ .
  - Si  $S_i = S_j$ , on ignore  $(v,u)$  et on passe à l'arête suivante;
  - sinon, on **fusionne**  $S_i$  avec  $S_j$  pour obtenir  $S_i \cup S_j$
- L'algorithme de Kruskal effectue (exactement)  $|V| - 1$  opérations **fusionner**
  - car nous avons 1 fusion par arête de l'arbre de recouvrement minimal
- L'algorithme de Kruskal effectue au plus  $2 \times |E|$  opérations **trouver**
  - car, en pire cas, toutes les arête de  $E$  seront examinées.
- Nous verrons que le total de ces opération **trouver** et **fusionner** nécessite un temps  $O(|V| \log |V| + |E|)$  ou de  $O(|V| + |E| \log |V|)$
- Le temps d'exécution total de l'algorithme de Kruskal sera alors dominé par l'étape du triage de  $E$ . **Le temps total d'exécution sera alors  $\in O(|E| \log |E|)$ .**



# Structures de données pour ensembles disjoints

---

- Examinons les structures de données pour ensembles disjoints pour effectuer, le plus efficacement possible, les opérations **trouver** et **fusionner**
- Nous avons un ensemble  $S$  de  $n$  éléments qui sont distribués, initialement, parmi  $n$  ensembles disjoints  $S_1, S_2, \dots, S_n$  où chaque  $S_i$  contient un seul élément.
- Nous effectuerons alors au plus  $n - 1$  opérations **fusionner**
  - Car, après ce nombre d'opérations, il reste forcément un seul ensemble.
- Supposons que nous désirons effectuer  $m$  opérations **trouver**
- Les opérations **trouver** sont entremêlées parmi les opérations **fusionner**



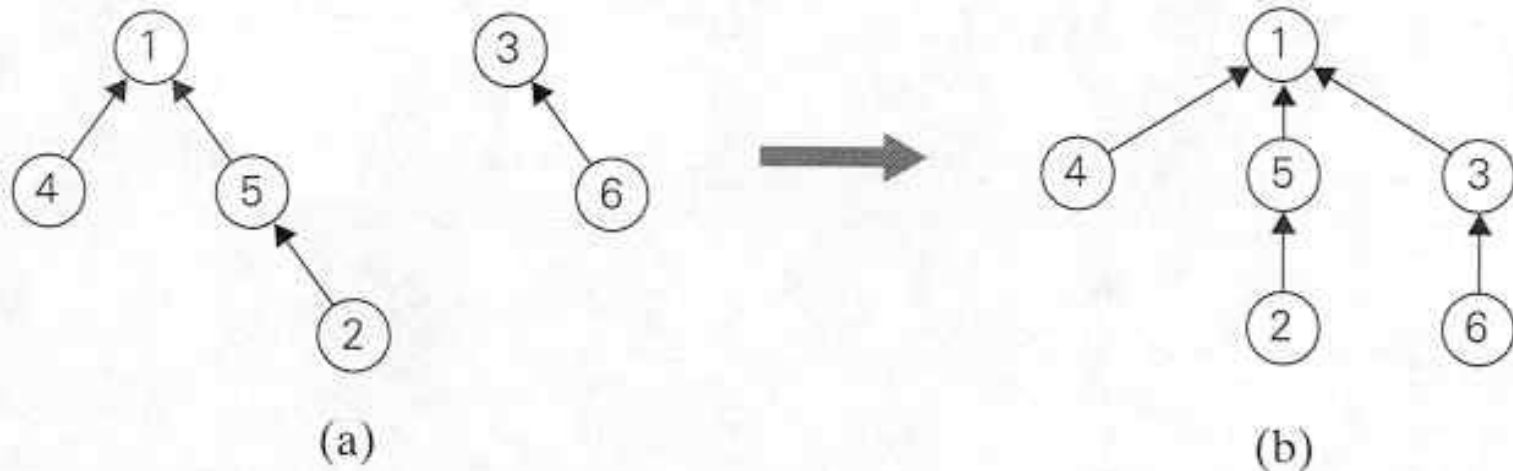
## Exemple

---

- Soit  $S = \{1,2,3,4,5,6\}$ . Initialement nous avons les ensembles disjoints:  
 $\{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}$
- Après les opérations fusionner(1,4) et fusionner(5,2), nous avons:  
 $\{1,4\}, \{5,2\}, \{3\}, \{6\}$
- Après les opérations fusionner(4,5) et fusionner(3,6), nous avons:  
 $\{1,4,5,2\}, \{3,6\}$
- Chaque ensemble est identifié par un seul de ses éléments, appelé le **représentant** (de l'ensemble)
  - Le choix du représentant est arbitraire

