DIU Enseignement de l'Info au Lycée

Cours 4 : Quelques compléments et exemples d'algorithmes

Université de Montpellier

2018 – 2019

1. Borne inférieure pour le tri

2. Arbre couvrant de poids minimum

3. Algorithme des k plus proches voisins

4. Tour de Hanoï

1. Borne inférieure pour le tri

2. Arbre couvrant de poids minimum

3. Algorithme des k plus proches voisins

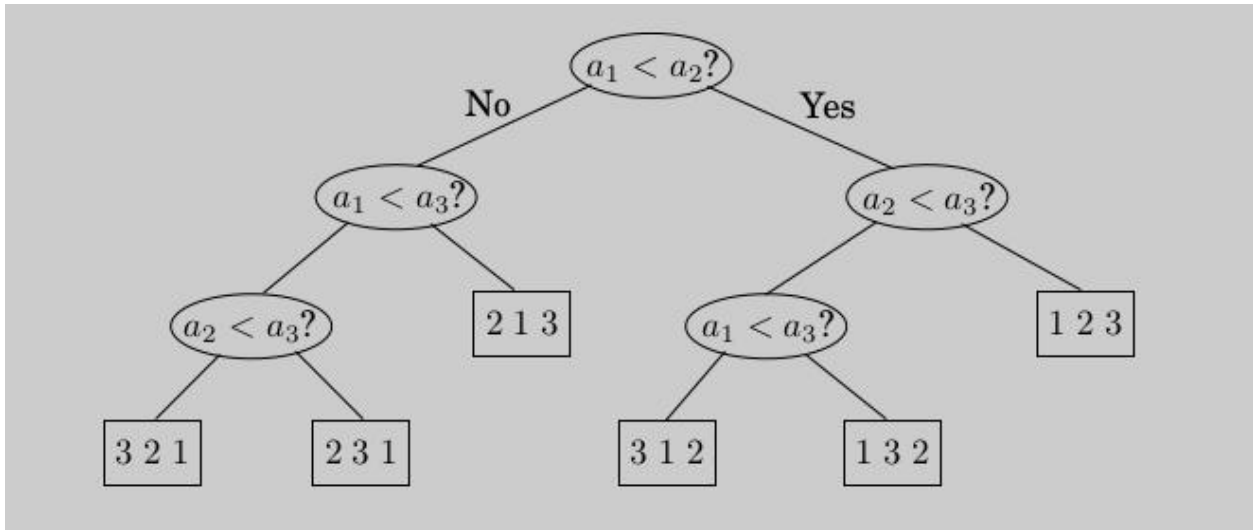
4. Tour de Hanoï

Borne inférieure pour le tri

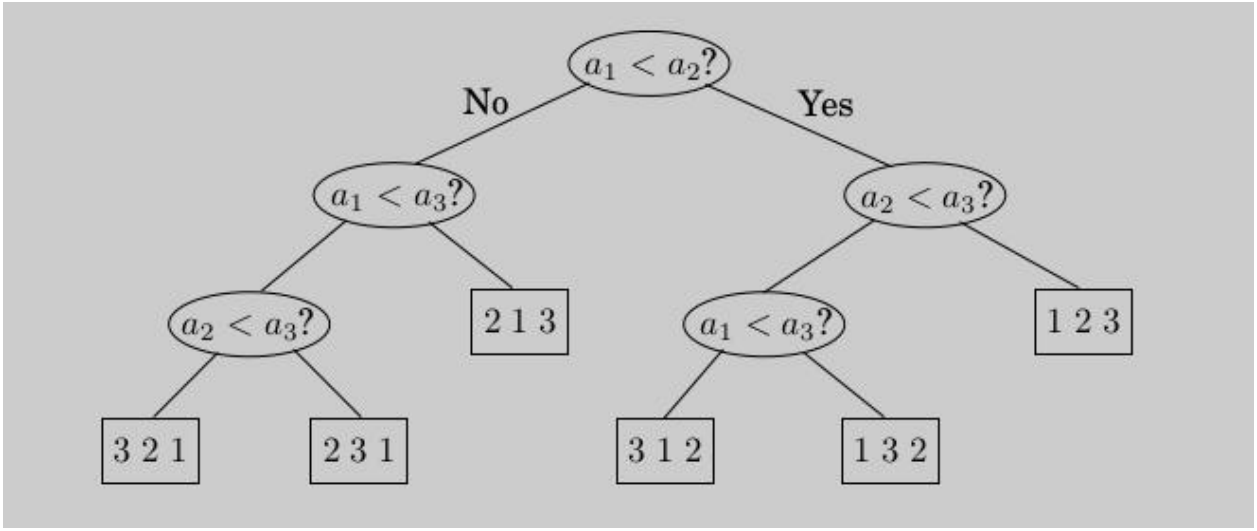
- On peut représenter un algorithme de tri par un arbre de décision :
  Chaque nœud interne de l'arbre correspond à une comparaison effectuée par l'algorithme. Les solutions du tri sont alors représentées par les feuilles de l'arbre.

Borne inférieure pour le tri

- On peut représenter un algorithme de tri par un arbre de décision :
  Chaque nœud interne de l'arbre correspond à une comparaison effectuée par l'algorithme. Les solutions du tri sont alors représentées par les feuilles de l'arbre.
- Par exemple, si on lance un TriBulles sur un tableau $T = [a_1, a_2, a_3]$, on commence par comparer a_1 et a_2
  Si $a_1 \leq a_2$ alors on compare a_2 et a_3, sinon on compare a_1 et a_3, etc...

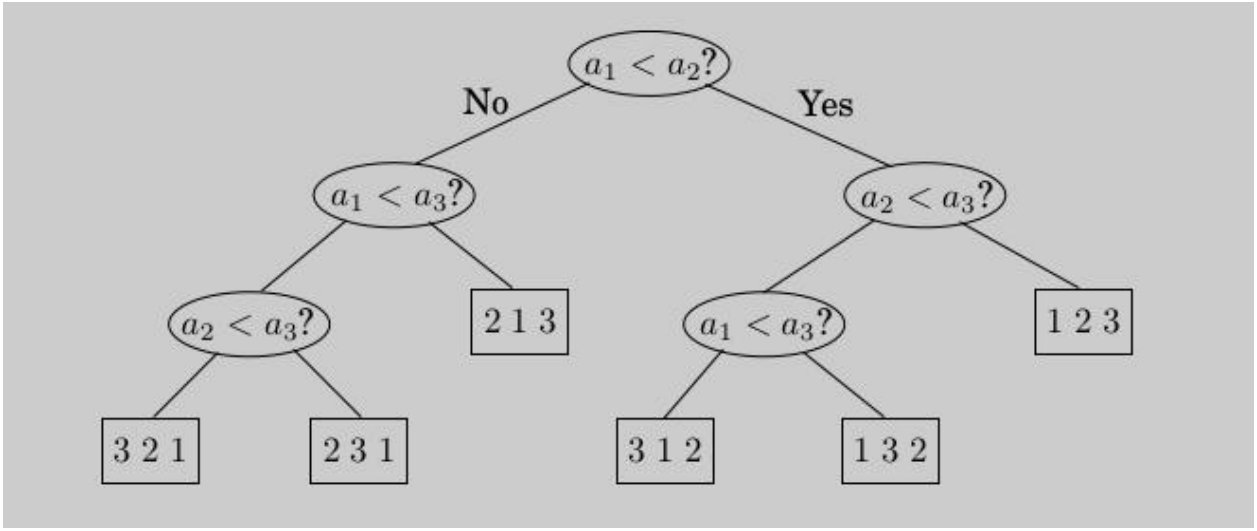


Borne inférieure pour le tri



- Sur les feuilles de l'arbre, la permutation 3 2 1 signifie que le tableau T trié est $[a_3, a_2, a_1]$
- Les 3 !=6 permutations possibles doivent toutes apparaître comme feuille de l'arbre au moins une fois.

Borne inférieure pour le tri



- Sur les feuilles de l'arbre, la permutation 3 2 1 signifie que le tableau T trié est $[a_3, a_2, a_1]$
- Les 3 !=6 permutations possibles doivent toutes apparaître comme feuille de l'arbre au moins une fois.
- Dans cet exemple, dans le pire des cas, il faut faire au moins 3 comparaisons (branche de gauche).

Borne inférieure pour le tri

Cas général : trier un tableau T de taille n :

- L'arbre de décision est binaire et doit avoir au moins $n!$ feuilles
- Dans le pire des cas, le nombre de comparaisons à effectuer correspond à un plus long chemin de la racine à une feuille, c'est-à-dire, à **la hauteur de l'arbre**.

Borne inférieure pour le tri

Cas général : trier un tableau T de taille n :

- L'arbre de décision est binaire et doit avoir au moins $n!$ feuilles
- Dans le pire des cas, le nombre de comparaisons à effectuer correspond à un plus long chemin de la racine à une feuille, c'est-à-dire, à **la hauteur de l'arbre**.

Lemme

Un arbre binaire de hauteur h possède au plus 2^h feuilles.

(C'est le cas pour un arbre binaire complet)

Borne inférieure pour le tri

Cas général : trier un tableau T de taille n :

- L'arbre de décision est binaire et doit avoir au moins $n!$ feuilles
- Dans le pire des cas, le nombre de comparaisons à effectuer correspond à un plus long chemin de la racine à une feuille, c'est-à-dire, à **la hauteur de l'arbre**.

Lemme

Un arbre binaire de hauteur h possède au plus 2^h feuilles.

(C'est le cas pour un arbre binaire complet)

- L'arbre de décision a au moins $n!$ feuilles, sa hauteur h vérifie :

$$2^h \geq n!$$

C'est-à-dire : $h \geq \log(n!)$

Borne inférieure pour le tri

- On a : $n! = n \times (n-1) \times (n-2) \times \cdots \times (n/2) \times (n/2-1) \times \cdots \times 2 \times 1$

Borne inférieure pour le tri

- On a : $n! = n \times (n-1) \times (n-2) \times \cdots \times (n/2) \times (n/2-1) \times \cdots \times 2 \times 1$
- Donc : $n! \geq (n/2) \times (n/2) \times (n/2) \times \cdots \times (n/2) \times 1 \times \cdots \times 1 \times 1$

  Soit $n! \geq (n/2)^{n/2}$

Borne inférieure pour le tri

- On a : $n! = n \times (n-1) \times (n-2) \times \cdots \times (n/2) \times (n/2-1) \times \cdots \times 2 \times 1$
- Donc : $n! \geq (n/2) \times (n/2) \times (n/2) \times \cdots \times (n/2) \times 1 \times \cdots \times 1 \times 1$

  Soit $n! \geq (n/2)^{n/2}$
- La hauteur h de l'arbre de décision vérifie :
  $h \geq \log(n!) \geq \log((n/2)^{n/2}) = (n/2).\log(n/2) = (n/2).(\log n - 1)$

Borne inférieure pour le tri

- On a : $n! = n \times (n-1) \times (n-2) \times \cdots \times (n/2) \times (n/2-1) \times \cdots \times 2 \times 1$
- Donc : $n! \geq (n/2) \times (n/2) \times (n/2) \times \cdots \times (n/2) \times 1 \times \cdots \times 1 \times 1$

  Soit $n! \geq (n/2)^{n/2}$
- La hauteur h de l'arbre de décision vérifie :
  $h \geq \log(n!) \geq \log((n/2)^{n/2}) = (n/2).\log(n/2) = (n/2).(\log n - 1)$
- Or, $\log n - 1 \geq (\log n)/2$ dès que $n \geq 4$, donc on a :

$$\mathbf{h \geq \frac{1}{4} n.\log n}$$

Borne inférieure pour le tri

- On a : $n! = n \times (n-1) \times (n-2) \times \cdots \times (n/2) \times (n/2-1) \times \cdots \times 2 \times 1$
- Donc : $n! \geq (n/2) \times (n/2) \times (n/2) \times \cdots \times (n/2) \times 1 \times \cdots \times 1 \times 1$

  Soit $n! \geq (n/2)^{n/2}$
- La hauteur h de l'arbre de décision vérifie :
  $h \geq \log(n!) \geq \log((n/2)^{n/2}) = (n/2).\log(n/2) = (n/2).(\log n - 1)$
- Or, $\log n - 1 \geq (\log n)/2$ dès que $n \geq 4$, donc on a :

$$\mathbf{h \geq \frac{1}{4} n.\log n}$$

- Dans le pire des cas, notre algorithme fera au moins $\frac{1}{4} n.\log n$ comparaison.

Borne inférieure pour le tri

Théorème
Tout algorithme de tri effectuera $\Omega(n.\log n)$ *comparaisons pour trier un tableau de taille* n.

Remarques :

- En terme de complexité algorithmique, TriFusion a une complexité optimale.

