Algorithmique

Vlady Ravelomanana

IRIF – UMR CNRS 8243 - Paris 7
vlad@irif.fr

Master 1

Généralités sur les algorithmes de parcours

Parcourir un graphe, c'est **visiter** (et traiter) un par un ses sommets. Les **parcours d'arbres** (voir transparents n°2 et n°3) sont des cas particuliers de parcours de graphes.

Généralités sur les algorithmes de parcours

Parcourir un graphe, c'est **visiter** (et traiter) un par un ses sommets. Les **parcours d'arbres** (voir transparents n°2 et n°3) sont des cas particuliers de parcours de graphes.

Les **parcours de graphes** sont des procédés qui permettent de choisir, à partir des sommets déà visités, le sommet suivant à visiter. Le problème consiste donc à établir un ordre sur les visites des sommets.

Généralités sur les algorithmes de parcours

Parcourir un graphe, c'est **visiter** (et traiter) un par un ses sommets. Les **parcours d'arbres** (voir transparents n°2 et n°3) sont des cas particuliers de parcours de graphes.

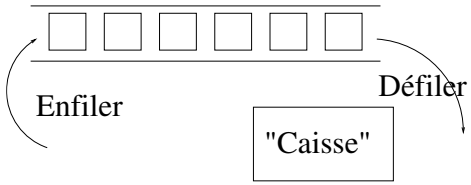
Les **parcours de graphes** sont des procédés qui permettent de choisir, à partir des sommets déà visités, le sommet suivant à visiter. Le problème consiste donc à établir un ordre sur les visites des sommets.

Comme pour les arbres, nous allons avoir besoin de structures de **file** et de **pile** pour nos parcours.

Rappels : File

Rappel File

- Une file (FIFO : first in first out, premier entré/premier servi) peut être implementée en utilisant soit un tableau, soit une liste chaînée.



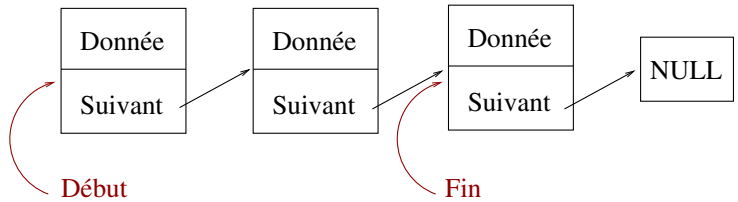
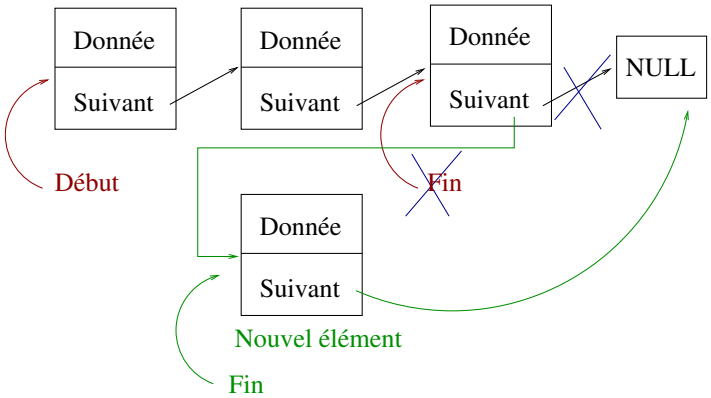
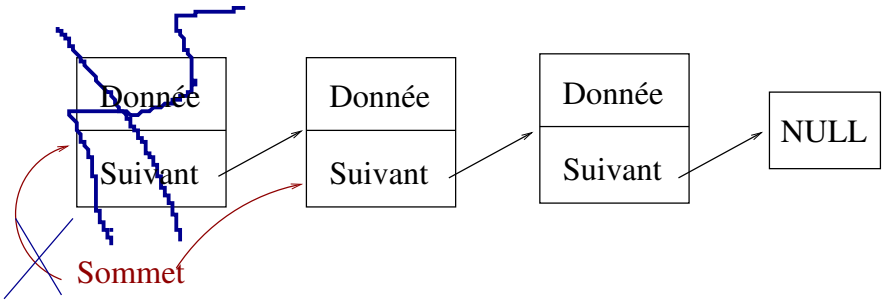


Schéma métaphore.



