Révision Intra

1) Notation asymptotique:

- On classe les algorithmes selon leur complexité en temps dans le pire des cas, en moyenne ou dans le meilleur des cas
  - Dans le cours, on s'est intéressé à la complexité dans le pire des cas

- Analyse théorique
  - Pour calculer la complexité en temps d'un algorithme, on compte le nombre d'opérations élémentaires qu'on doit exécuter dans le pire des cas
  - On utilise la notation asymptotique pour exprimer la complexité d'un algorithme

2) Grand $\mathcal{O}$:

- Soit une fonction $f : \mathbb{N} \longrightarrow \mathbb{R}^{\geq 0}$. On définit l'ordre de $f(n)$ par $\mathcal{O}(f(n))$ comme l'ensemble des fonctions bornées supérieurement par $f(n)$:

$$\mathcal{O}(f(n)) = \{t : \mathbb{N} \longrightarrow \mathbb{R}^{\geq 0} \mid \exists c \in \mathbb{R}^{+}, \exists n_0 \in \mathbb{N}\ t.q.\ \forall n > n_0, t(n) \leq cf(n)\}$$

- Règle du seuil: Soient deux fonctions strictement positives $f, t : \mathbb{N} \longrightarrow \mathbb{R}^{+}$ alors

$$t(n) \in \mathcal{O}(f(n)) \Longleftrightarrow \exists c' \in \mathbb{R}^{+}\ t.q.$$

$$\forall n \in \mathbb{N}, t(n) \leq c' f(n)$$

2) Grand $\mathcal{O}$ (suite):

- La relation $\mathcal{O}$ est réflexive et transitive i.e.
  - $t(n) \in \mathcal{O}(t(n))$
  - Si $f(n) \in \mathcal{O}(g(n))$ et $g(n) \in \mathcal{O}(h(n))$ alors $f(n) \in \mathcal{O}(h(n))$

- Règle du maximum: Soient deux fonctions $f, g : \mathbb{N} \longrightarrow \mathbb{R}^{\geq 0}$ alors

$$\mathcal{O}(f(n) + g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$$

- $\mathcal{O}(f(n)) = \mathcal{O}(g(n)) \Longleftrightarrow f(n) \in \mathcal{O}(g(n))$ et $g(n) \in \mathcal{O}(f(n))$

3) Grand Ω:

- Soit une fonction $f : \mathbb{N} \longrightarrow \mathbb{R}^{\geq 0}$. On définit $\Omega(f(n))$ comme l'ensemble des fonctions bornées inférieurement par $f(n)$

$$\Omega(f(n)) = \{t : \mathbb{N} \longrightarrow \mathbb{R}^{\geq 0} \,|\, \exists c \in \mathbb{R}^{+}, \exists n_0 \in \mathbb{N} \ t.q. \quad \forall n \geq n_0, \ t(n) \geq cf(n)\}$$

- Règle de dualité:

$$t(n) \in \Omega(f(n)) \Longleftrightarrow f(n) \in \mathcal{O}(t(n))$$

4) Grand Θ:

- On dit que $t(n)$ est dans l'ordre exact de $f(n)$, dénoté $t(n) \in \Theta(f(n))$, si

$$t(n) \in \mathcal{O}(f(n)) \cap \Omega(f(n))$$

De façon équivalente:

$$\Theta(f(n)) = \{t : \mathbb{N} \longrightarrow \mathbb{R}^{\geq 0} \mid \exists c, d \in \mathbb{R}^{+}, \exists n_0 \in \mathbb{N}\ t.q. \\ \forall n \geq n_0,\ cf(n) \leq t(n) \leq df(n)\}$$

5) Règle de la limite:

- Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} \in \mathbb{R}^+$ alors $f(n) \in \Theta(g(n))$
  ou $\mathcal{O}(f(n)) = \mathcal{O}(g(n))$

- Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ alors $f(n) \in \mathcal{O}(g(n))$ et
  $g(n) \notin \mathcal{O}(f(n))$

- Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty$ alors $g(n) \in \mathcal{O}(f(n))$ et
  $f(n) \notin \mathcal{O}(g(n))$

6) Notation asymptotique conditionnelle:

- Soient $f, g : \mathbb{N} \longrightarrow \mathbb{R}^{\geq 0}$ et $P : \mathbb{N} \longrightarrow \{\text{vrai}, \text{faux}\}$

$$\mathcal{O}(f(n) \mid P(n)) = \{t : \mathbb{N} \longrightarrow \mathbb{R}^{\geq 0} \mid \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}\ t.q.$$

$$\forall n \geq n_0, \text{ si } P(n) \to t(n) \leq cf(n)$$

- De la même façon, on peut définir $\Omega(f(n)|P(n))$ et $\Theta(f(n)|P(n))$

7) Règle de l'harmonie

Définitions:

- Une fonction $f : \mathbb{N} \longrightarrow \mathbb{R}^{\geq 0}$ est éventuellement non décroissante ou e.n.d. s'il $\exists n_0 \in \mathbb{N}$ tel que

$$\forall n \geq n_0, \quad f(n+1) \geq f(n)$$

- Une fonction $f : \mathbb{N} \longrightarrow \mathbb{R}^{\geq 0}$ est b-harmonieuse pour $b \in \mathbb{N},\ b \geq 2$ si

  1) elle est e.n.d.

  2) $f(bn) \in \mathcal{O}(f(n))$

- Une fonction $f : \mathbb{N} \longrightarrow \mathbb{R}^{\geq 0}$ est harmonieuse si elle est b-harmonieuse pour tout $b \in \mathbb{N},\ b \geq 2$

7) Règle de l'harmonie (suite)

- Règle de l'harmonie:

Soit $f : \mathbb{N} \longrightarrow \mathbb{R}^{\geq 0}$ une fonction harmonieuse, soit $b \in \mathbb{N},\ b \geq 2$ et soit $t : \mathbb{N} \longrightarrow \mathbb{R}^{\geq 0}$ une fonction e.n.d. alors

$$t(n) \in \Theta(f(n) \mid n \text{ puissance de } b)$$

$$\Longleftrightarrow$$

$$t(n) \in \Theta(f(n))$$

8) Résolution de récurrences linéaires homogènes à coefficients constants:

Soit la récurrence R

$$a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = 0$$

Voici les étapes de la résolution:

1) Trouver le polynôme caractéristique $P(x)$ de la récurrence R

2) Trouver les racines de $P(x)$

Si ces racines sont distinctes

3) La solution générale est de la forme $t_n = \sum_{i=1}^{k} c_i r_i^n$

4) Résoudre le système d'équations linéaires donné par les conditions initiales pour trouver la valeur des constantes $c_1, c_2, \ldots, c_k$

5) Écrire la solution t_n en fonction de ces constantes c_i

8) Résolution de récurrences linéaires homogènes à coefficients constants:

Soit la récurrence R

$$a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = 0$$

Voici les étapes de la résolution:

1) Trouver le polynôme caractéristique $P(x)$ de la récurrence R

2) Trouver les racines de $P(x)$

Si ces racines ne sont pas toutes distinctes

3) La solution générale est de la forme $t_n = \sum_{i=1}^{\ell} \sum_{j=0}^{m_i - 1} c_{ij} n^j r_i^n$

où on a ℓ racines r_i de multiplicité m_i

4) Résoudre le système d'équations linéaires donné par les conditions initiales pour trouver la valeur des constantes $c_1, c_2, \ldots, c_k$

5) Écrire la solution t_n en fonction de ces constantes c_i

9) Résolution de récurrences linéaires non-homogènes à coefficients constants (cas particulier):

Soit la récurrence R

$$a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = b^n$$

où b est une constante.