Borne inférieure pour le tri

Théorème
Tout algorithme de tri effectuera $\Omega(n.\log n)$ *comparaisons pour trier un tableau de taille* n.

Remarques :

- En terme de complexité algorithmique, TriFusion a une complexité optimale.
- En pratique, ce n'est pourtant pas forcément le plus rapide, TriRapide (QuickSort) est généralement considéré comme le tri le plus rapide, bien qu'il soit de complexité $O(n^2)$ (dans le pire des cas...)

Une autre borne inférieure classique

- Pour le problème algorithmique suivant :

NombresEgaux$(x_1, x_2, \ldots, x_n)$
Entrée : Une suite de n nombres (entiers) : $(x_1, x_2, \ldots, x_n)$
Sortie : VRAI si il existe $i \neq j$ tels que $x_i = x_j$, FAUX sinon

Une autre borne inférieure classique

- Pour le problème algorithmique suivant :

NombresEgaux$(x_1, x_2, \ldots, x_n)$
Entrée : Une suite de n nombres (entiers) : $(x_1, x_2, \ldots, x_n)$
Sortie : VRAI si il existe $i \neq j$ tels que $x_i = x_j$, FAUX sinon

- On a le théorème suivant :

Théorème
Tout algorithme de résolution de NombresEgaux *a une complexité en* $\Omega(n.\log n)$

Une autre borne inférieure classique

- Pour le problème algorithmique suivant :

NombresEgaux$(x_1, x_2, \ldots, x_n)$
Entrée : Une suite de n nombres (entiers) : $(x_1, x_2, \ldots, x_n)$
Sortie : VRAI si il existe $i \neq j$ tels que $x_i = x_j$, FAUX sinon

- On a le théorème suivant :

Théorème
Tout algorithme de résolution de NombresEgaux *a une complexité en* $\Omega(n.\log n)$

Remarques :

- Du coup, l'algorithme consistant en 'trier les $(x_1, x_2, \ldots, x_n)$ puis vérifier si il existe deux nombres consécutifs égaux' est optimal en complexité.

Une autre borne inférieure classique

- Pour le problème algorithmique suivant :

NombresEgaux$(x_1, x_2, \ldots, x_n)$
Entrée : Une suite de n nombres (entiers) : $(x_1, x_2, \ldots, x_n)$
Sortie : VRAI si il existe $i \neq j$ tels que $x_i = x_j$, FAUX sinon

- On a le théorème suivant :

Théorème
Tout algorithme de résolution de NombresEgaux *a une complexité en* $\Omega(n.\log n)$

Remarques :

- Du coup, l'algorithme consistant en 'trier les $(x_1, x_2, \ldots, x_n)$ puis vérifier si il existe deux nombres consécutifs égaux' est optimal en complexité.
- Cette borne inf est beaucoup (beaucoup) plus dure à établir que pour le tri...

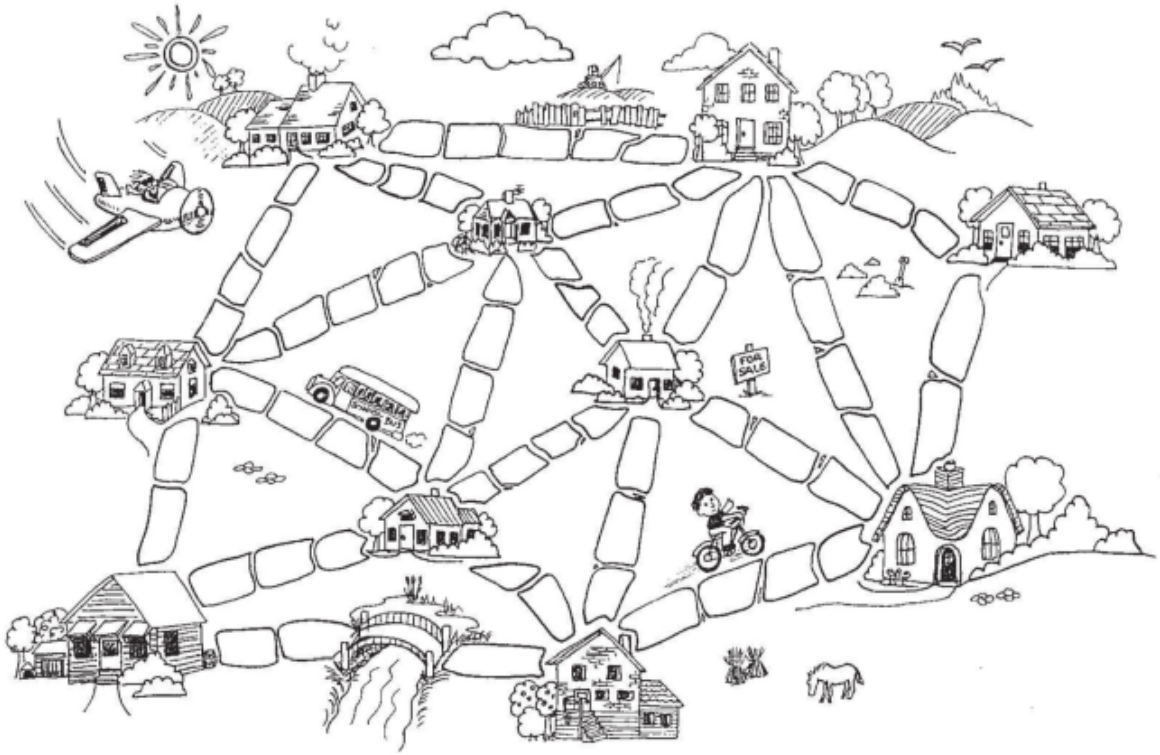
1. Borne inférieure pour le tri

2. Arbre couvrant de poids minimum

3. Algorithme des k plus proches voisins

4. Tour de Hanoï

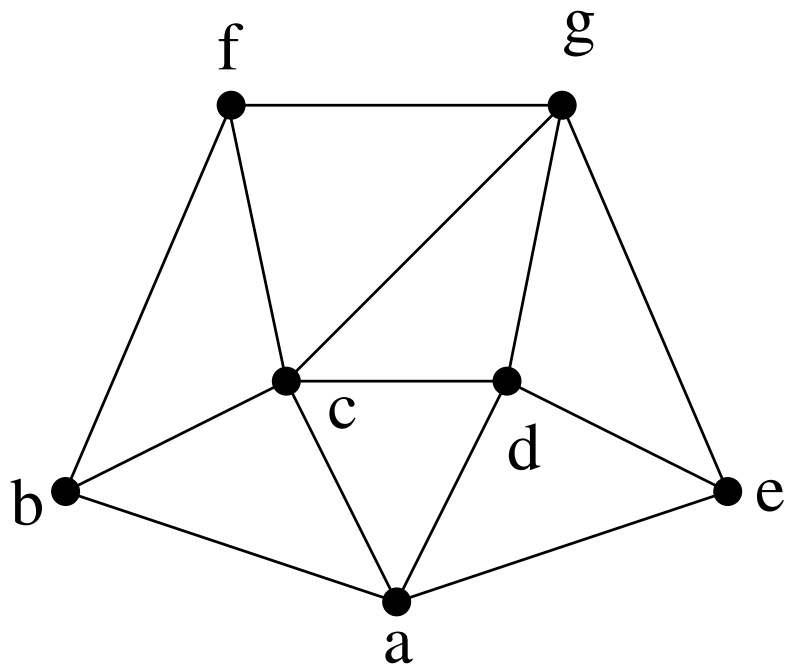
La ville embourbée



- Comment paver la ville (sur les emplacements possibles) avec le nombre minimum de pavés afin que chacun puisse toujours atteindre n'importe quel lieu en ne marchant que sur des pavés (ou sur le pont) ? [1]

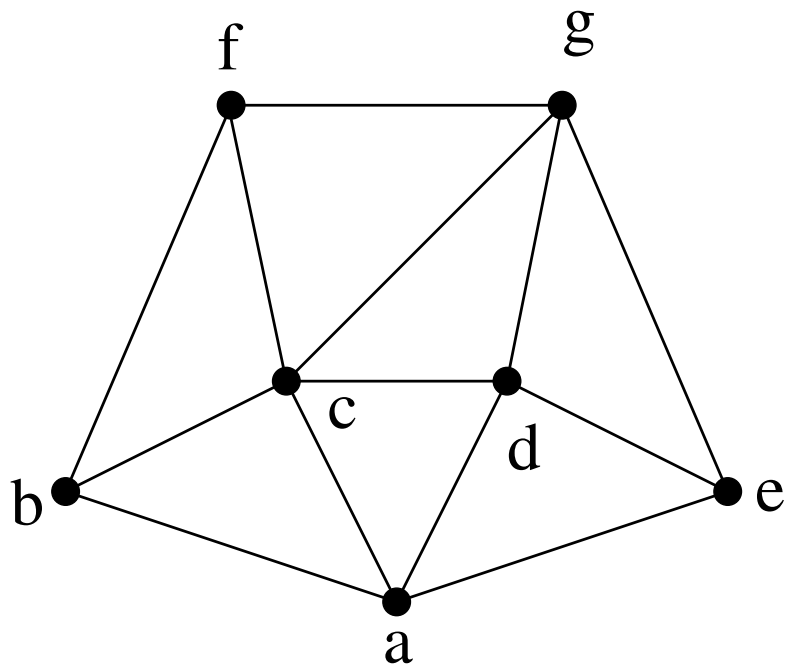
1. Tiré de l'excellent ouvrage *'Computer Science Unplugged'*

Un graphe



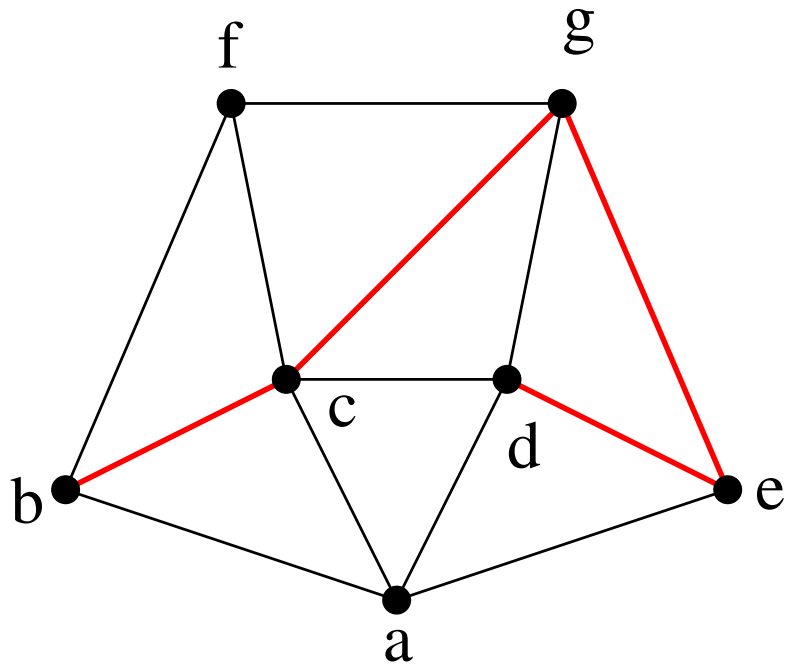
- Un **graphe** G est formé d'un ensemble V de **sommets** ($\{a, b, c, d, e, f, g\}$) et d'un ensemble E d'**arêtes** ($\{ab, ac, bc, bf, \ldots\}$).

Un graphe



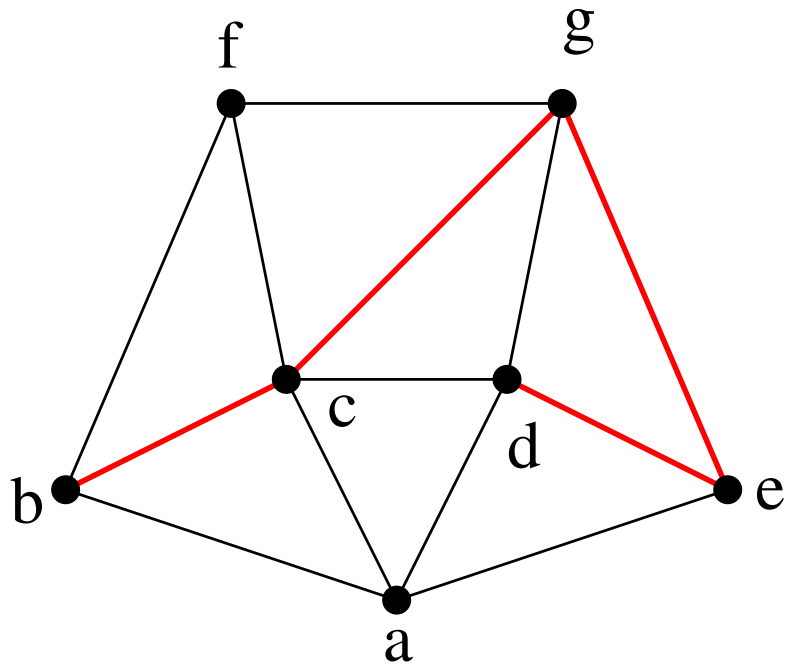
- Un **graphe** G est formé d'un ensemble V de **sommets** ($\{a, b, c, d, e, f, g\}$) et d'un ensemble E d'**arêtes** ($\{ab, ac, bc, bf, \ldots\}$).
- Classiquement, on note **n** le nombre de sommets du graphe et **m** son nombre d'arêtes.