Représentation informatique avec une liste chaînée.



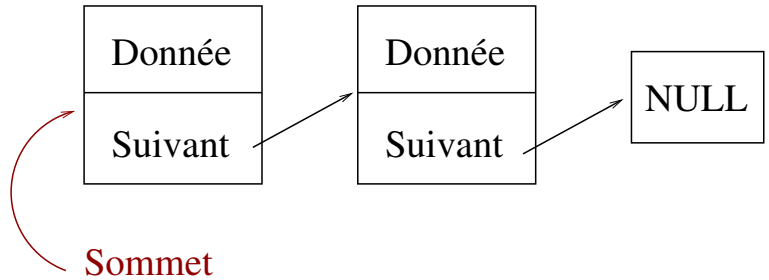
Insertion dans une file : premier entré, premier servi, l'insertion se fait à la fin.

Suppression : premier entré, premier servi, on supprime le début.

Pile

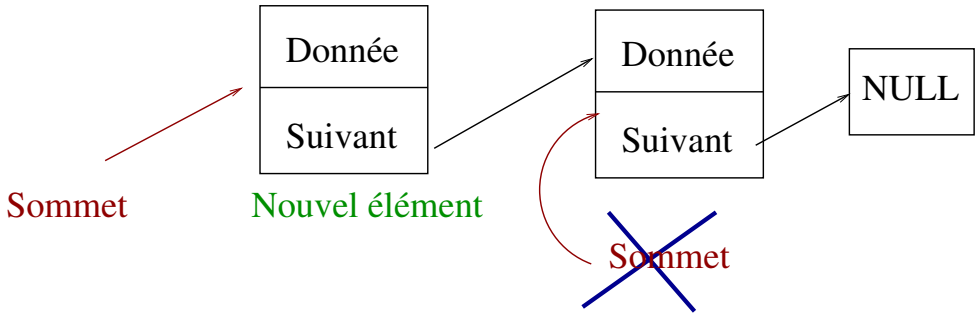
Rappel Pile

- Une pile (LIFO : last in first out, dernier entré/premier servi) est comme une pile d'assiettes ou **mieux de containers** (très lourds) : on ne manipule que le **sommet** de la pile.

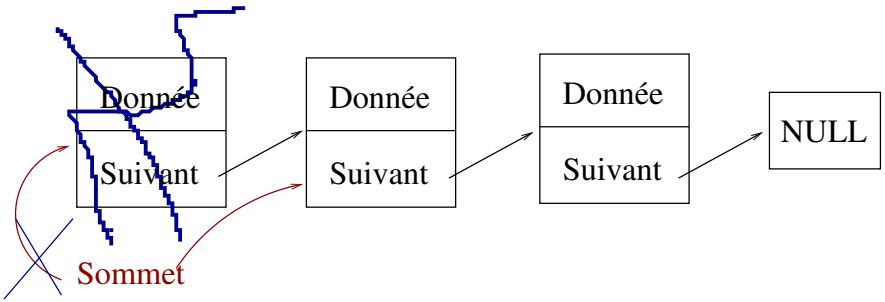


Représentation informatique avec une liste chaînée.

Schéma métaphore.



Empiler dans une pile : dernier rentré, premier servi, on empile au sommet.



Dépiler : dernier entré, premier servi, on dépile au sommet.

Parcours en largeur

Exemple

Vous devez visiter un grand nombre de **pages sur des sites Web** : les pages sont les sommets et un lien entre deux pages est une arête entre ces deux sommets. Vous devez alors classifier ces sites.

Parcours en largeur

Exemple

Vous devez visiter un grand nombre de **pages sur des sites Web** : les pages sont les sommets et un lien entre deux pages est une arête entre ces deux sommets. Vous devez alors classifier ces sites.