## La structure fusionner-rapide

- Cette structure optimise les opérations **fusionner** au prix d'un ralentissement des opérations **trouver**
- Ici la structure est celle d'une forêt: chaque arbre représente un ensemble. La racine de l'arbre est le représentant de l'ensemble.
- Pour trouver le représentant de l'ensemble contenant un élément  $x$ , il suffit de remonter du nœud  $x$  jusqu'à la racine.



**FIGURE 9.7** (a) Forest representation of subsets  $\{1, 4, 5, 2\}$  and  $\{3, 6\}$  used by *quick union*. (b) Result of *union*(5, 6).



## La structure fusionner-rapide (suite)

---

- Lorsque l'on fusionne deux ensembles, il suffit de connecter l'un des arbres à la racine de l'autre arbre.
  - Cette opération s'effectue en un temps  $\Theta(1)$  car, pour cela, il suffit de mettre à jour un seul pointeur.
- **Ainsi chaque opération fusionner s'effectue en temps  $\Theta(1)$ .**
- Par contre une opération trouver(i) pourrait nécessiter un temps  $\Theta(n)$  lorsque i est situé au niveau inférieur d'un arbre de profondeur n.
  - Rappel: trouver(i) consiste à déterminer le représentant (donc l'élément racine) de l'ensemble contenant l'élément i.
- Il faut donc éviter, le plus possible, de construire des arbres profonds.
  - Pour tenter d'arriver à cette fin, connectons la racine de l'arbre le moins profond (à celui de l'arbre le plus profond) lors de chaque opération **fusionner**.



## La structure fusionner-rapide (suite)

- **Théorème:** En utilisant cette technique pour fusionner deux ensembles, après une séquence arbitraire d'opérations **fusionner**, tout arbre contenant  $k$  nœuds aura une hauteur d'au plus  $\lfloor \lg(k) \rfloor$
- **Preuve** (par induction):
  - C'est vrai pour  $k=1$  car un arbre d'un seul nœud possède une hauteur  $= 0 = \lfloor \lg(1) \rfloor$
  - Supposons que cela soit vrai pour tous les arbres de  $m$  nœuds tels que  $1 \leq m < k$ .
  - Démontrons que cela est nécessairement vrai pour un arbre de  $k$  nœuds obtenu par la fusion de deux arbres contenant respectivement,  $a$  nœuds et  $b$  nœuds tels que  $a + b = k$
  - Nous avons nécessairement:  $1 \leq a, b < k$
  - Désignons par  $h_a$  la hauteur de l'arbre contenant  $a$  nœuds
  - Désignons par  $h_b$  la hauteur de l'arbre contenant  $b$  nœuds
  - Désignons par  $h_k$  la hauteur de l'arbre contenant  $a+b$  nœuds