Connexité



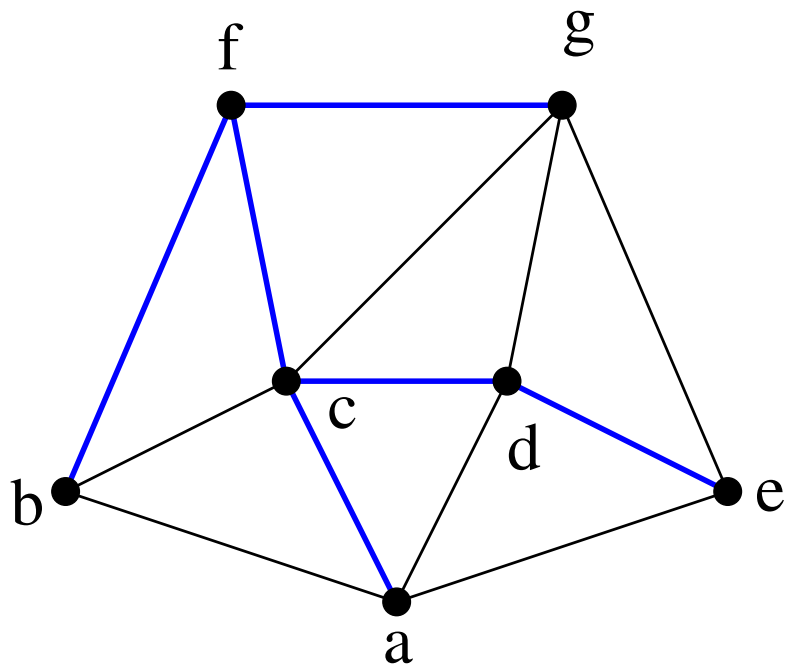
- Un **chemin** de G est une suite $P = (x_0, x_1, \ldots, x_p)$ de sommets *distincts* de G telle que $x_i x_{i+1}$ est une arête de G pour tout $i = 0, \ldots, p-1$.
- Si de plus $x_0 x_p$ est une arête de G alors on dit que P est un **cycle** de G.

Connexité



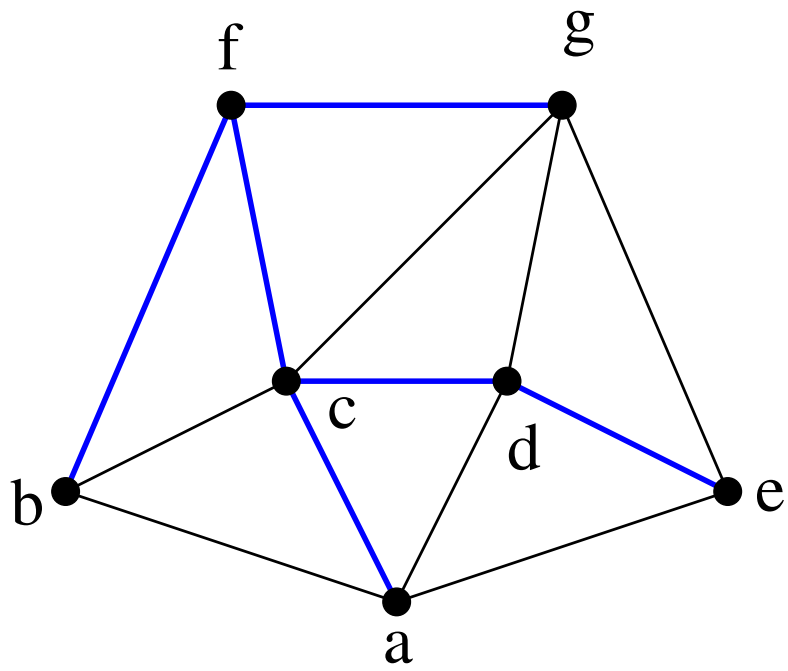
- Un **chemin** de G est une suite $P = (x_0, x_1, \ldots, x_p)$ de sommets *distincts* de G telle que $x_i x_{i+1}$ est une arête de G pour tout $i = 0, \ldots, p-1$.
- Si de plus $x_0 x_p$ est une arête de G alors on dit que P est un **cycle** de G.
- G est **connexe** si pour tous sommets u et v de G, il existe un chemin de G de u à v.

Arbre couvrant



- Un **arbre couvrant** T de G est un sous-graphe couvrant de G, connexe et sans cycle.

Arbre couvrant

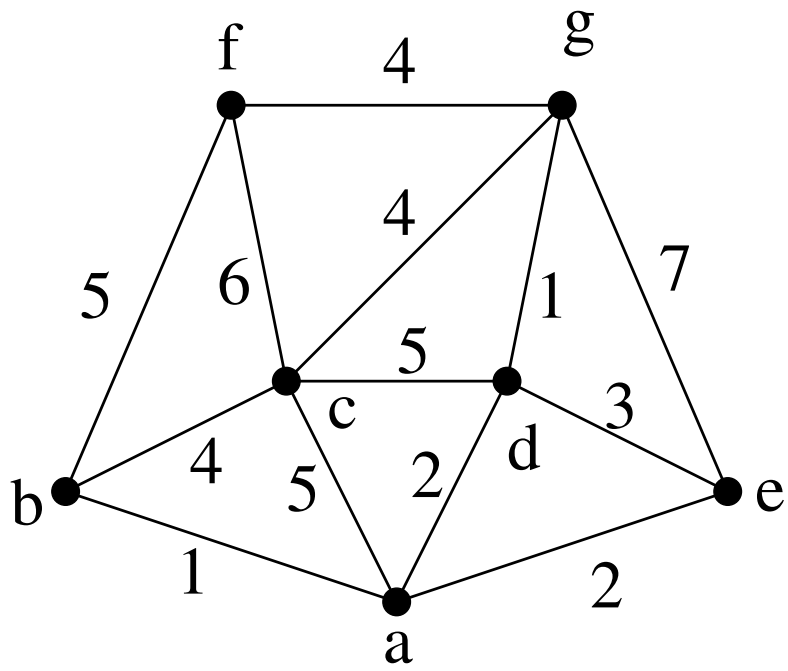


- Un **arbre couvrant** T de G est un sous-graphe couvrant de G, connexe et sans cycle.

Lemme

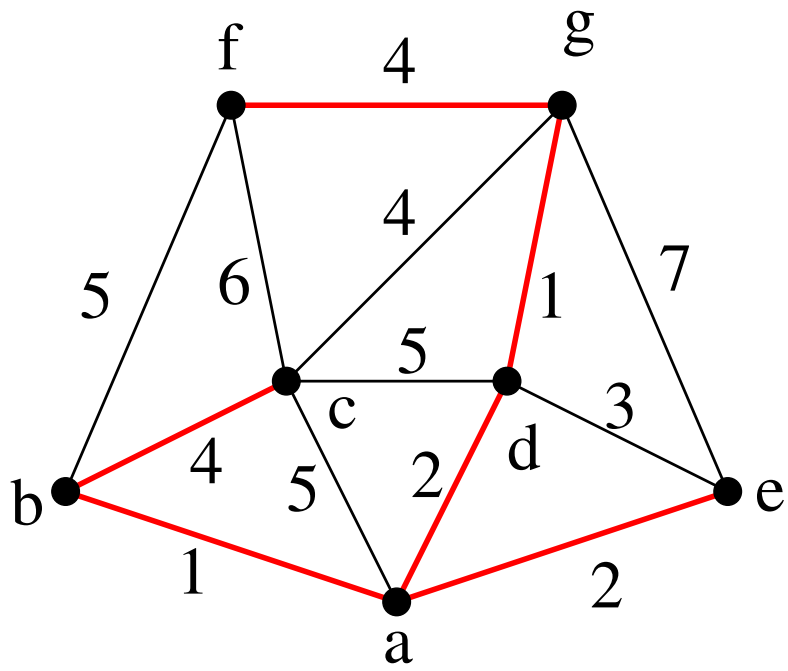
Un graphe est connexe si, et seulement si, il admet un arbre couvrant.

Graphe valué, ACPM



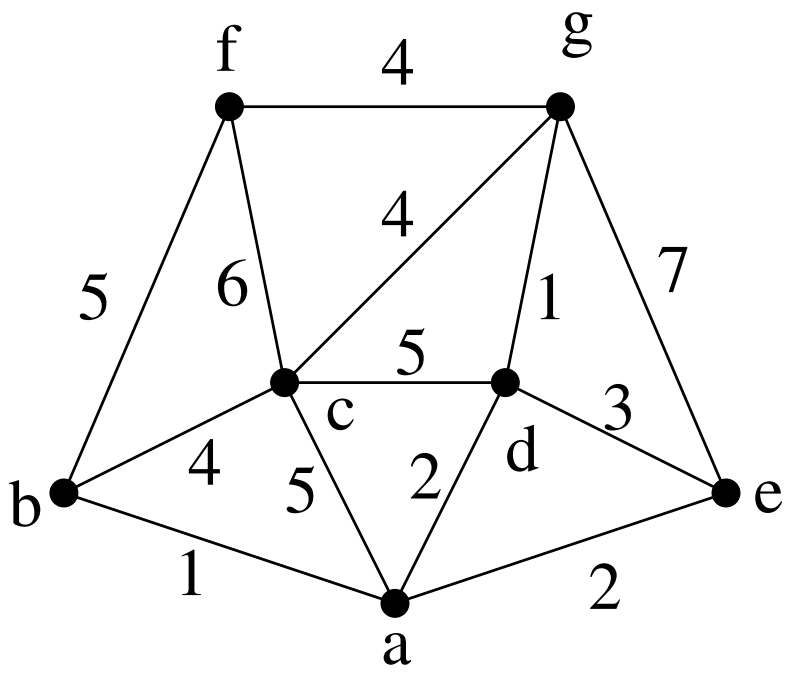
- Une **valuation** de G est l'ajout d'un poids positif (ou coût) sur chaque arête de G : c'est une fonction $p : E \to \mathbb{R}^+$

Graphe valué, ACPM



- Une **valuation** de G est l'ajout d'un poids positif (ou coût) sur chaque arête de G : c'est une fonction $p : E \to \mathbb{R}^+$
- Le **problème de l'arbre couvrant de poids minimum (ACPM)** consiste à trouver un arbre T de G avec un poids total minimum (c-à-d $\sum_{uv \text{ arête de } T} p(uv)$ minimum).