Voici les étapes de la résolution:

1) On commence par transformer cette récurrence en une récurrence homogène

Pour ce cas particulier, on peut multiplier la récurrence R par b
et ensuite remplacer n par $n-1$ dans l'équation obtenue

Si on soustrait de la récurrence initiale, cette nouvelle récurrence, nous obtenons une récurrence homogène R^*

2) Résoudre R^* (cas homogène) comme d'habitude mais en s'assurant que les conditions initiales satisfont aussi l'équation de départ

$$a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = b^n$$

10) Changement de variables:

- Il arrive qu'il soit plus facile de commencer par faire un changement de variables lorsque l'on veut résoudre une récurrence

- Par exemple, soit $T(n) = 4t(n/2) + n^2$. En faisant le changement de variables $t_i = T(2^i)$ on obtient la récurrence linéaire non-homogène $t_i - 4t_{i-1} = 4^i$

- On résout cette récurrence de la façon habituelle et ensuite on exprime la solution obtenue pour les t_i en fonction des $T(n)$ en utilisant le fait que $n = 2^i$ et donc que $i = \log_2 n$

11) Algorithmes voraces:

- Facile à développer
- On choisit un optimum local sans se soucier des effets dans le futur (pas de retour en arrière)
- On aimerait que cette stratégie locale nous amène à un optimum global
- On doit faire une preuve d’optimalité pour démontrer qu’un algorithme vorace trouve la solution optimale
- Exemples vus en classe:
  - Retour Monnaie
  - Arbre couvrant minimal (Kruskal - Prim)
  - Plus courts chemins (Dijkstra)
  - Sac à dos
  - Files d’attente simples

11) Algorithmes voraces - Retour Monnaie:

Problème: On a un nombre illimité de pièces de différentes valeurs. On veut faire la monnaie d'un montant n de sorte qu'on retourne le moins de pièces possibles

Solution vorace: On commence par donner le maximum de pièces de la plus grande valeur (optimum local) et ensuite on complète le montant n avec les pièces de valeurs plus petites

Optimalité: L'optimalité dépend ici de la valeur des pièces en notre possession et du fait qu'on suppose qu'on a un nombre illimité de chacune des pièces.

11) Algorithmes voraces - Arbre Couvrant Minimal:

Problème: Étant donné $G = (N, A)$ un graphe non-orienté connexe et $c : A \longrightarrow \mathbb{R}^+$ une fonction de coût, on veut trouver $A' \subseteq A$ tel que $T(N, A')$ est un arbre et tel que la somme

$$c(T) = \sum_{a \in A'} c(A)$$

est minimale.

Solutions voraces:

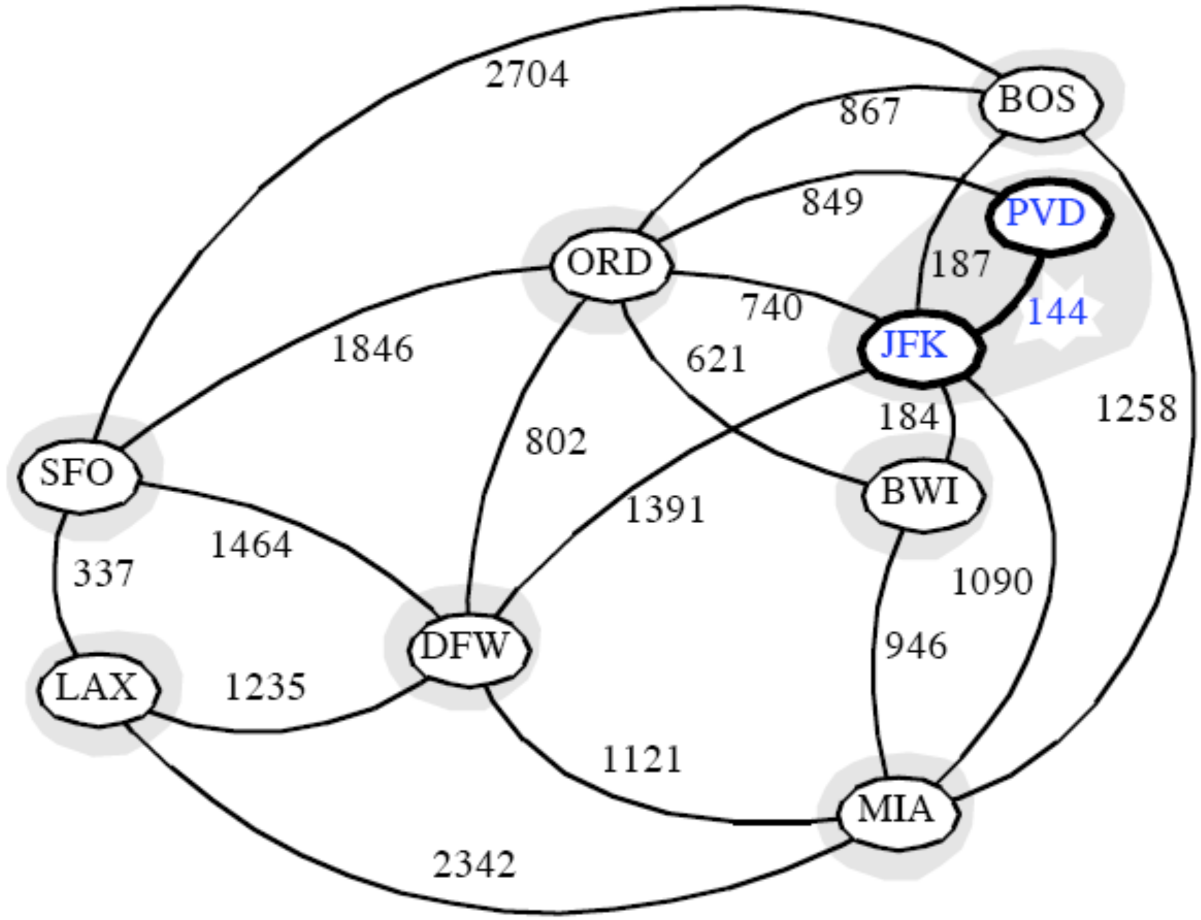
1) Commencer par un ensemble vide d'arêtes et sélectionner à chaque étape l'arête de plus petit coût qui n'a pas encore été choisie ou rejetée (peut importe où elle se situe dans le graphe) — Kruskal

2) Commencer dans un sommet du graphe et construire un arbre à partir de ce sommet en sélectionnant à chaque étape l'arête de coût minimal qui ajoute à l'arbre existant un nouveau noeud — Prim

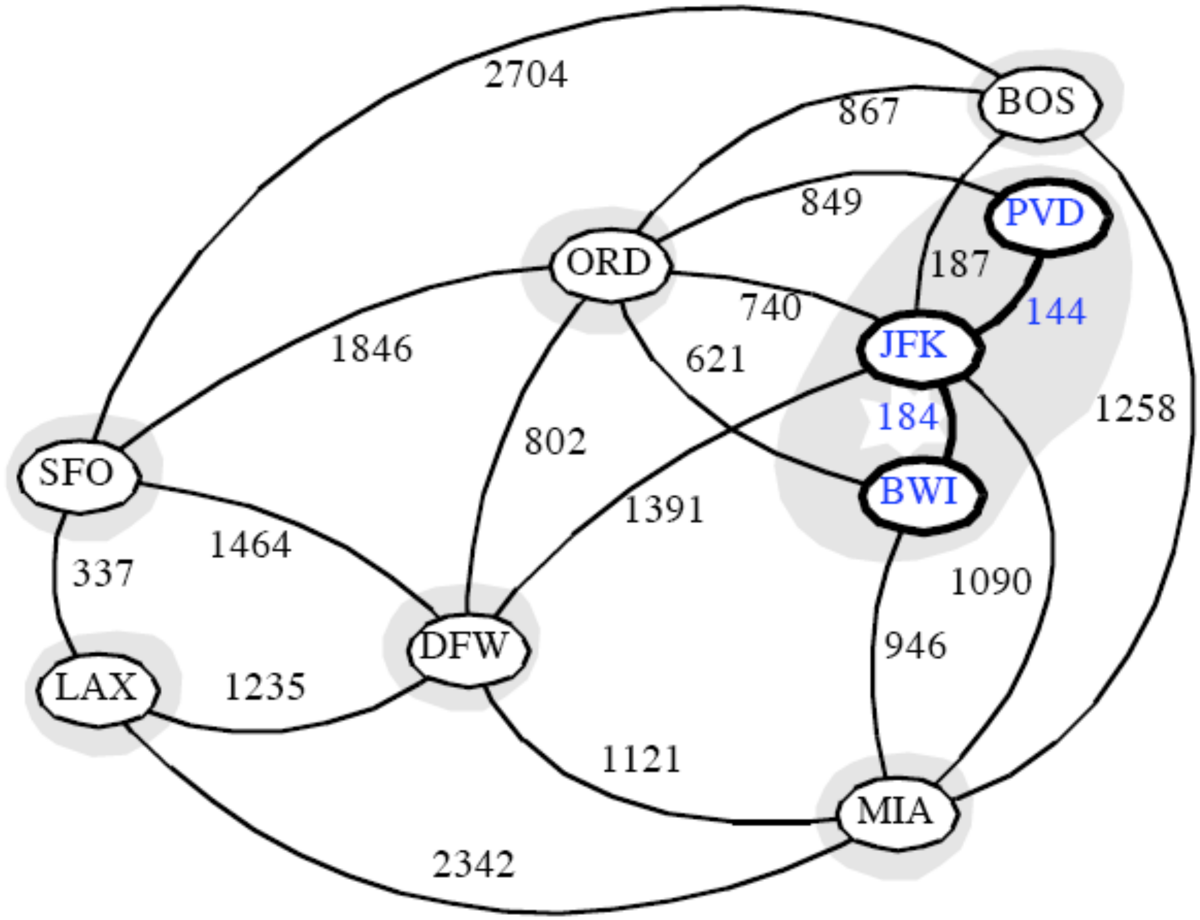
11) Algorithmes voraces - Arbre Couvrant Minimal -Kruskal:

- L'algorithme maintient une forêt d'arbres
- À chaque itération, on choisit l'arête de coût minimal
- Cette arête est acceptée, si elle relit deux arbres distincts, sinon elle est rejetée (pourrait forme un cycle)
- L'algorithme se termine lorsqu'on a un seul arbre

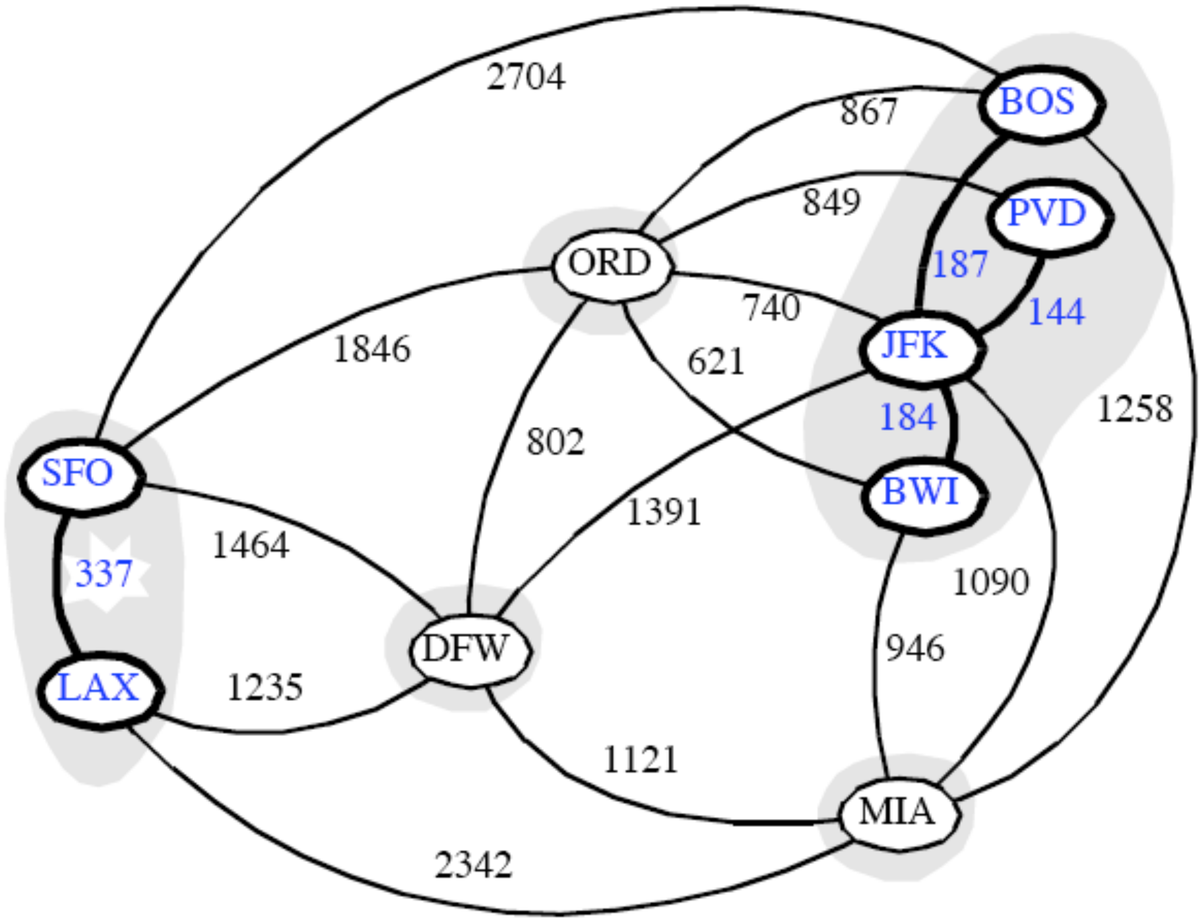
Exemple de Kruskal:



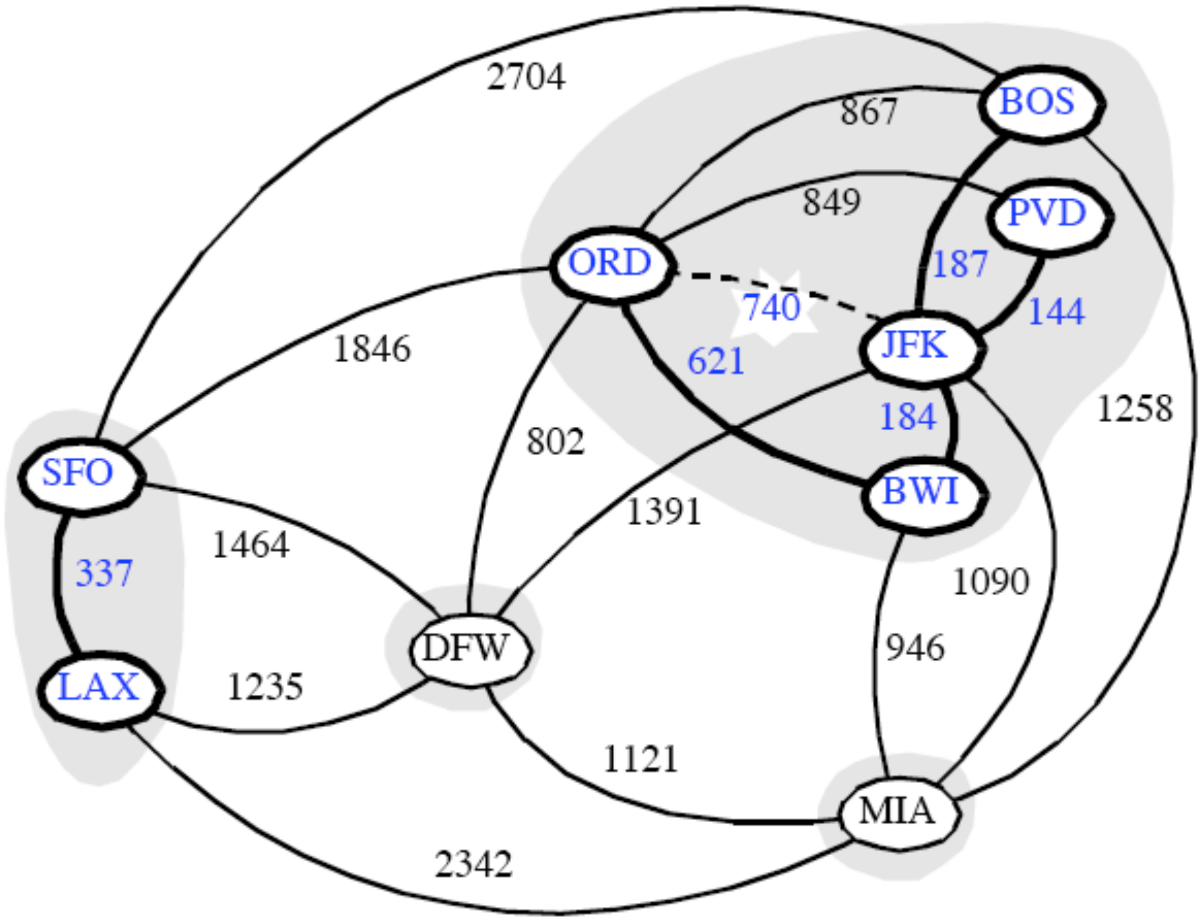
Exemple de Kruskal:



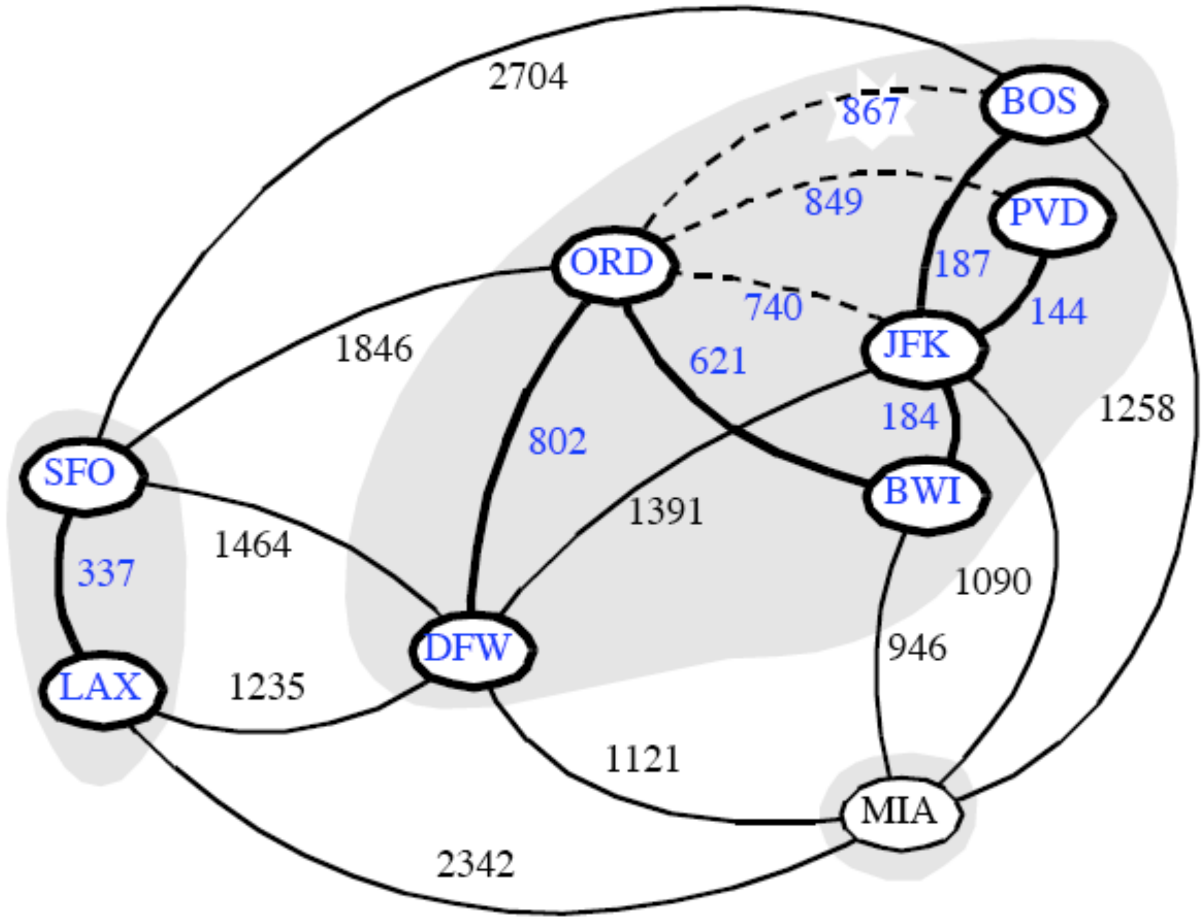
Exemple de Kruskal:



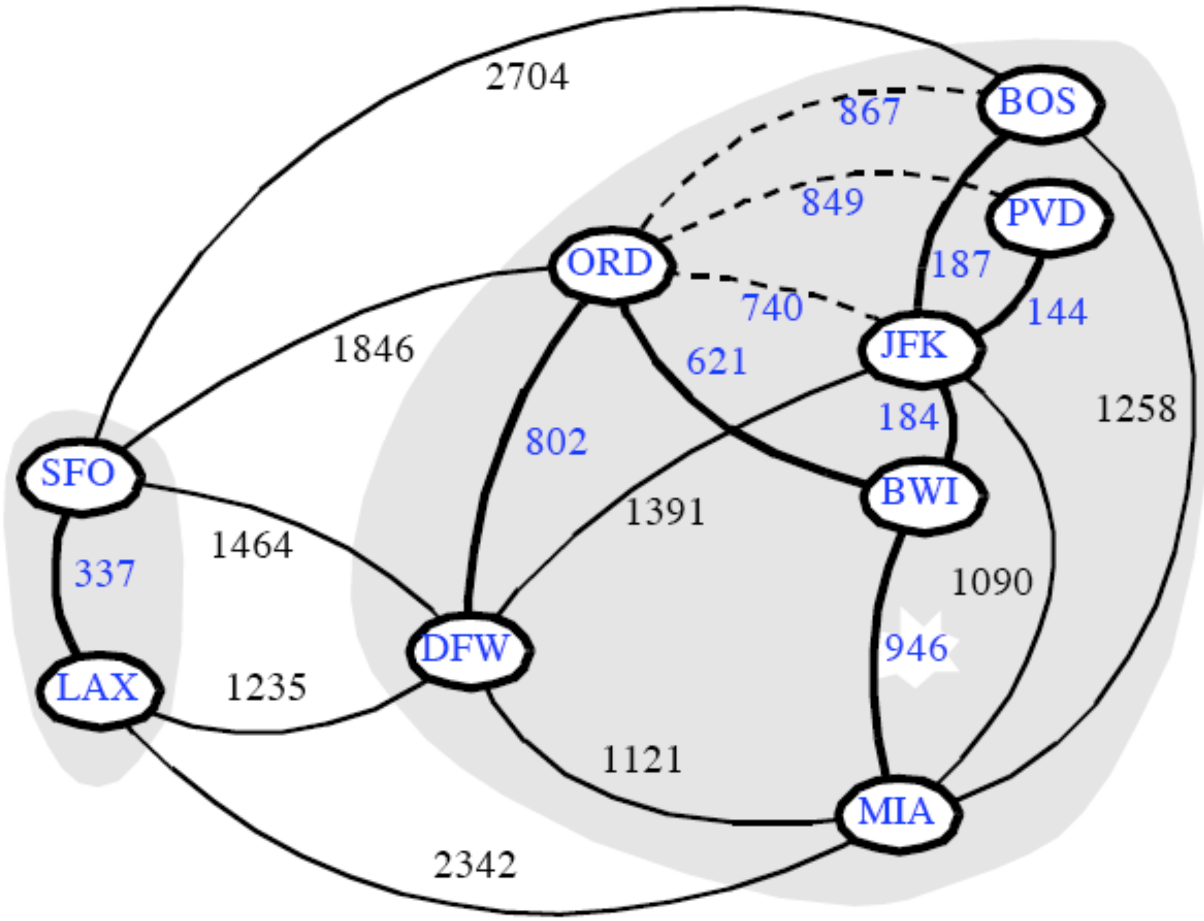
Exemple de Kruskal:



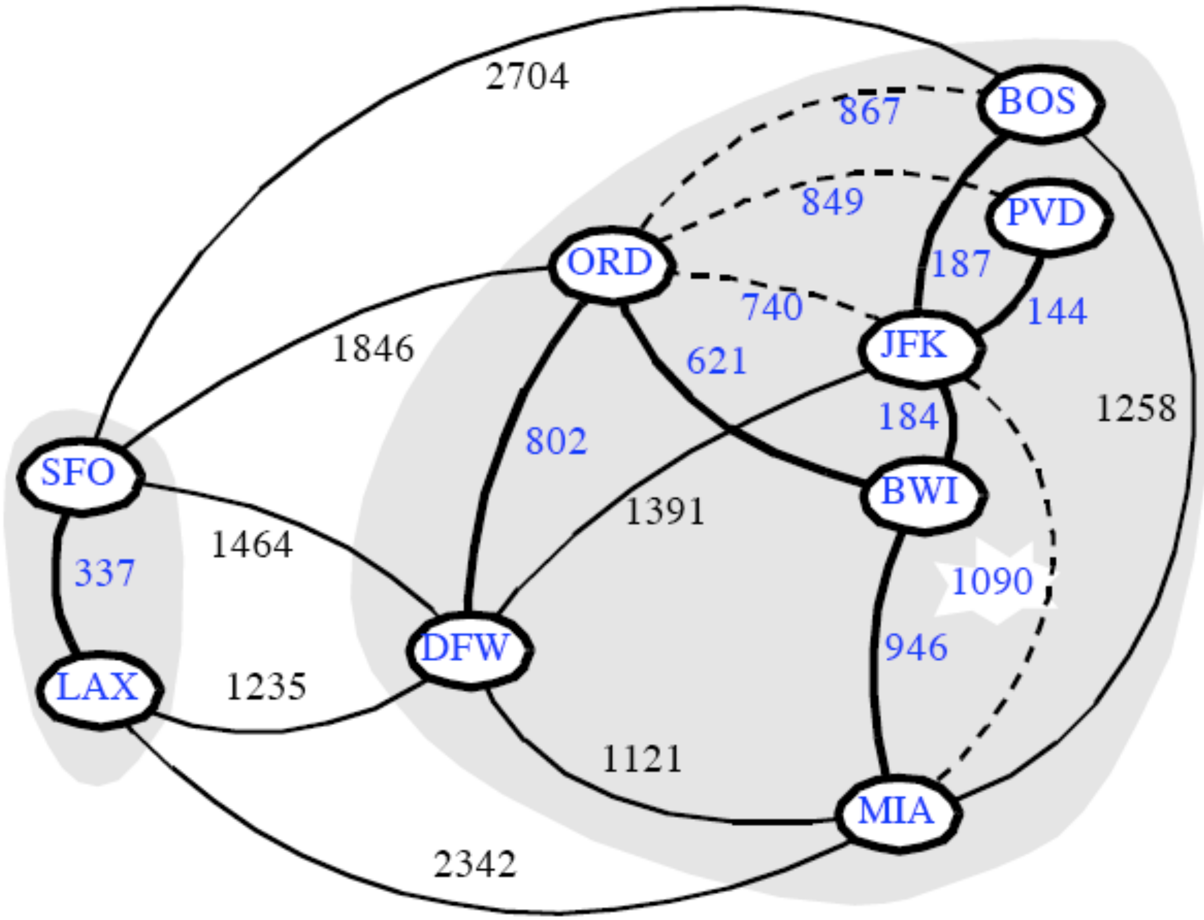
Exemple de Kruskal:



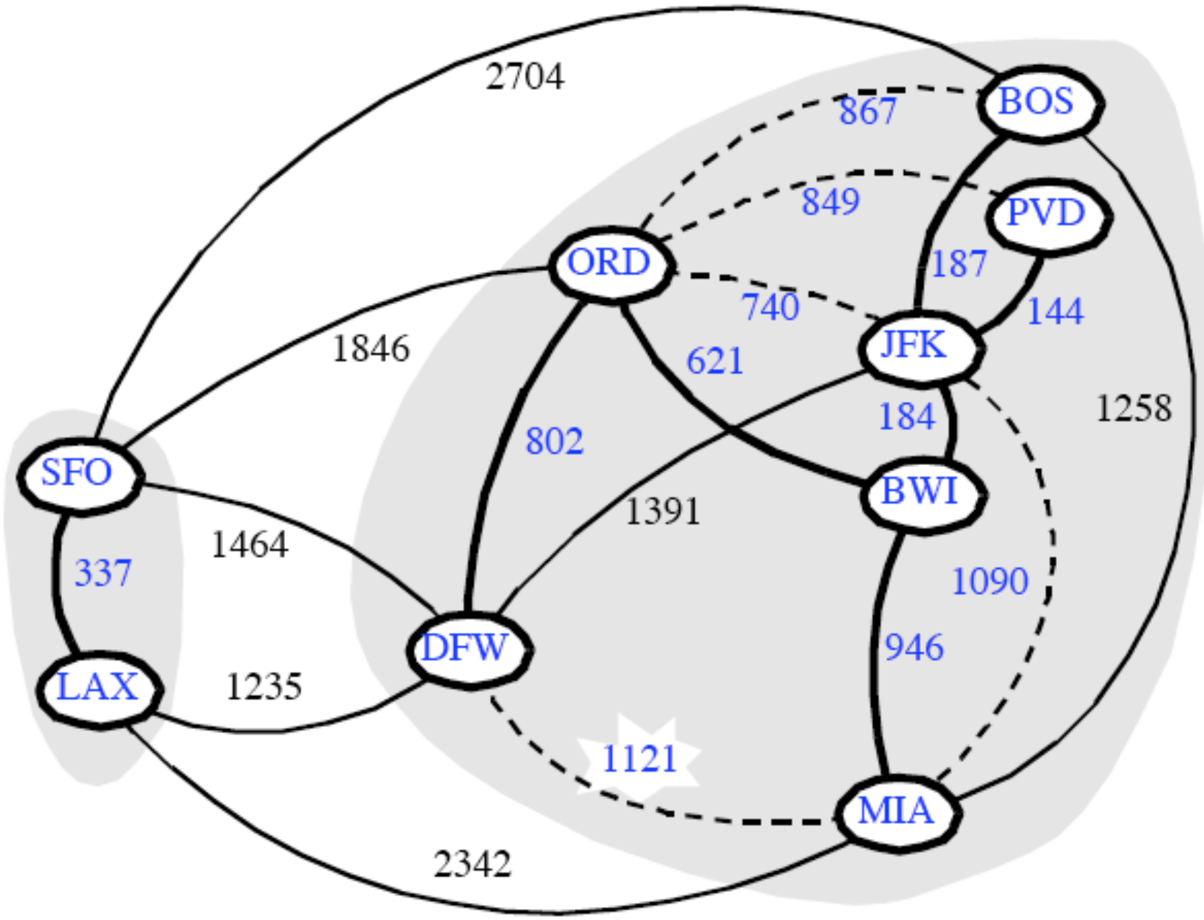
Exemple de Kruskal:



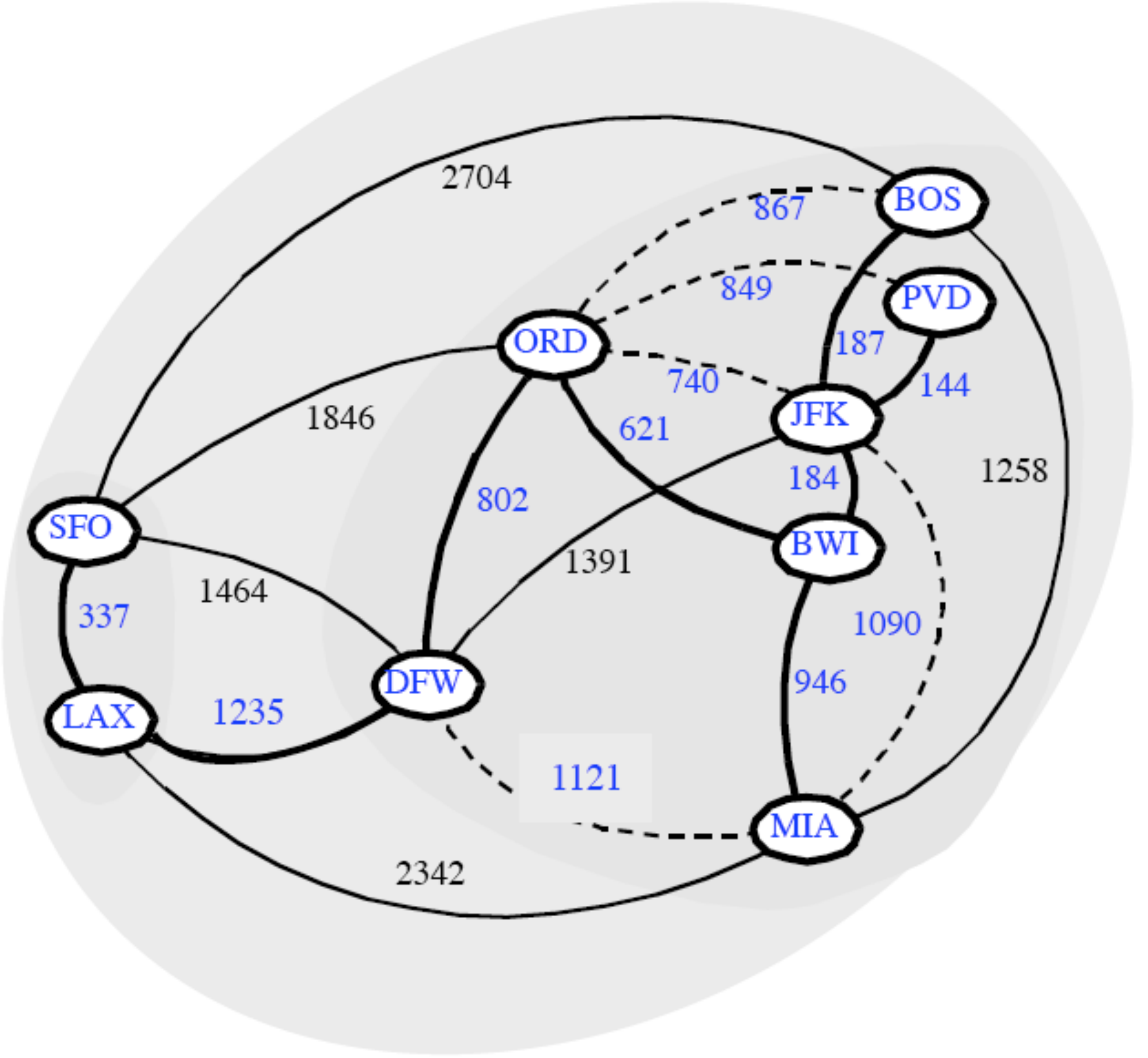
Exemple de Kruskal:



Exemple de Kruskal:



Exemple de Kruskal:



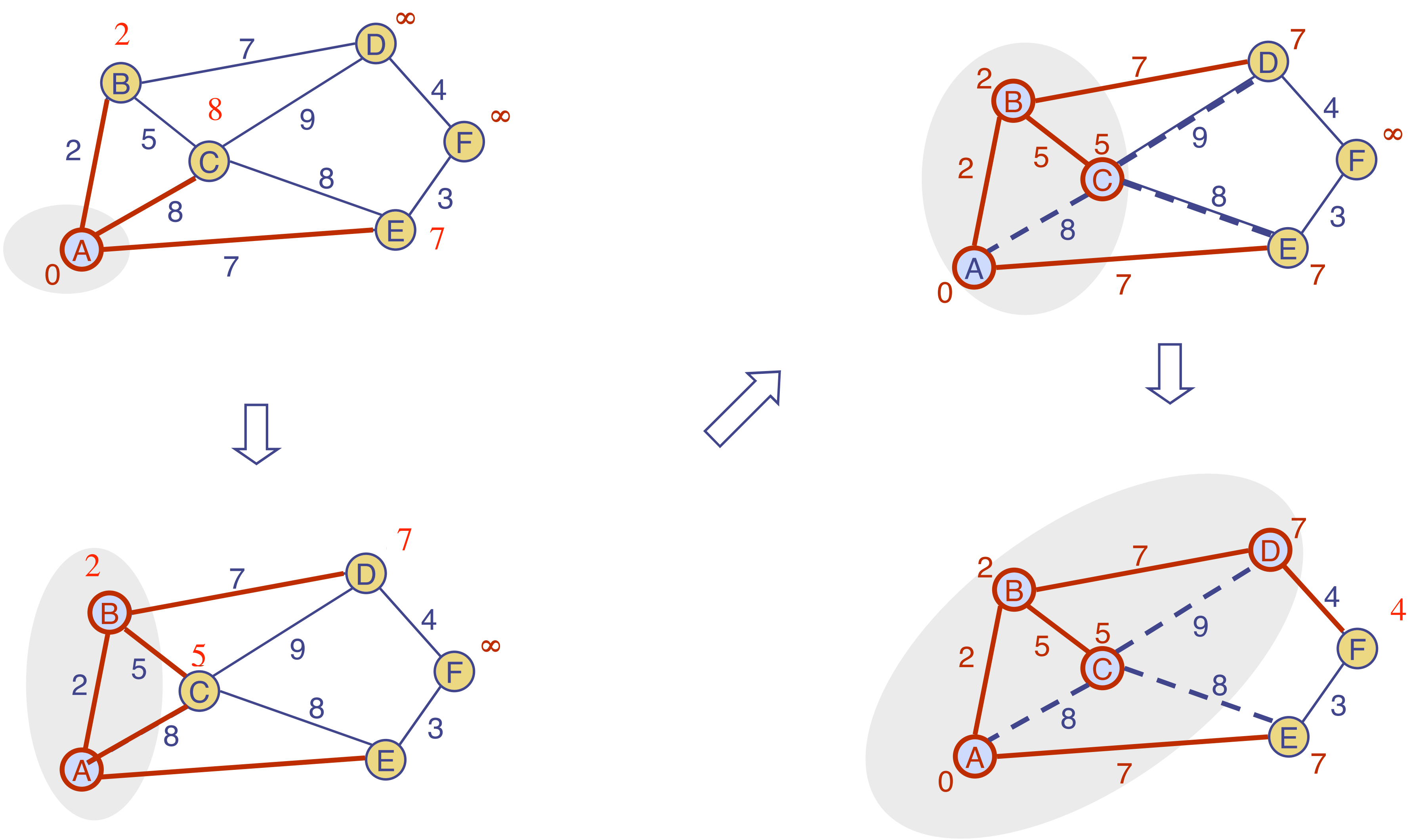
11) Algorithmes voraces - Arbre Couvrant Minimal -Prim:

- On choisit un sommet s aléatoirement qu'on met dans un "nuage" et on construit l'arbre couvrant minimal en faisant grossir le "nuage" d'un sommet à la fois.

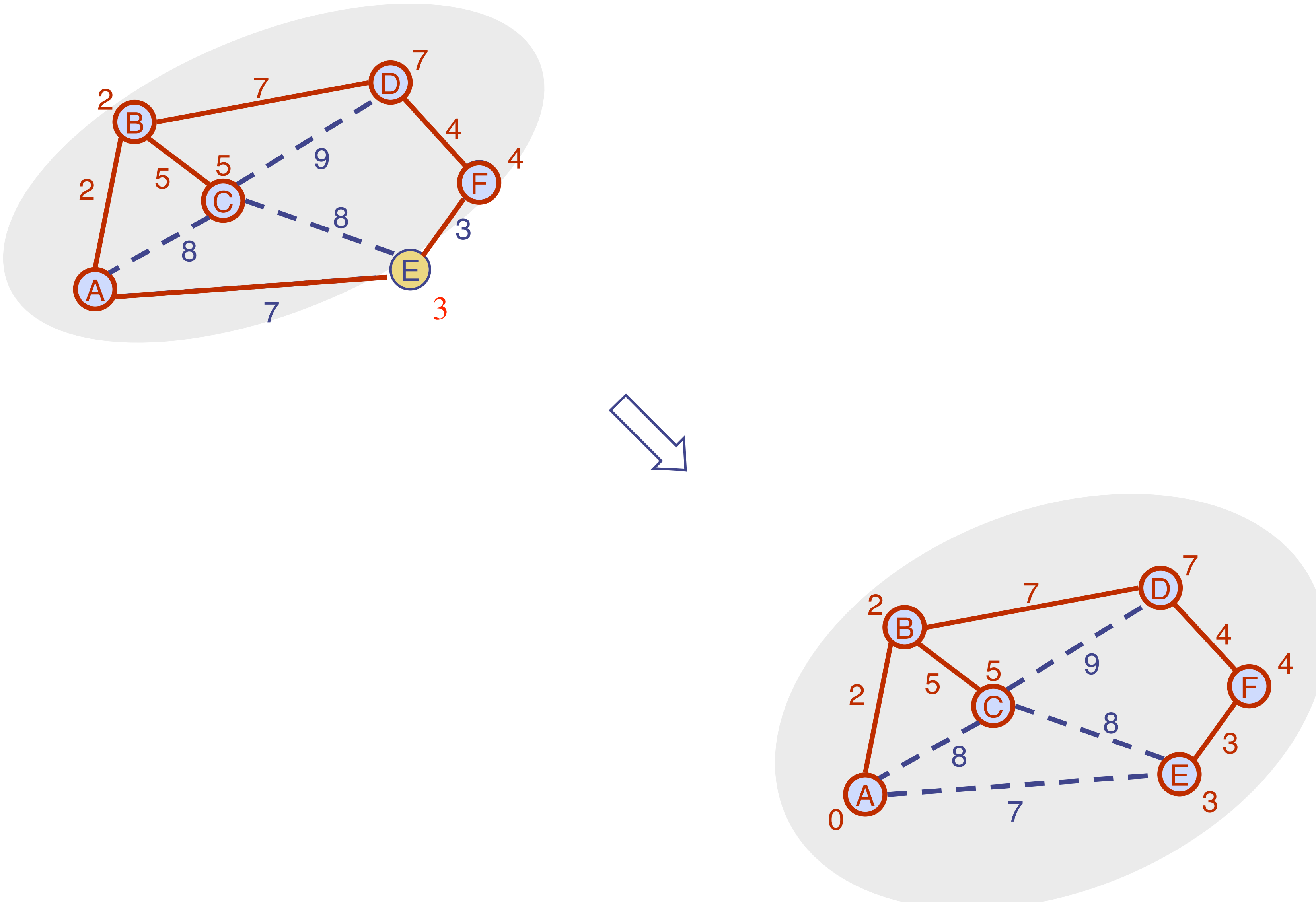
- On garde en mémoire à chaque sommet v, une étiquette d(v) qui ici est égale au poids minimal parmi les poids des arêtes reliant v à un sommet à l'intérieur du nuage.