## La structure fusionner-rapide (suite)

---

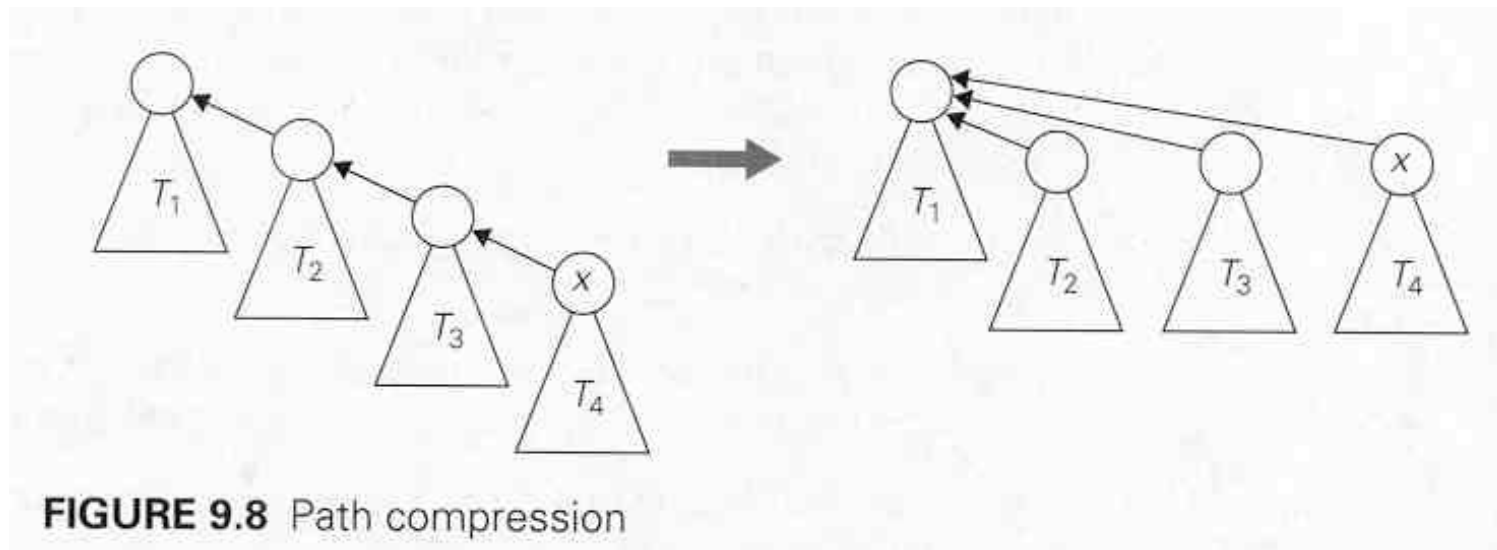
- **Preuve (...suite...):**

- Si  $h_a \neq h_b$ , alors  $h_k = \max(h_a, h_b) \leq \max(\lfloor \lg(a) \rfloor, \lfloor \lg(b) \rfloor) \leq \lfloor \lg(k) \rfloor$
- Si  $h_a = h_b$ , supposons, sans perte de généralité que  $a \leq b$ . (Nous avons donc  $a \leq k/2$ .)
  - Alors  $h_k = h_a + 1 \leq \lfloor \lg(a) \rfloor + 1 \leq \lfloor \lg(k/2) \rfloor + 1 = \lfloor \lg(k) \rfloor$
- Ainsi dans tous les cas  $h_k \leq \lfloor \lg(k) \rfloor$ . **CQFD.**

- **Ainsi m opérations trouver s'effectuent en temps  $O(m \log n)$**
- Il est cependant possible de faire mieux.

# Compression de chemins

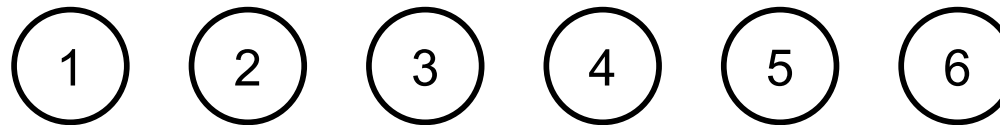
- Il est possible de réduire substantiellement la profondeur des arbres si, durant l'exécution de `trouver(x)`, nous connectons le nœud  $x$ , et chaque ancêtre du nœud  $x$ , à la racine.
- Ceci nécessite de parcourir 2 fois le chemin de  $x$  à la racine et ralentit donc d'un facteur 2 chaque opération `trouver`
- Mais chaque opération **trouver** effectuée (à l'avenir) sur cet arbre sera accélérée en raison de la réduction de la profondeur de l'arbre
- Une analyse sophistiquée montre que le temps requis pour effectuer  $m$  opérations **trouver** est presque linéaire en  $m$