Algorithme de Kruskal

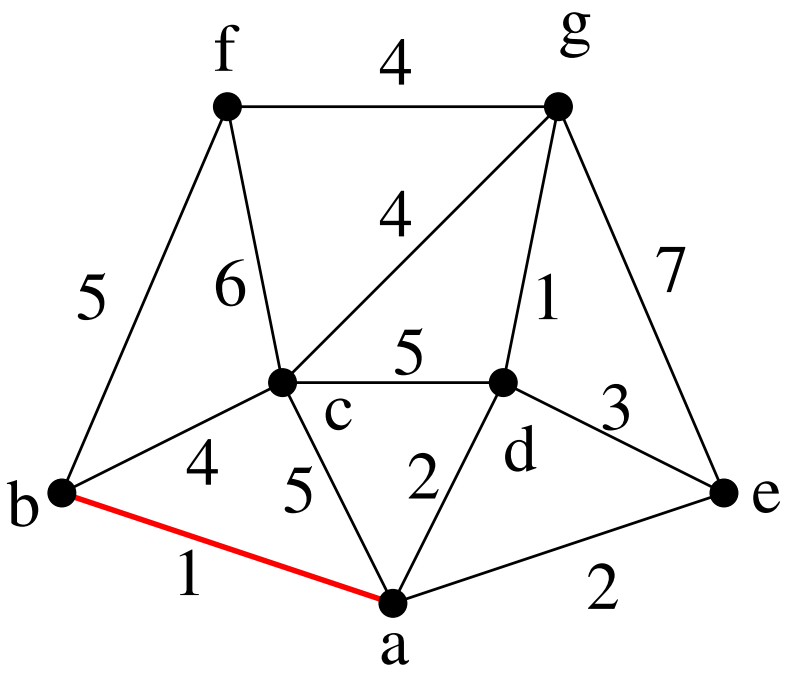


- **Algorithme de Kruskal** pour le du calcul d'un ACPM :

Algorithme : KRUSKAL(G, p)

Trier les arêtes de G par poids croissant selon p ;
$T \leftarrow \emptyset$;
pour *toute arête uv de G selon l'ordre calculé* **faire**
    si *uv ne forme pas un cycle avec les arêtes de T* **alors**
        $T \leftarrow T \cup \{uv\}$;
    Retourner T ;

Algorithme de Kruskal

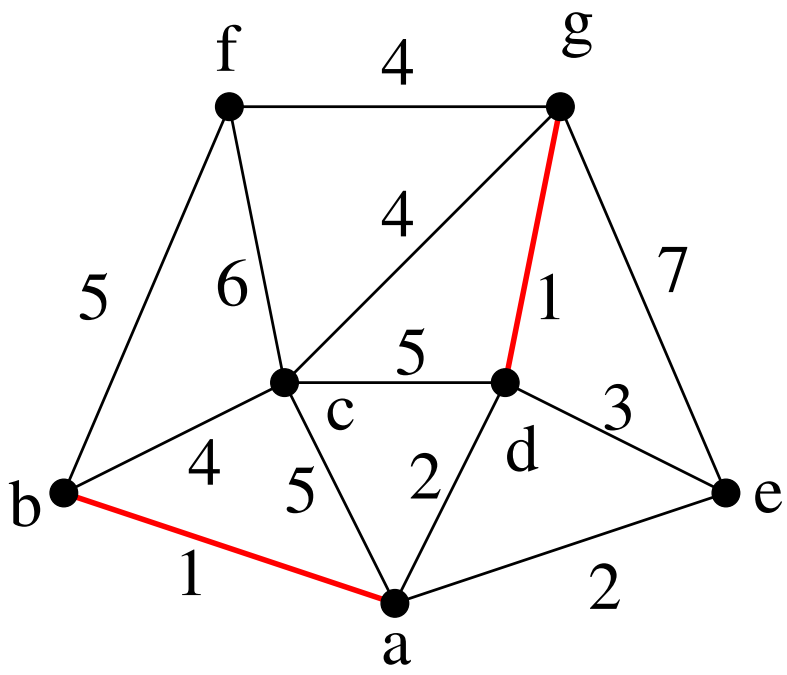


- **Algorithme de Kruskal** pour le du calcul d'un ACPM :

Algorithme : KRUSKAL(G, p)

Trier les arêtes de G par poids croissant selon p ;
$T \leftarrow \emptyset$;
pour *toute arête uv de G selon l'ordre calculé* **faire**
    si *uv ne forme pas un cycle avec les arêtes de T* **alors**
        $T \leftarrow T \cup \{uv\}$;
    Retourner T ;

Algorithme de Kruskal

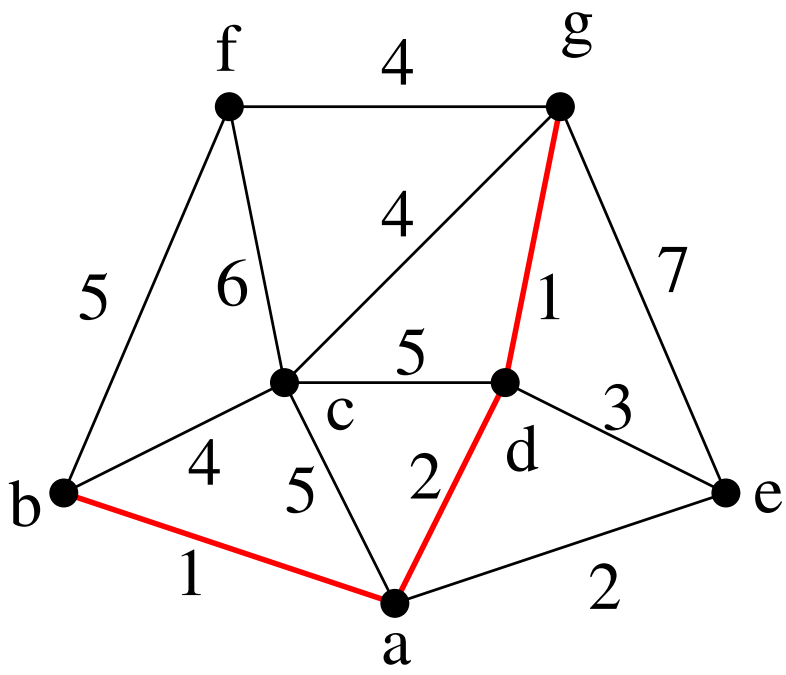


- **Algorithme de Kruskal** pour le du calcul d'un ACPM :

Algorithme : KRUSKAL(G, p)

Trier les arêtes de G par poids croissant selon p ;
$T \leftarrow \emptyset$;
pour *toute arête uv de G selon l'ordre calculé* **faire**
    si *uv ne forme pas un cycle avec les arêtes de T* **alors**
        $T \leftarrow T \cup \{uv\}$;
    Retourner T ;

Algorithme de Kruskal

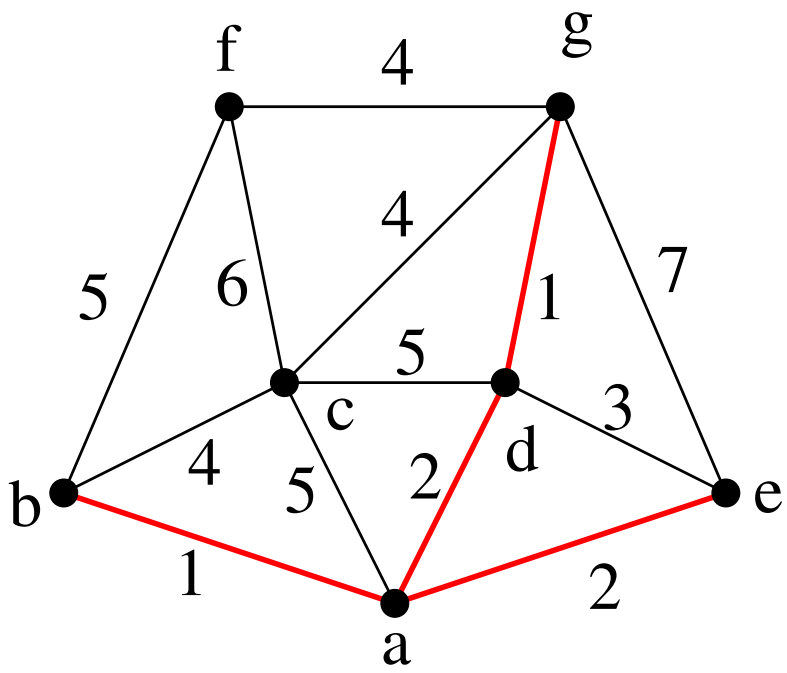


- **Algorithme de Kruskal** pour le du calcul d'un ACPM :

Algorithme : KRUSKAL(G, p)

Trier les arêtes de G par poids croissant selon p ;
$T \leftarrow \emptyset$;
pour *toute arête uv de G selon l'ordre calculé* **faire**
    si *uv ne forme pas un cycle avec les arêtes de T* **alors**
        $T \leftarrow T \cup \{uv\}$;
    Retourner T ;

Algorithme de Kruskal

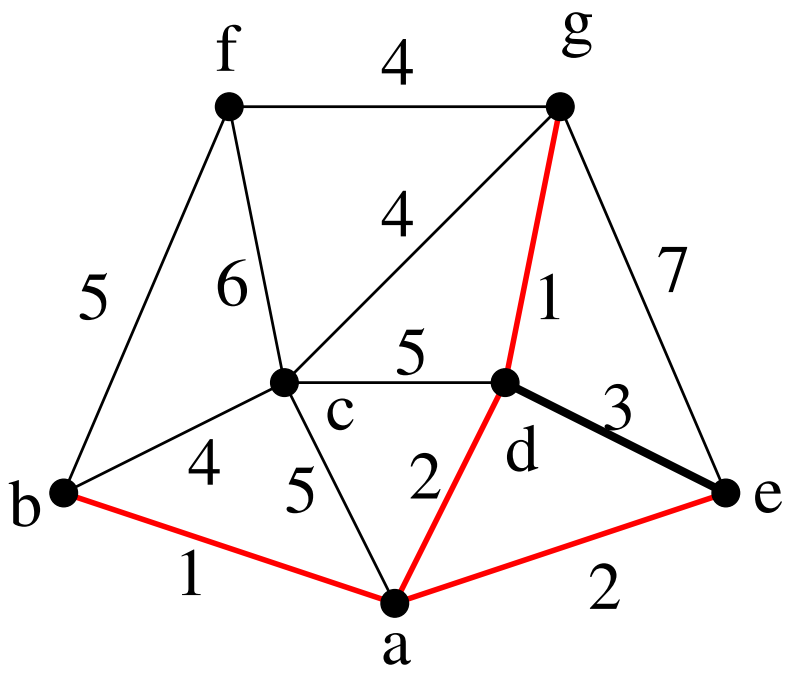


- **Algorithme de Kruskal** pour le du calcul d'un ACPM :

Algorithme : KRUSKAL(G, p)

Trier les arêtes de G par poids croissant selon p ;
$T \leftarrow \emptyset$;
pour *toute arête uv de G selon l'ordre calculé* **faire**
    si *uv ne forme pas un cycle avec les arêtes de T* **alors**
        $T \leftarrow T \cup \{uv\}$;
    Retourner T ;

Algorithme de Kruskal

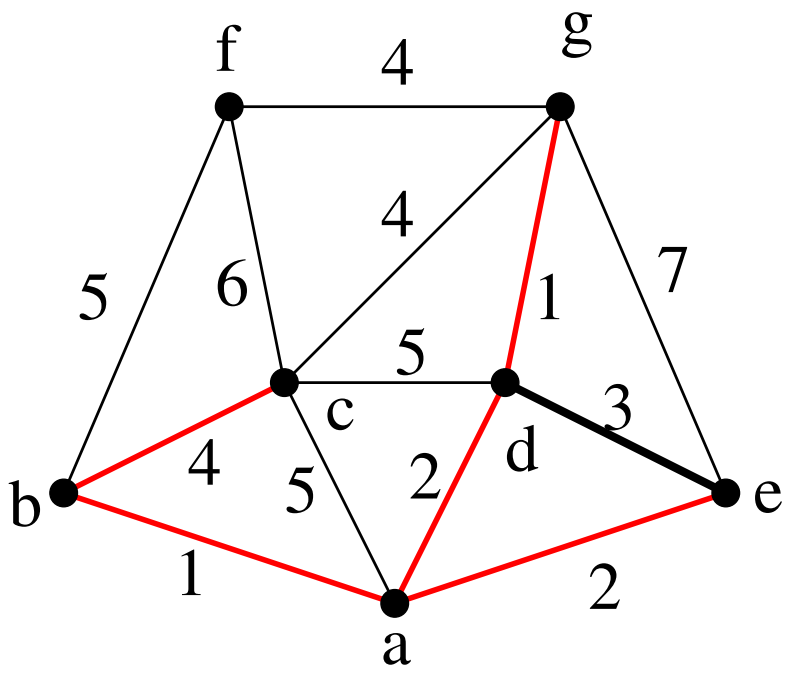


- **Algorithme de Kruskal** pour le du calcul d'un ACPM :

Algorithme : KRUSKAL(G, p)

Trier les arêtes de G par poids croissant selon p ;
$T \leftarrow \emptyset$;
pour *toute arête uv de G selon l'ordre calculé* **faire**
    si *uv ne forme pas un cycle avec les arêtes de T* **alors**
        $T \leftarrow T \cup \{uv\}$;
    Retourner T ;