- À chaque étape:
  - On ajoute au nuage le sommet u extérieur ayant la plus petite étiquette d(u)
  - On met à jour les étiquettes des sommets adjacents à u

Exemple:

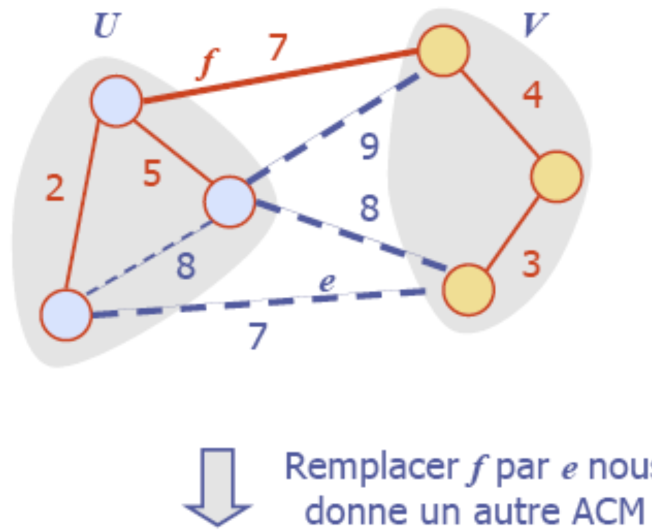


Exemple (suite)



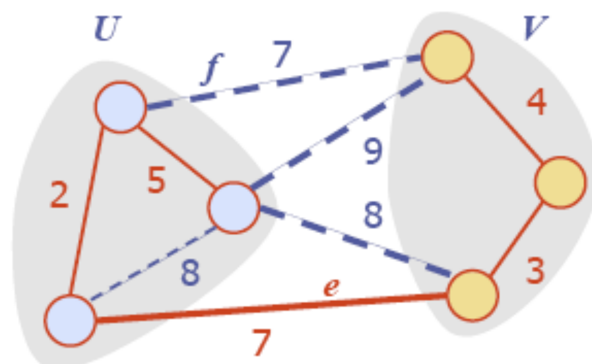
11) Algorithmes voraces - Arbre Couvrant Minimal - Optimalité:

- **Propriété de partition:**
  - Considérons une partition des sommets de G en deux ensembles U et V
  - Soit e une arête de poids minimal entre U et V
  - Alors, il existe un arbre couvrant minimal de G contenant e



- **Preuve:**
  - Soit T un arbre couvrant minimal de G
  - Si T ne contient pas e, soit C le cycle formé par l'addition de e à l'arbre T et soit f, une arête entre U et V
  - Par la propriété de cycles, on a que

$$poids(f) \leq poids(e)$$



  - Comme on avait pris e de poids minimal, on a que poids(f) = poids(e) et alors on obtient un autre ACM en remplaçant f par e

11) Algorithmes voraces - Arbre Couvrant Minimal - Optimalité:

L’optimalité des deux algorithmes découle de la propriété de partition des ACMs

- Kruskal:

  La partition du graphe considérée est ici, étant donné une arête (u,v) de coût minimum, d’un côté tous les sommets faisant partie de la composante connexe de u et de l’autre tous les autres sommets.
  Si u et v ne font pas partie de la même composante connexe, la propriété de partition garantie que l’arête (u,v) fait partie d’un ACM

- Prim:

  Ici, la partition considérée est nuage/ non-nuage

11) Algorithmes voraces - Plus courts chemins

Problème: Soit $G = (N, A)$ un graphe non-orienté (ou orienté) connexe et $c : A \longrightarrow \mathbb{R}^+$ une fonction de coût. Étant donné un sommet source, on veut trouver les plus courts chemins de cette source à tous les autres sommets du graphe.

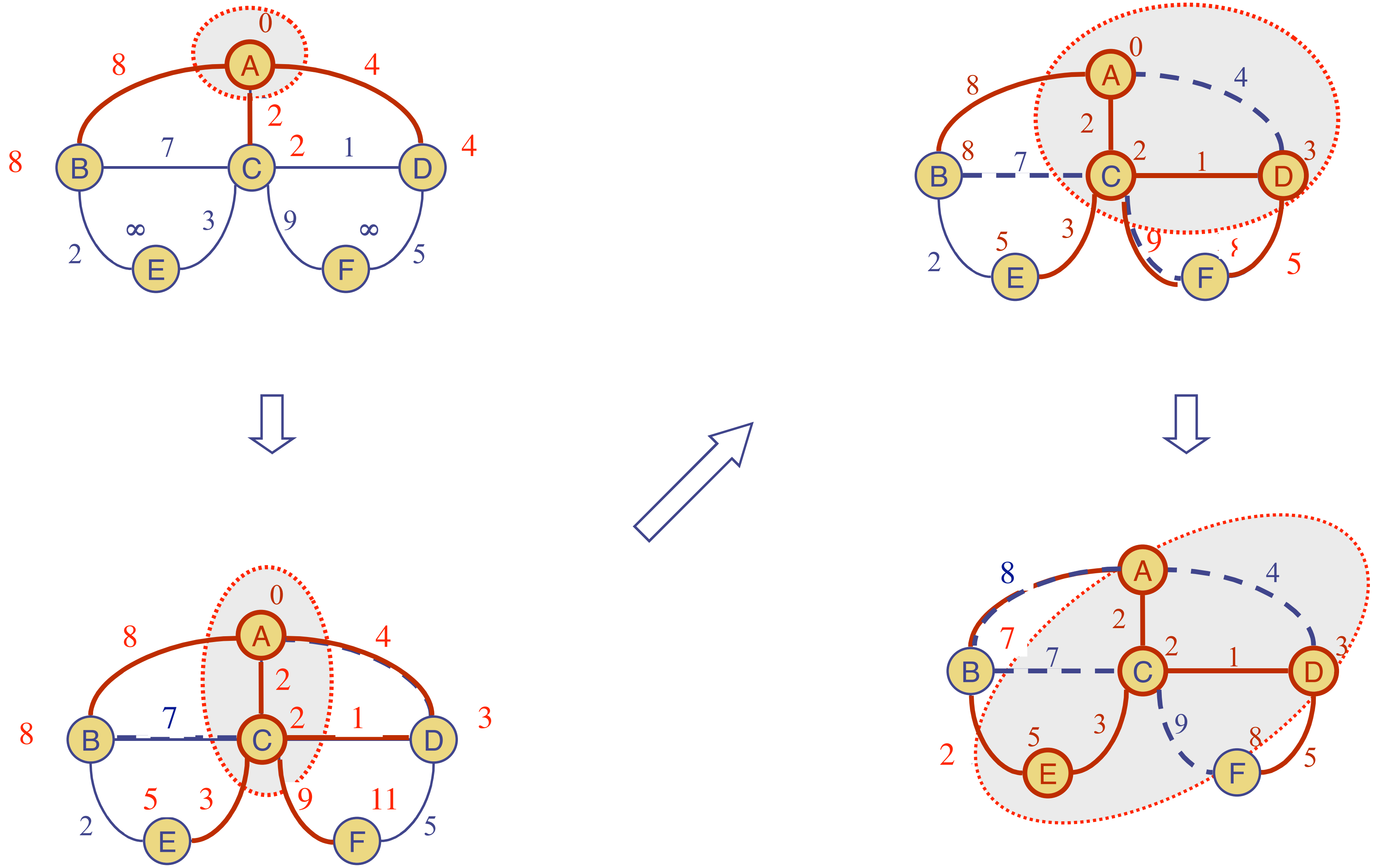
Solution vorace:

Algorithme de Dijkstra

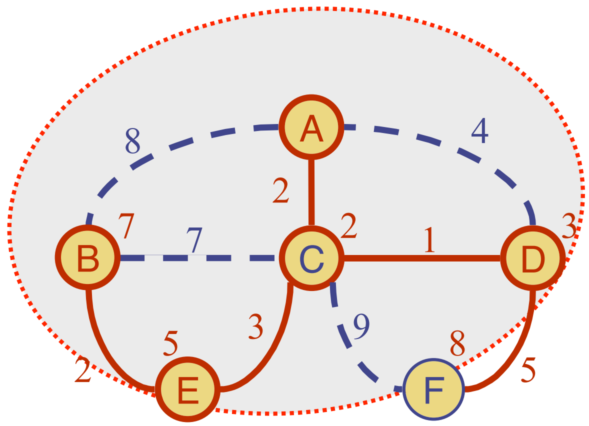
Algorithme de Dijkstra

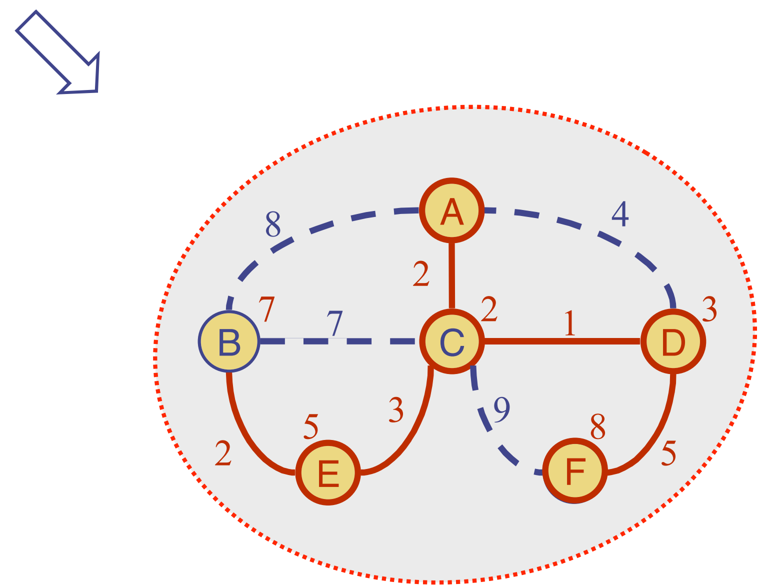
- La distance entre un sommet v et un sommet s est le poids total minimal d'un chemin entre v et s
- L'algorithme de Dijkstra est un algorithme glouton qui calcule les distances entre un sommet v et tous les autres sommets d'un graphe
- Ici, on va assumer:
  - Le graphe est connexe
  - Les poids des arêtes sont non-négatifs
- On va faire grossir un "nuage" de sommets, contenant au départ v et couvrant éventuellement tous les sommets
- On va donner une étiquette d(u) à chaque sommet, représentant la distance entre v et u dans le sous-graphe constitué des sommets dans le nuage et des arêtes adjacentes
- À chaque étape:
  - On ajoute au nuage le sommet u extérieur au nuage qui a la plus petite étiquette d(u)
  - On met à jour les étiquettes des sommets adjacents à u

Exemple:



Exemple (suite):





11) Algorithmes voraces - Plus courts chemins

Problème: Soit $G = (N, A)$ un graphe non-orienté (ou orienté) connexe et $c : A \longrightarrow \mathbb{R}^+$ une fonction de coût. Étant donné un sommet source, on veut trouver les plus courts chemins de cette source à tous les autres sommets du graphe.

Solution vorace:

Algorithme de Dijkstra

Preuve d'optimalité: vient de la proposition suivante:

Proposition: Avec Dijkstra, chaque fois qu'un sommet u est inclus dans le nuage, D(u) est le coût d'un chemin minimal entre u et le sommet source et ce chemin est inclus entièrement dans le nuage

11) Algorithmes voraces - Sac à dos

Problème: On dispose de n objets de poids positifs $w_1, w_2, \ldots, w_n$ et de valeurs positives $v_1, v_2, \ldots, v_n$. Notre sac à doc à une capacité maximale en poids de W

Notre but est de remplir le sac de sorte de maximiser la valeur des objets inclus dans le sac tout en respectant la contrainte de poids.

Pour pouvoir résoudre se problème avec un algorithme vorace, on suppose qu'on peut apporter une fraction x_i de chaque objet $i, \quad 0 \leq x_i \leq 1$.

But: Maximiser $\displaystyle\sum_{i=1}^{n} x_i v_i$ tel que $\displaystyle\sum_{i=1}^{n} x_i w_i = W$ et $0 \leq x_i \leq 1$

11) Algorithmes voraces - Sac à dos

Solution vorace:

Sélectionner chaque objet à son tour dans un certain ordre, mettre la plus grande fraction possible de cet objet dans le sac (sans dépasser la capacité en poids du sac) et arrêter quand le sac est plein.

On sélectionne l'objet dans la valeur par unité de poids est maximale

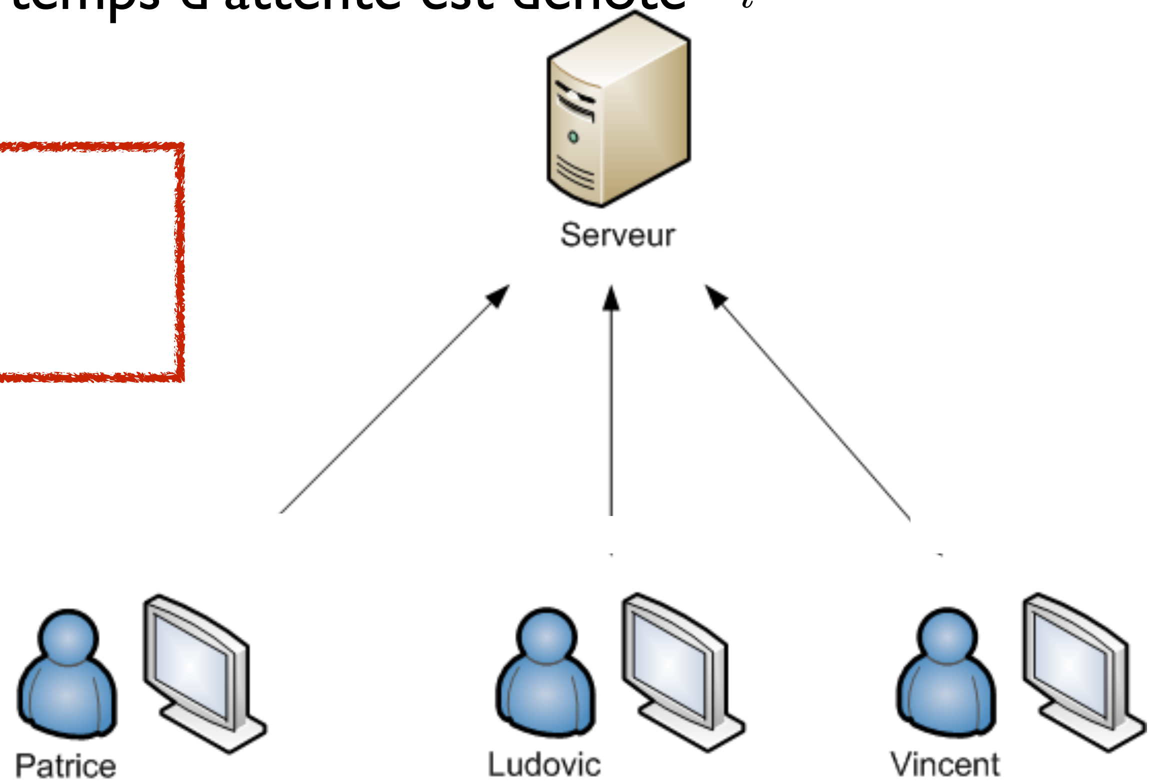
Preuve d'optimalité: vient du théorème suivant:

Théorème: Si les objets sont choisis par ordre décroissant de $\frac{v_i}{w_i}$ alors cette stratégie retourne une solution optimale

11) Algorithmes voraces - File d'attente simple

- On a un serveur et n clients
- Chaque client i a besoin d'un temps de service t_i
- Si un client i doit attendre, son temps d'attente est dénoté a_i

Problème: Minimiser $\sum_{i=1}^{n} t_i + a_i$



http://web.univ-pau.fr/~puiseux/enseignement/python/tutoQt-zero/qt15/tutoriel-3-11396-0-communiquer-en-reseau-avec-son-programme.html

Problème de la file d'attente simple

Théorème: Si on sert les clients selon un ordre croissant du temps de service demandé, l'algorithme vorace trouver une solution optimale.

Preuve d'optimalité: On suppose qu'une solution optimale existe où l'ordre de service des clients est différent de l'ordre croissant du temps de service demandé et on en dérive une contradiction.