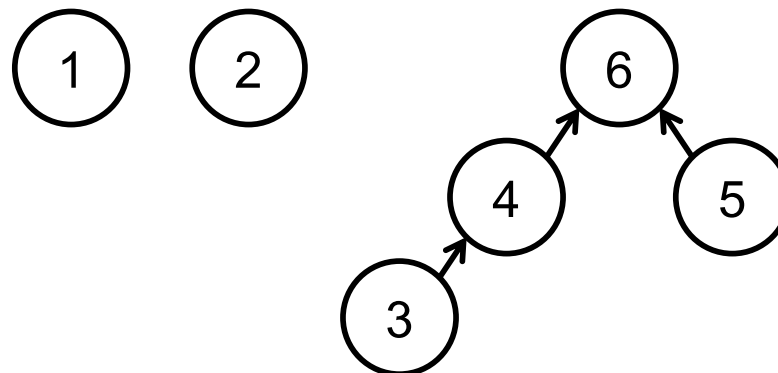


# Implémentation de la structure de données des ensembles disjoints

- Nous encodons les arbres avec un vecteur  $R[1..n]$ .
  - $R[i] < 0$  si  $i$  est un représentant.  $|R[i]|$  est un de plus que la hauteur de l'arbre
  - $R[i] \geq 0$  si  $i$  est dans le même ensemble que  $R[i]$ .
- Initialement, nous avons  $R[1..n] = [-1, -1, \dots, -1]$  ce qui correspond à une forêt d'arbres triviaux.



- Après des appels successifs à  $\text{fusion}(3, 4)$ ,  $\text{fusion}(5, 6)$ ,  $\text{fusion}(4, 6)$ , nous obtenons le vecteur  $R[1..n] = [-1, -1, 4, 6, 6, -3]$



---

**Algorithme 1** : Fusionner( $r_1, r_2, R$ )

---

// Fusionne les ensembles dont les représentants sont  $r_1$  et  $r_2$ .

// Entrée : Les représentants  $r_1$  et  $r_2$  et le vecteur  $R$  qui encode la forêt d'arbres.

// Sortie : Le vecteur  $R$  est modifié pour représenter la nouvelle forêt.

Assertion( $R[r_1] < 0 \wedge R[r_2] < 0$ ) //  $r_1$  et  $r_2$  sont des représentants.

**si**  $R[r_1] < R[r_2]$  **alors**

$R[r_2] \leftarrow r_1$

**sinon si**  $R[r_1] > R[r_2]$  **alors**

$R[r_1] \leftarrow r_2$

**sinon**

$R[r_1] \leftarrow r_2$

$R[r_2] \leftarrow R[r_2] - 1$

---

---

**Algorithme 2 : Trouver( $x, R$ )**

---

// Retourne le représentant de l'ensemble contenant l'élément  $x$

$r \leftarrow x$

**tant que**  $R[r] \geq 0$  **faire**

$r \leftarrow R[r]$

//  $r$  est le représentant

// Compression de l'arbre

**tant que**  $x \neq r$  **faire**

$t \leftarrow R[x]$

$R[x] \leftarrow r$

$x \leftarrow t$

**retourner**  $r$

---



## Analyse de la structure de données

- La fonction Fusionner s'exécute en temps  $\Theta(1)$  puisque la fonction ne comporte que des instructions élémentaires et aucune boucle.
- D'après le théorème que nous avons prouvé, la hauteur d'un arbre de  $k$  éléments ne dépasse pas  $\lfloor \lg(k) \rfloor$ . La fonction Trouver s'exécute donc en pire cas en  $\Theta(\log n)$ .
- Cependant, la compression des arbres fait en sorte qu'on ne peut pas atteindre le pire cas à chaque appel à Trouver.
- Tarjan a démontré que le temps d'exécution amorti de la fonction Trouver est  $\Theta(\alpha(n))$  où  $\alpha$  est l'inverse de la fonction d'Ackermann.

$n$	$\alpha(n)$
3	1
7	2
61	3
$2^{2^{2^{65536}}}$	4



## Retour à l'analyse de l'algorithme de Kruskal

- Résumé des opérations:
  - Tri:  $\Theta(|E| \log |E|)$
  - Entre  $|V| - 1$  et  $|E|$  opérations Trouver
    - $C_{\text{BEST}}(|V|, |E|) \in \Theta(|V| \alpha(|V|))$
    - $C_{\text{WORST}}(|V|, |E|) \in \Theta(|E| \alpha(|V|))$
  - $|V| - 1$  opérations Fusionner:  $\Theta(|V|)$
- Le tri domine toutes les opérations.
- L'algorithme de Kruskal s'exécute donc en temps  $\Theta(|E| \log |E|)$ .
- Note:
  - Si les poids des arêtes sont des entiers entre 1 et  $|E|$ , on peut utiliser le tri par dénombrement dont l'efficacité est  $\Theta(|E|)$ .
  - L'algorithme de Kruskal s'exécute alors en temps
    - $C_{\text{BEST}}(|V|, |E|) \in \Theta(|E| + |V| \alpha(|V|))$
    - $C_{\text{WORST}}(|V|, |E|) \in \Theta(|E| \alpha(|V|))$