Algorithme de Kruskal

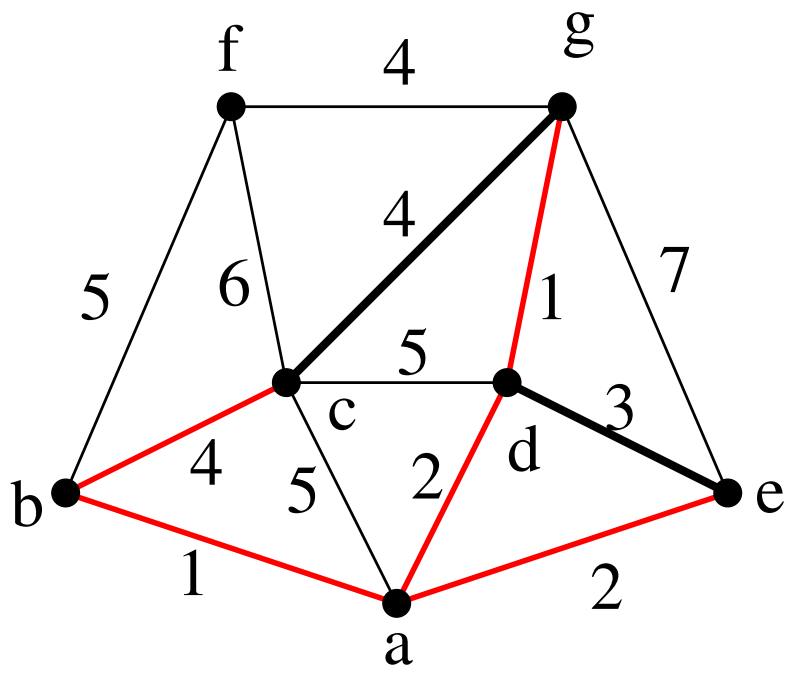


- **Algorithme de Kruskal** pour le du calcul d'un ACPM :

Algorithme : KRUSKAL(G, p)

Trier les arêtes de G par poids croissant selon p ;
$T \leftarrow \emptyset$;
pour *toute arête uv de G selon l'ordre calculé* **faire**
    si *uv ne forme pas un cycle avec les arêtes de T* **alors**
        $T \leftarrow T \cup \{uv\}$;
    Retourner T ;

Algorithme de Kruskal

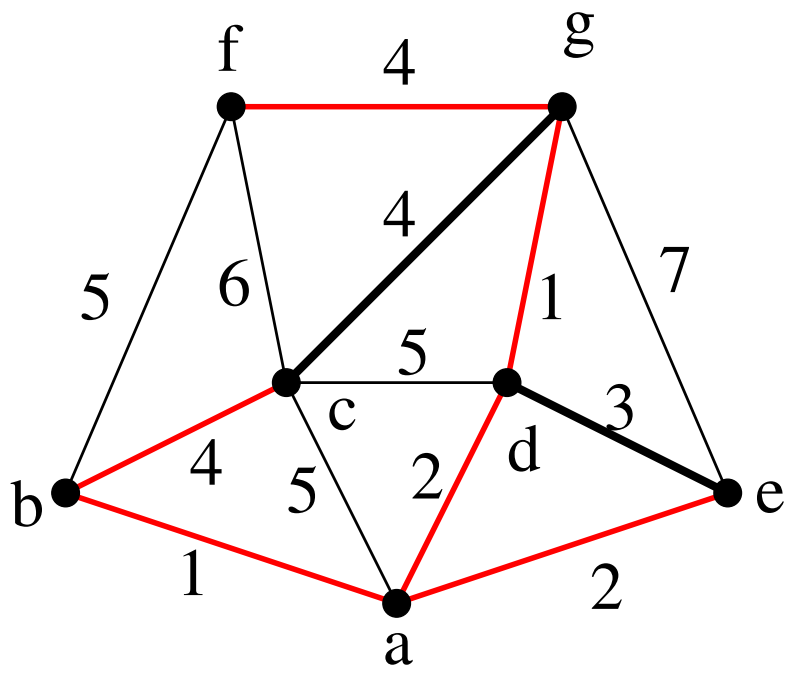


- **Algorithme de Kruskal** pour le du calcul d'un ACPM :

Algorithme : KRUSKAL(G, p)

Trier les arêtes de G par poids croissant selon p ;
$T \leftarrow \emptyset$;
pour *toute arête uv de G selon l'ordre calculé* **faire**
    si *uv ne forme pas un cycle avec les arêtes de T* **alors**
        $T \leftarrow T \cup \{uv\}$;
    Retourner T ;

Algorithme de Kruskal

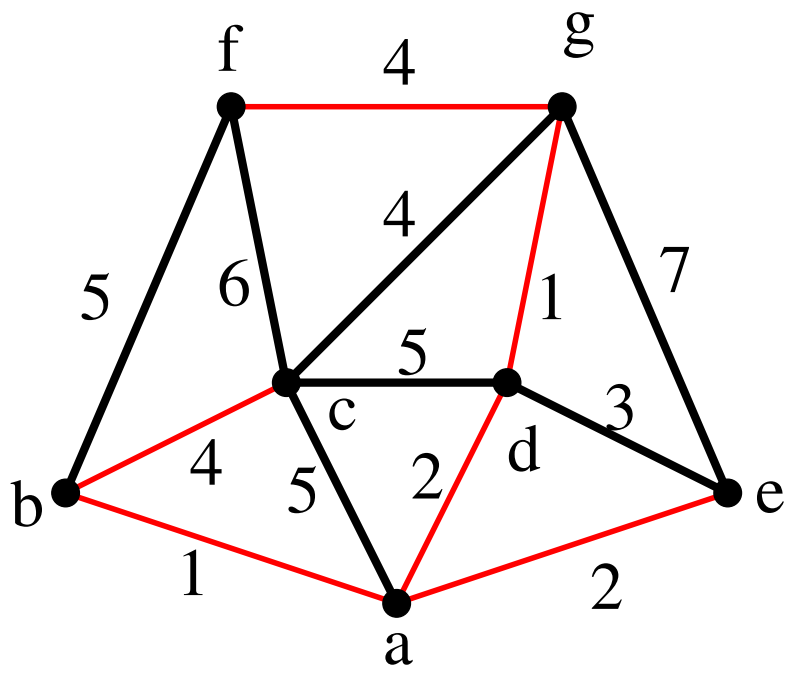


- **Algorithme de Kruskal** pour le du calcul d'un ACPM :

Algorithme : KRUSKAL(G, p)

Trier les arêtes de G par poids croissant selon p ;
$T \leftarrow \emptyset$;
pour *toute arête uv de G selon l'ordre calculé* **faire**
    si *uv ne forme pas un cycle avec les arêtes de T* **alors**
        $T \leftarrow T \cup \{uv\}$;
    Retourner T ;

Algorithme de Kruskal



- **Algorithme de Kruskal** pour le du calcul d'un ACPM :

Algorithme : KRUSKAL(G, p)

Trier les arêtes de G par poids croissant selon p ;
$T \leftarrow \emptyset$;
pour *toute arête uv de G selon l'ordre calculé* **faire**
    si *uv ne forme pas un cycle avec les arêtes de T* **alors**
        $T \leftarrow T \cup \{uv\}$;
    Retourner T ;

Algorithme de Kruskal

- C'est un algorithme **glouton** !
  À chaque étape on prend l'arête de poids minimum qui convient.

Algorithme de Kruskal

- C'est un algorithme **glouton** !
  À chaque étape on prend l'arête de poids minimum qui convient.
- Comment implémenter (un peu efficacement) la condition 'uv ne forme pas un cycle avec les arêtes de T' ?

Algorithme de Kruskal

- C'est un algorithme **glouton** !
  À chaque étape on prend l'arête de poids minimum qui convient.
- Comment implémenter (un peu efficacement) la condition 'uv ne forme pas un cycle avec les arêtes de T' ?

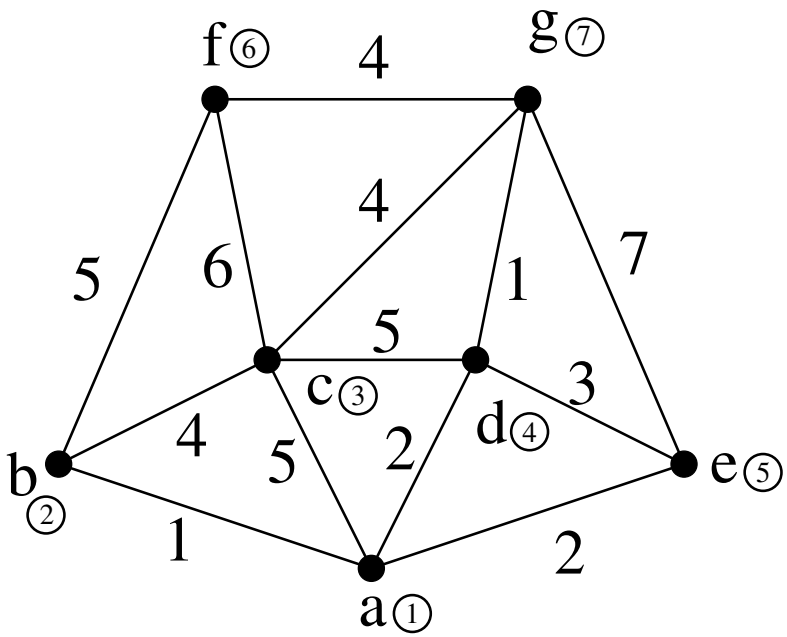
  On va attribuer à chaque sommet x de G un **numéro de composante** $comp(x)$, qui va vérifier $comp(x) = comp(y)$ au cours de l'algorithme ssi il existe un chemin dans T de x à y (ie. x et y sont dans la même composante connexe de T)

Algorithme de Kruskal

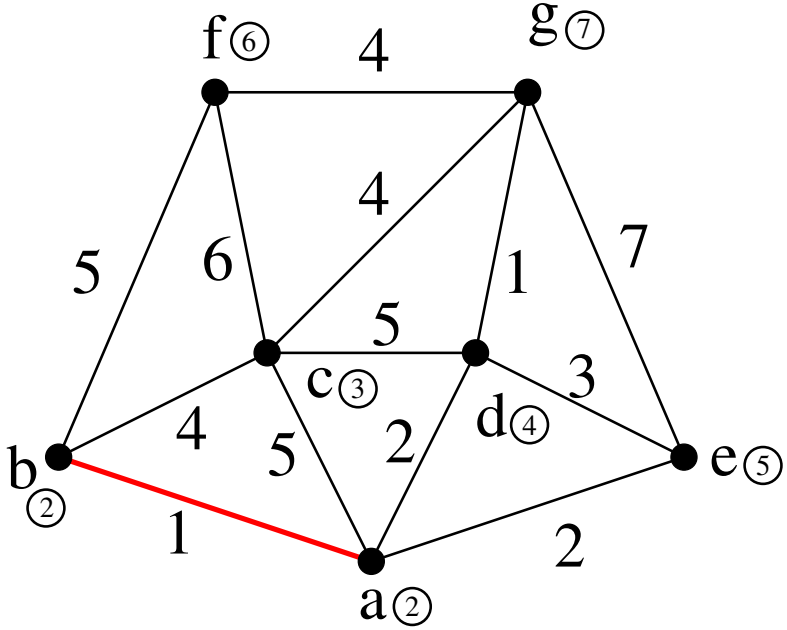
- C'est un algorithme **glouton** !
  À chaque étape on prend l'arête de poids minimum qui convient.
- Comment implémenter (un peu efficacement) la condition 'uv ne forme pas un cycle avec les arêtes de T' ?

  On va attribuer à chaque sommet x de G un **numéro de composante** $comp(x)$, qui va vérifier $comp(x) = comp(y)$ au cours de l'algorithme ssi il existe un chemin dans T de x à y (ie. x et y sont dans la même composante connexe de T)
  - Au début, chaque sommet de G reçoit un numéro de composante différent
  - Lors de l'examen de l'arête uv :
    - Si $comp(u) = comp(v)$, on fait rien, u et v sont déjà reliés par un chemin dans T, si on ajoute uv, on va créer un cycle !
    - Si $comp(u) \neq comp(v)$, on ajoute uv à T et on met à jour tous les sommets de numéro $comp(u)$ en leur attribuant un nouveau numéro de composante valant $comp(v)$.

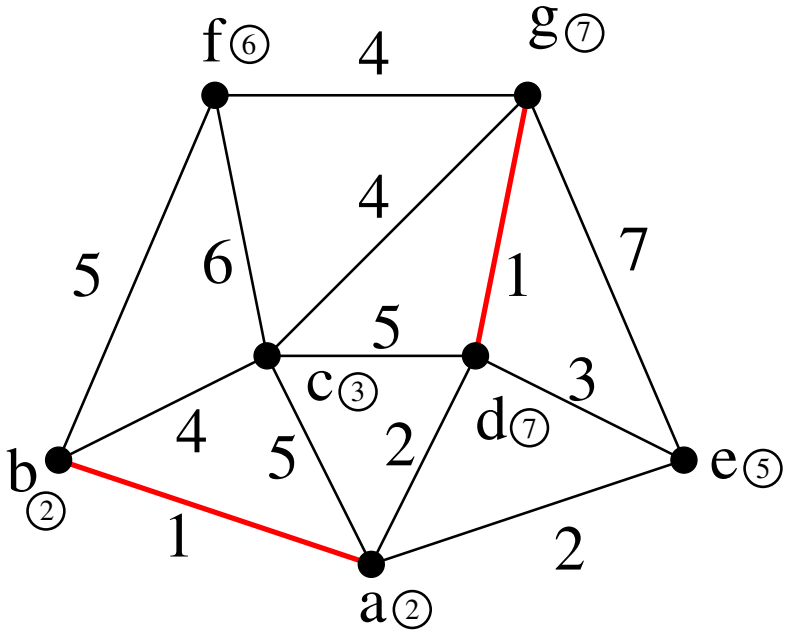
Algorithme de Kruskal



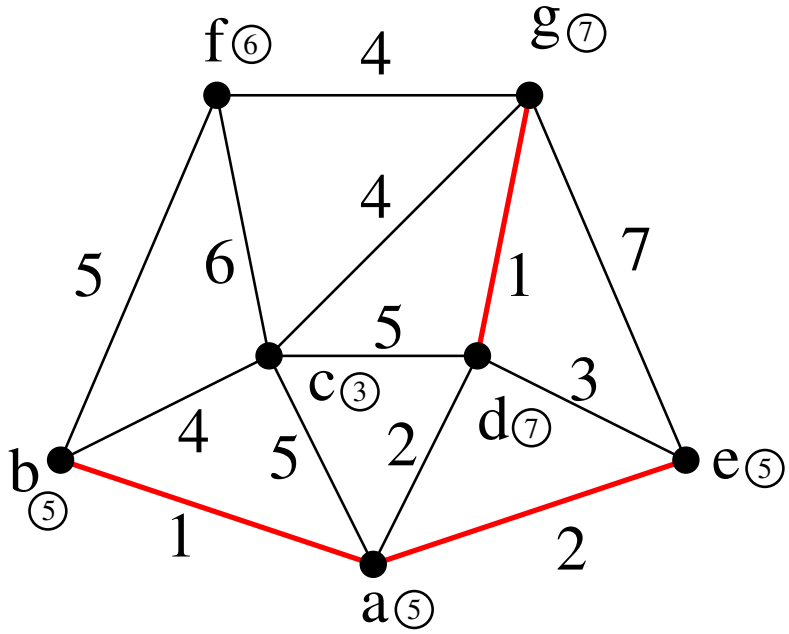
Algorithme de Kruskal



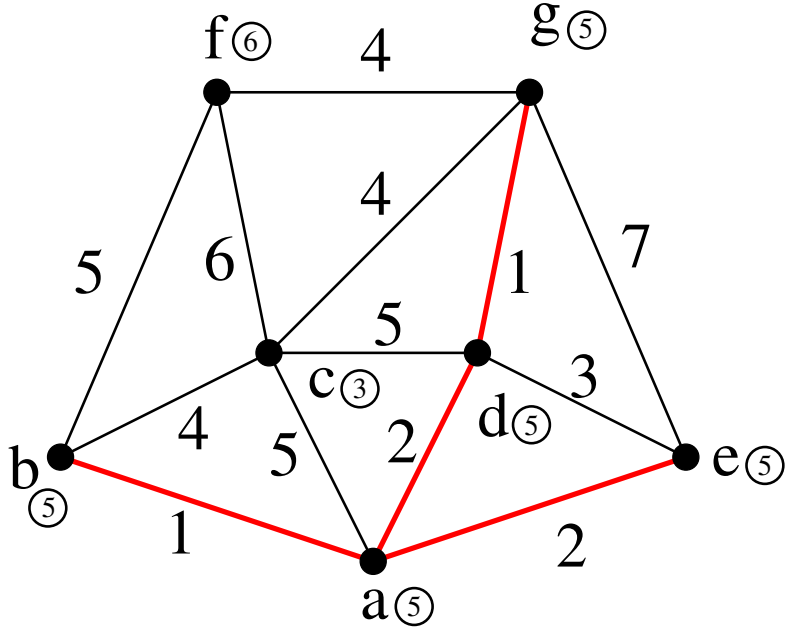
Algorithme de Kruskal



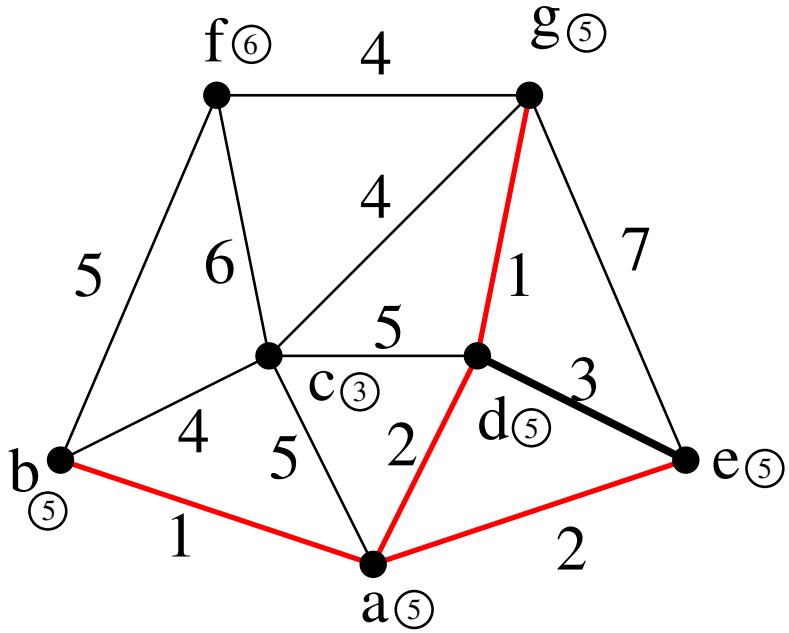
Algorithme de Kruskal



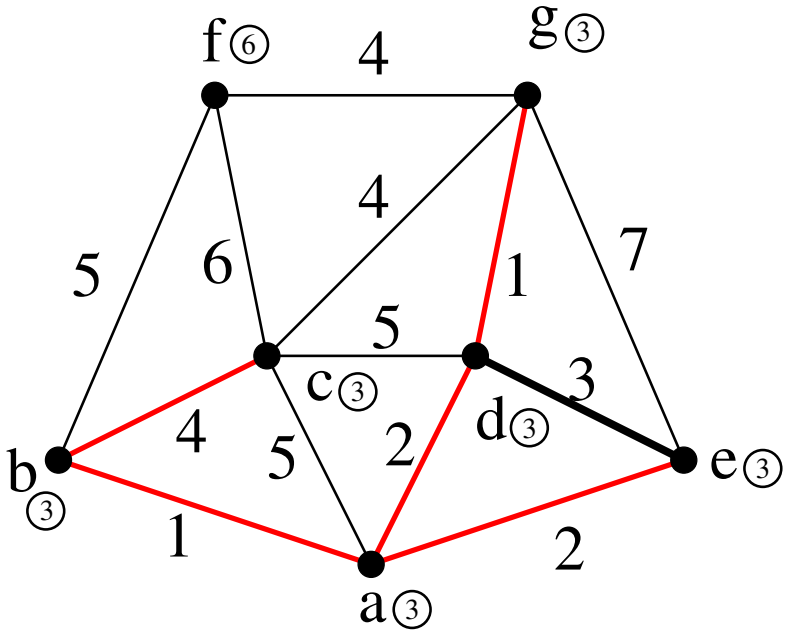
Algorithme de Kruskal



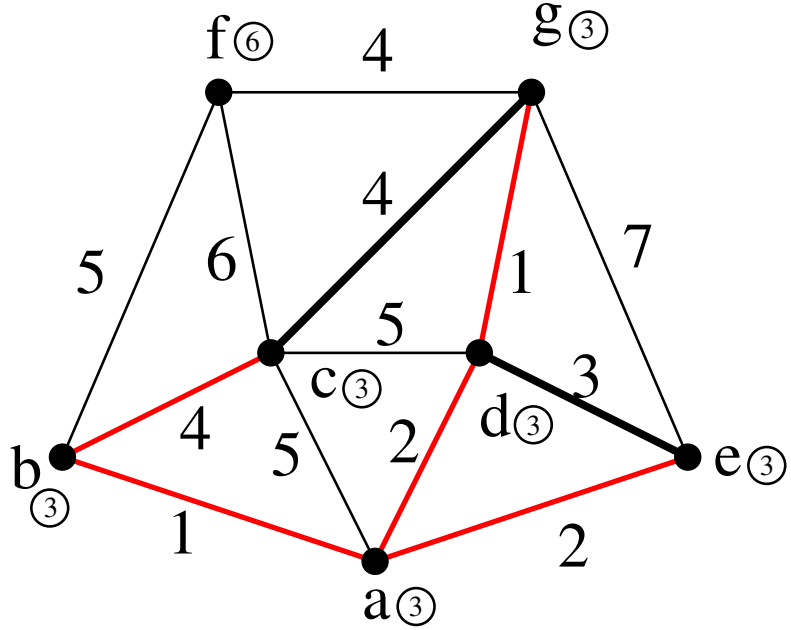
Algorithme de Kruskal



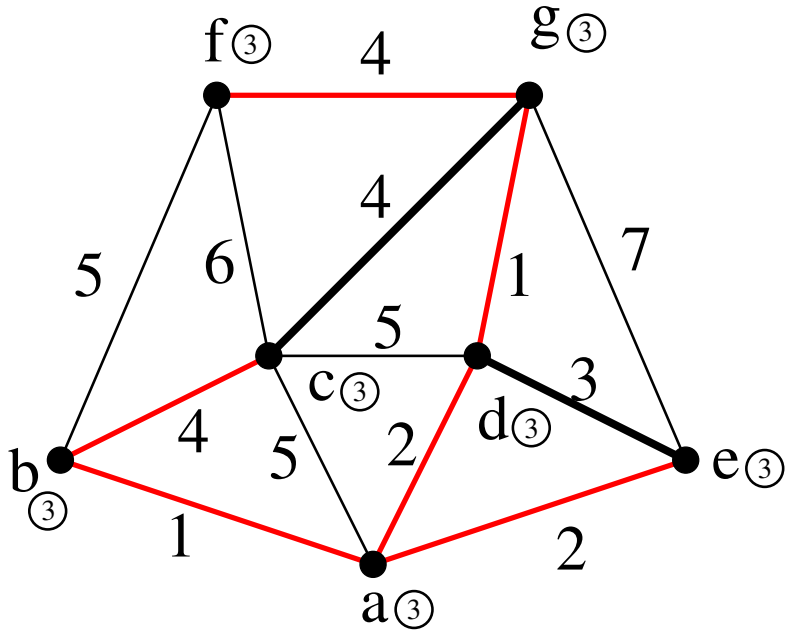
Algorithme de Kruskal



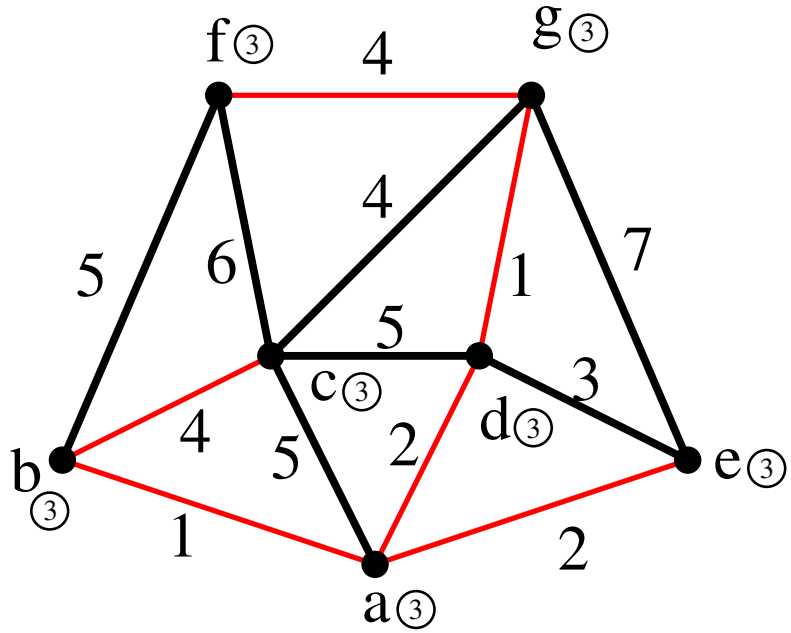
Algorithme de Kruskal



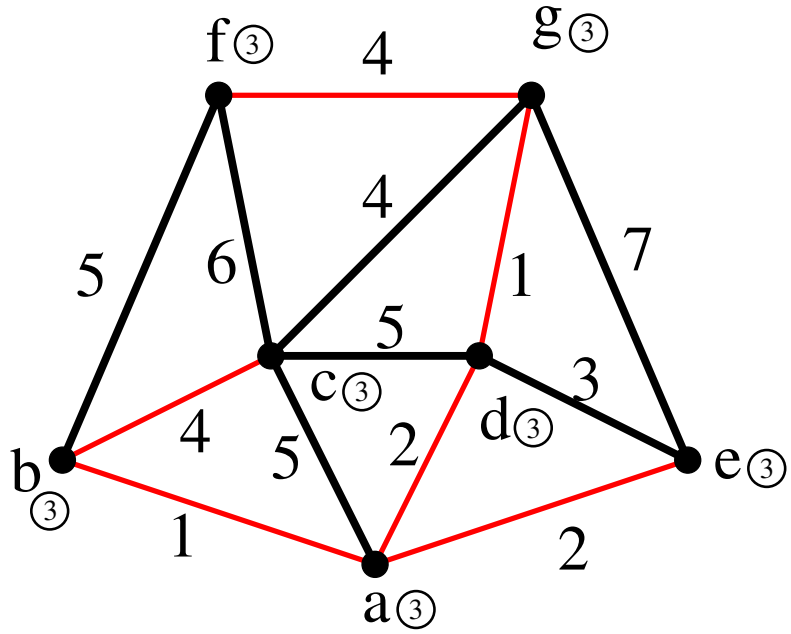
Algorithme de Kruskal



Algorithme de Kruskal



Algorithme de Kruskal



Remarque :

- A la fin de l'algorithme, tous les sommets ont le même numéro de composante...
- Une implémentation possible :

Algorithme de Kruskal

Algorithme : KRUSKAL(G, p)

Trier les arêtes de G par poids croissant selon p ;
$T \leftarrow \emptyset$;
$i \leftarrow 1$;
pour $x \in V$ **faire** $comp(x) \longleftarrow i$; $i \leftarrow i+1$;
pour *toute arête* uv *de* G *selon l'ordre calculé* **faire**
    si $comp(u) \neq comp(v)$ **alors**
        $T \leftarrow T \cup \{uv\}$;
        $aux \leftarrow comp(u)$;
        pour tous les $w \in V$ **faire**
            si $comp(w) = aux$ **alors** $comp(w) \leftarrow comp(v)$
retourner T ;

Algorithme de Kruskal

Algorithme : KRUSKAL(G, p)

Trier les arêtes de G par poids croissant selon p ;
$T \leftarrow \emptyset$;
$i \leftarrow 1$;
pour $x \in V$ **faire** $comp(x) \longleftarrow i$; $i \leftarrow i+1$;
pour *toute arête* uv *de* G *selon l'ordre calculé* **faire**
    si $comp(u) \neq comp(v)$ **alors**
        $T \leftarrow T \cup \{uv\}$;
        $aux \leftarrow comp(u)$;
        pour tous les $w \in V$ **faire**
            si $comp(w) = aux$ **alors** $comp(w) \leftarrow comp(v)$
retourner T ;

Remarques :

- Si on n'utilise pas la variable aux, au moment où l'algo va traiter $w = u$, il va changer la valeur de $comp(u)$ en $comp(v)$ et la suite de la boucle '**pour tous les** $w \in V$ **faire**' ne va plus renommer les numéros des sommets de numéros de composante $comp(u)$...

Algorithme de Kruskal

Algorithme : KRUSKAL(G, p)

Trier les arêtes de G par poids croissant selon p ;
$T \leftarrow \emptyset$;
$i \leftarrow 1$;
pour $x \in V$ **faire** $comp(x) \longleftarrow i$; $i \leftarrow i+1$;
pour *toute arête* uv *de* G *selon l'ordre calculé* **faire**
    si $comp(u) \neq comp(v)$ **alors**
        $T \leftarrow T \cup \{uv\}$;
        $aux \leftarrow comp(u)$;
        pour tous les $w \in V$ **faire**
            si $comp(w) = aux$ **alors** $comp(w) \leftarrow comp(v)$
retourner T ;

Remarques :

- le tri prend un temps $O(m.\log m)$, mais dans un graphe (sans arête parallèle ni boucle...), on a $m \leq \frac{n(n-1)}{2}$ (pourquoi ?) d'où $m \leq n^2$ et $O(m.\log m) = O(m.\log n)$

Algorithme de Kruskal

Algorithme : KRUSKAL(G, p)

Trier les arêtes de G par poids croissant selon p ;
$T \leftarrow \emptyset$;
$i \leftarrow 1$;
pour $x \in V$ **faire** $comp(x) \longleftarrow i$; $i \leftarrow i+1$;
pour *toute arête* uv *de* G *selon l'ordre calculé* **faire**
    si $comp(u) \neq comp(v)$ **alors**
        $T \leftarrow T \cup \{uv\}$;
        $aux \leftarrow comp(u)$;
        pour tous les $w \in V$ **faire**
            si $comp(w) = aux$ **alors** $comp(w) \leftarrow comp(v)$
retourner T ;

Théorème

KRUSKAL *calcule un ACPM de* G *et a une complexité en* $O(n.m)$

Algorithme de Kruskal

Algorithme : KRUSKAL(G, p)

Trier les arêtes de G par poids croissant selon p ;
$T \leftarrow \emptyset$;
$i \leftarrow 1$;
pour $x \in V$ **faire** $comp(x) \longleftarrow i$; $i \leftarrow i+1$;
pour *toute arête* uv *de* G *selon l'ordre calculé* **faire**
    si $comp(u) \neq comp(v)$ **alors**
        $T \leftarrow T \cup \{uv\}$;
        $aux \leftarrow comp(u)$;
        pour tous les $w \in V$ **faire**
            si $comp(w) = aux$ **alors** $comp(w) \leftarrow comp(v)$
retourner T ;

Remarques :

- En calculant plus finement on arrive à une complexité en $O(n^2 + m \log n)$
- Et avec une optimisation classique, on arrive à $O(m + n \log n)$
- Obtenir une complexité en $O(m+n)$ pour le calcul d'un ACPM est un problème ouvert !

1. Borne inférieure pour le tri

2. Arbre couvrant de poids minimum

3. Algorithme des k plus proches voisins

4. Tour de Hanoï

Algorithme des k plus proches voisins

- Algorithme d'apprentissage 'supervisé', assez simple.
- On dispose d'un ensemble $\mathscr{P}$ de n points $P_1, \ldots, P_n$ du plan (ou de l'espace..), qui sont **classifiés**, c'est-à-dire que chaque P_i appartient à une classe $cl(P_i)$.
  Il y a p classes (avec $p << n$ souvent). Les classes sont des valeurs ou des couleurs typiquement.

Algorithme des k plus proches voisins

- Algorithme d'apprentissage 'supervisé', assez simple.
- On dispose d'un ensemble $\mathscr{P}$ de n points $P_1, \ldots, P_n$ du plan (ou de l'espace..), qui sont **classifiés**, c'est-à-dire que chaque P_i appartient à une classe $cl(P_i)$.
  Il y a p classes (avec $p << n$ souvent). Les classes sont des valeurs ou des couleurs typiquement.
- Le problème consiste **à classifier un nouveau point** M (sous la 'supervision' de $P_1, \ldots, P_n$).

Algorithme des k plus proches voisins

- Algorithme d'apprentissage 'supervisé', assez simple.
- On dispose d'un ensemble $\mathscr{P}$ de n points $P_1, \ldots, P_n$ du plan (ou de l'espace..), qui sont **classifiés**, c'est-à-dire que chaque P_i appartient à une classe $cl(P_i)$.
  Il y a p classes (avec $p << n$ souvent). Les classes sont des valeurs ou des couleurs typiquement.
- Le problème consiste **à classifier un nouveau point** M (sous la 'supervision' de $P_1, \ldots, P_n$).
- Pour cela, l'algorithme des k **plus proches voisins** (ou k-**NN**) détermine les k points de $\mathscr{P}$ les plus proches de M, calcule la classe majoritaire dans cet ensemble de points et l'attribue à M.

k-NN, Exemple de classification

- Les iris se repartissent en trois sous-espèces : les *setosa*, les *virginica* et les *versicolor* :

iris setosa

iris virginica

iris versicolor

k-NN, Exemple de classification

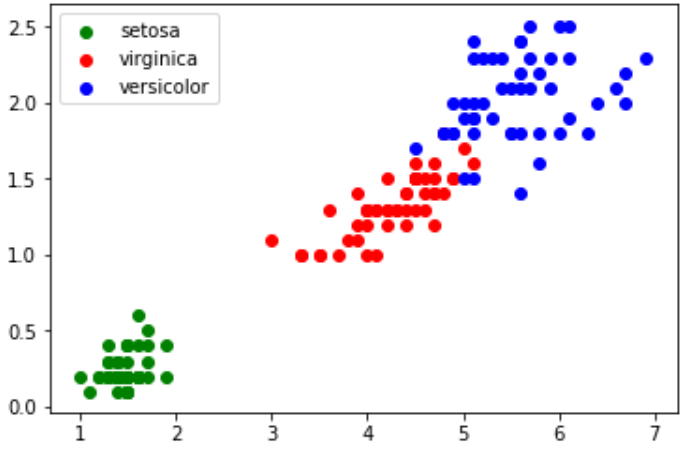
- Les iris se repartissent en trois sous-espèces : les *setosa*, les *virginica* et les *versicolor* :

iris setosa

iris virginica

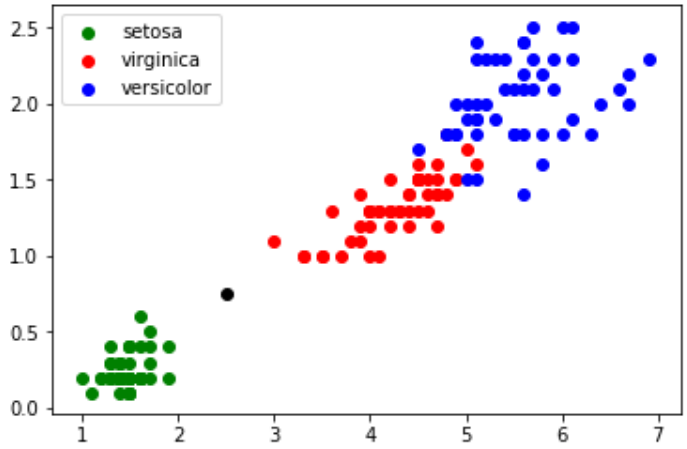
iris versicolor

- On classe un échantillon d'iris en fonction de la longueur et largeur de leurs pétales :



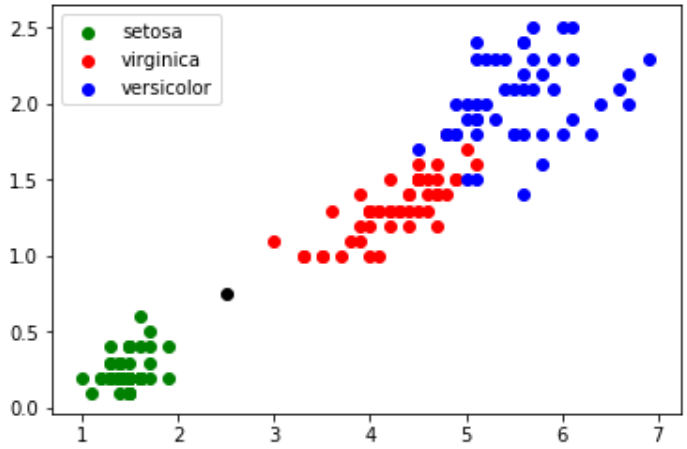
k-NN, Exemple de classification

- On trouve un nouvel iris en se promenant (le point noir) :

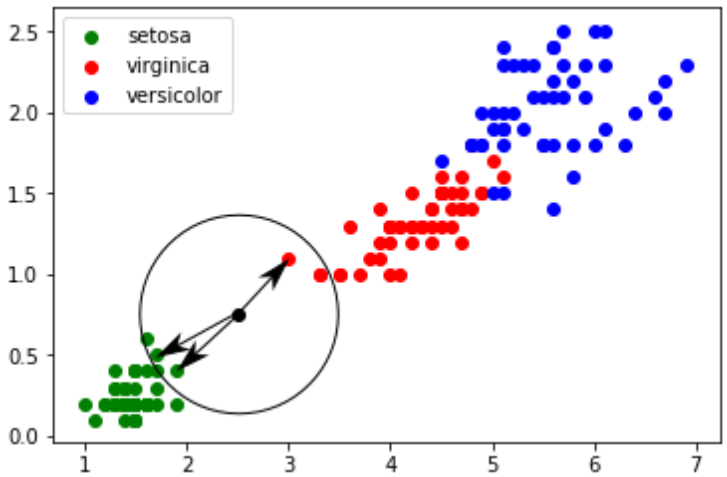


k-NN, Exemple de classification

- On trouve un nouvel iris en se promenant (le point noir) :

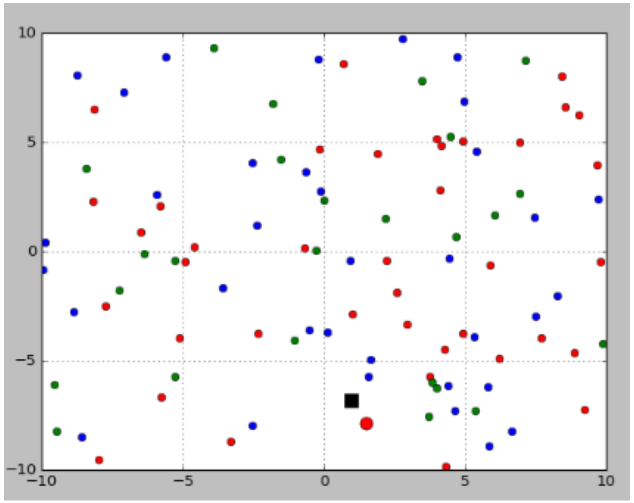


- On le classifie avec $k = 5$: c'est un setosa !



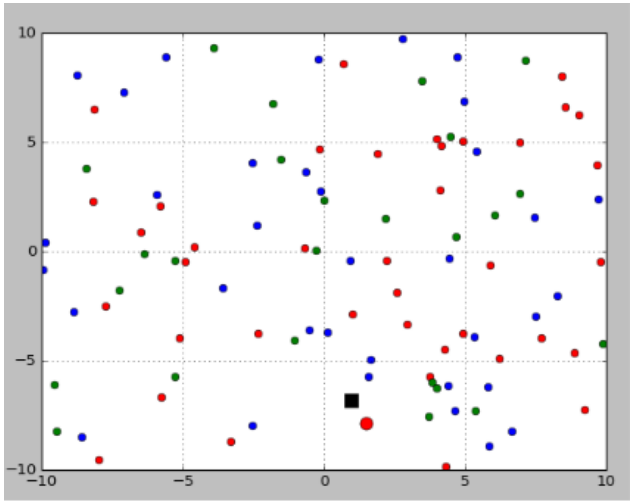
k-NN, Exemple de clusterisation

- On a toujours un ensemble de n point du plan classifiés

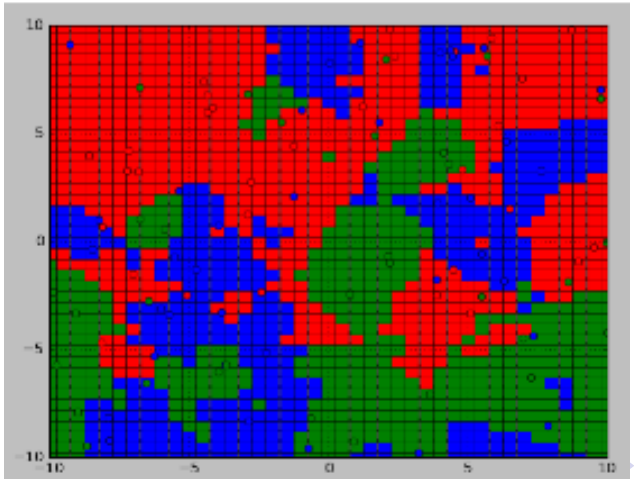


k-NN, Exemple de clusterisation

- On a toujours un ensemble de n point du plan classifiés

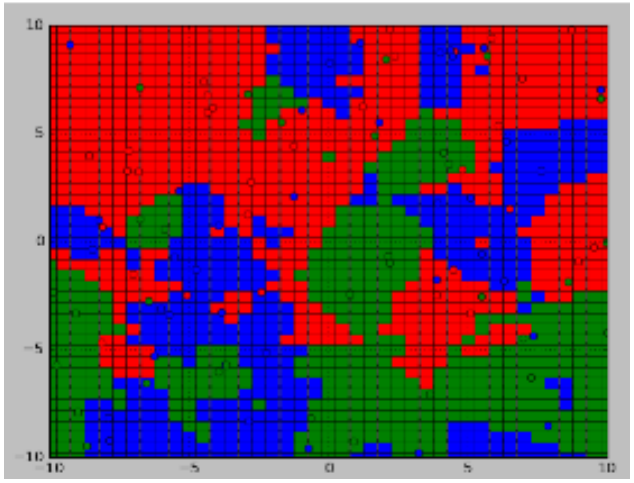


- Cette fois, on vaut classifier tous les points du plan non classifiés, en les prenant un à un, au hasard et en appliquant k-NN.



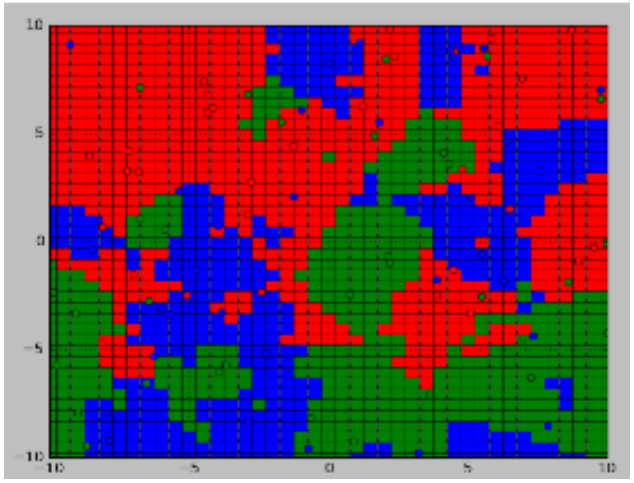
k-NN, Exemple de clusterisation

- Apparaissent alors des régions du plan connexes de points de la même classe (des 'clusters').

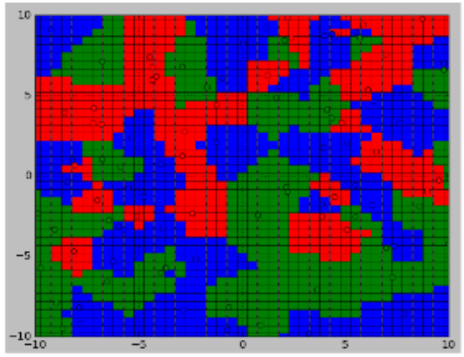
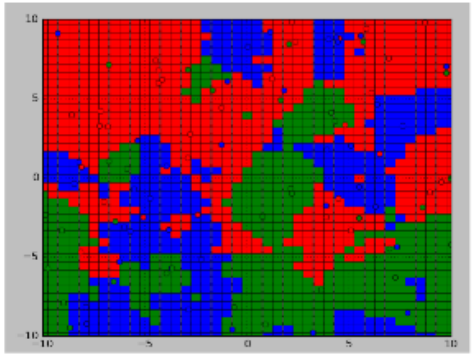
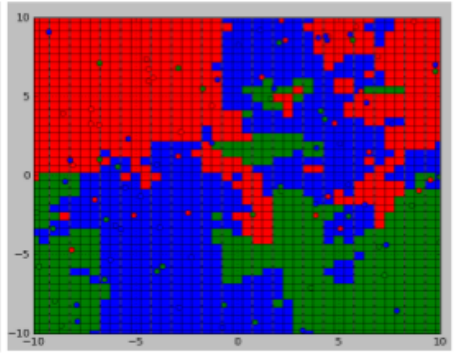


k-NN, Exemple de clusterisation

- Apparaissent alors des régions du plan connexes de points de la même classe (des 'clusters').



- En faisant varier k, la forme des clusters se modifie. Ici, pour $k = 1$, $k = 3$ et $k = 10$ respectivement

1. Borne inférieure pour le tri

2. Arbre couvrant de poids minimum

3. Algorithme des k plus proches voisins

4. Tour de Hanoï

Le jeu de la tour de Hanoï

- Le jeu est constitué de trois tours (tiges, nommées A, B et C), et de n palets de tailles différentes qui se place sur les tiges. En position de départ, tous les palets sont placés sur la tour A, comme indiquée ci-dessous :

Le jeu de la tour de Hanoï

- Le jeu est constitué de trois tours (tiges, nommées A, B et C), et de n palets de tailles différentes qui se place sur les tiges. En position de départ, tous les palets sont placés sur la tour A, comme indiquée ci-dessous :

- Le but est de faire passer tous les palets sur la tour C en respectant les règles :
  - On ne déplace qu'un palet à la fois.
  - Un palet ne peut s'empiler que sur un palet de diamètre supérieur (ou sur une tour vide).

Résolution de la tour de Hanoï

- On va utiliser un **algorithme récursif** (mais pas du Diviser pour Régner...), pour écrire les coups menant à une résolution du problème.

Résolution de la tour de Hanoï

- On va utiliser un **algorithme récursif** (mais pas du Diviser pour Régner...), pour écrire les coups menant à une résolution du problème.
- L'ensemble $\{X, Y, Z\}$ désigne l'ensemble des trois tours ($\{X, Y, Z\} = \{A, B, C\}$).
  La fonction récursive HANOI(p, X, Y) permet de déplacer p disques de diamètres consécutifs de la tour X à la tour Y :

Algorithme : HANOI(p, X, Y)

si $p = 1$ **alors**
  Afficher `'X->Y'` ;
sinon
  HANOI$(p-1, X, Z)$;
  Afficher `'X->Y'` ;
  HANOI$(p-1, Z, Y)$;

Résolution de la tour de Hanoï

- On va utiliser un **algorithme récursif** (mais pas du Diviser pour Régner...), pour écrire les coups menant à une résolution du problème.
- L'ensemble $\{X, Y, Z\}$ désigne l'ensemble des trois tours ($\{X, Y, Z\} = \{A, B, C\}$).
  La fonction récursive HANOI(p, X, Y) permet de déplacer p disques de diamètres consécutifs de la tour X à la tour Y :

Algorithme : HANOI(p, X, Y)

si $p = 1$ **alors**
  Afficher `'X->Y'` ;
sinon
  HANOI$(p-1, X, Z)$;
  Afficher `'X->Y'` ;
  HANOI$(p-1, Z, Y)$;

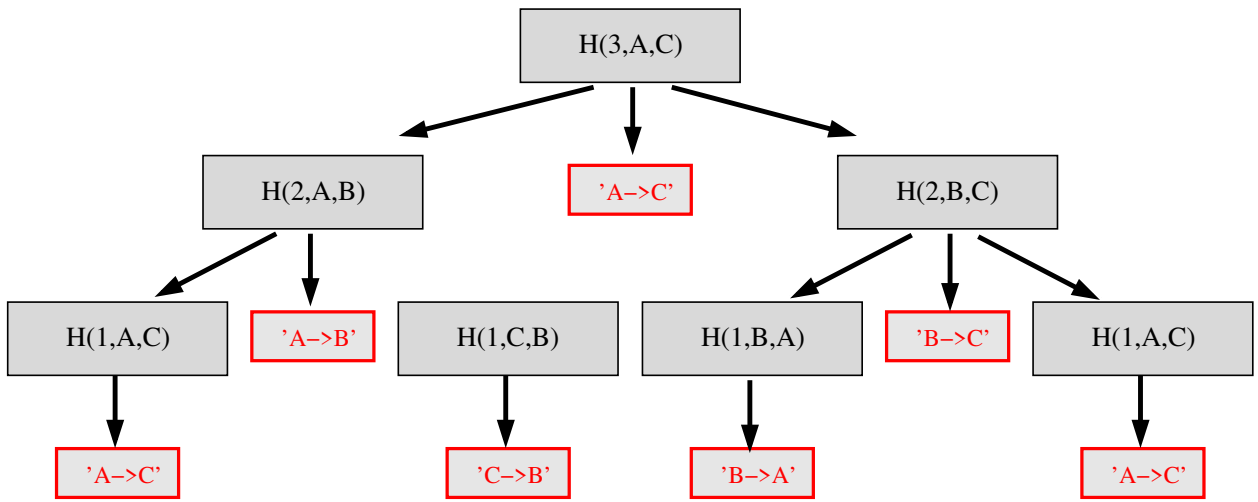
- L'appel initial est HANOI(n, A, C).

Résolution de la tour de Hanoï

Algorithme : HANOI(p, X, Y)

si $p = 1$ **alors**
  Afficher `'X->Y'` ;
sinon
  HANOI$(p-1, X, Z)$;
  Afficher `'X->Y'` ;
  HANOI$(p-1, Z, Y)$;

- Par exemple, pour HANOI$(3, A, C)$ on obtient :



`'A->C','A->B','C->B','A->C','B->A','B->C','A->C'`

Analyse de HANOI

Algorithme : HANOI(p, X, Y)

si $p = 1$ **alors**
  Afficher `'X->Y'` ;
sinon
  HANOI$(p-1, X, Z)$;
  Afficher `'X->Y'` ;
  HANOI$(p-1, Z, Y)$;

On a le résultat suivant :

Théorème

HANOI(n, A, C) *résout le problème en* $2^n - 1$ *coups.*

Preuve par récurrence ! Voir en TD...