Le principe de l'algorithme de **parcours en largeur** est alors le suivant.

Entrée : Un graphe.
Sortie : Les sommets du graphe rangés dans l'ordre du parcours en largeur.
On utilise une **file** F. On enfile le sommet de départ et on le marque.
tant que F n'est pas **vide** **faire**
    On visite les voisins de la tête de file.
    On les enfile s'ils ne sont pas déjà marqués.
    On les marque alors.
    On défile (et on traite le sommet défilé).
fin tant que

Parcours en largeur

Exemple

Vous devez visiter un grand nombre de **pages sur des sites Web** : les pages sont les sommets et un lien entre deux pages est une arête entre ces deux sommets. Vous devez alors classifier ces sites.

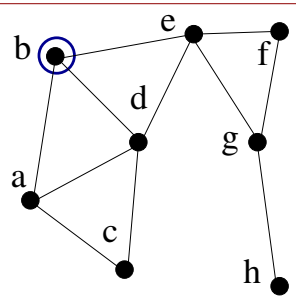
Le principe de l'algorithme de **parcours en largeur** est alors le suivant.

Entrée : Un graphe.
Sortie : Les sommets du graphe rangés dans l'ordre du parcours en largeur.
On utilise une **file** F. On enfile le sommet de départ et on le marque.
tant que F n'est pas **vide faire**
    On visite les voisins de la tête de file.
    On les enfile s'ils ne sont pas déjà marqués.
    On les marque alors.
    On défile (et on traite le sommet défilé).
fin tant que

Théorème (Complexités du parcours en largeur de graphe)

Dans le pire des cas, la **complexité en temps** est de $O(|V| + |E|)$, en traitant tous les sommets et toutes les arêtes d'un graphe $G = (V, E)$. La **complexité en espace** est de $O(|V|^2)$.

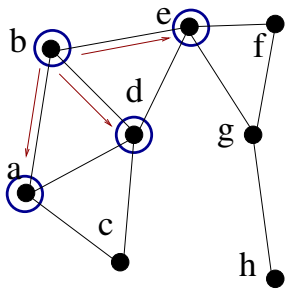
Parcours en largeur (exemple)



File:

b →

Marqués: {b}

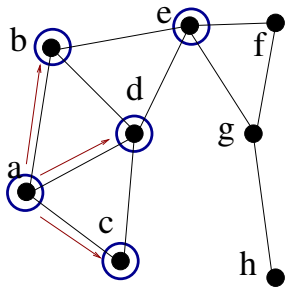


File:

~~e, d, a, b~~ →

Marqués: {b,a,d,e}

b est traité et supprimé.

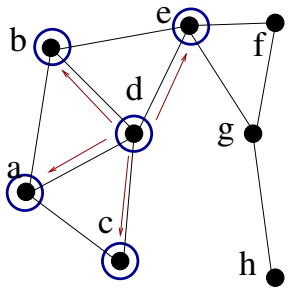


File:

~~c, e, d, a~~ →

Marqués: {b, a, d, e, c}

a est traité et supprimé.



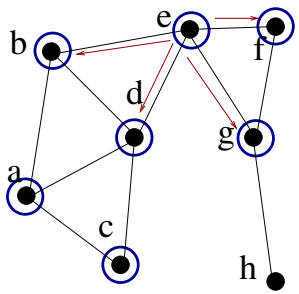
File:

~~c, e, d~~ →

Marqués: {b, a, d, e, c}

d est traité et supprimé.

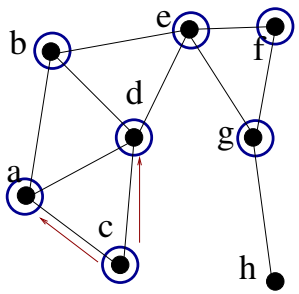
Parcours en largeur (exemple)



File: ~~f, g, c, e~>

Marqués: {b, a, d, e, c, g, f}

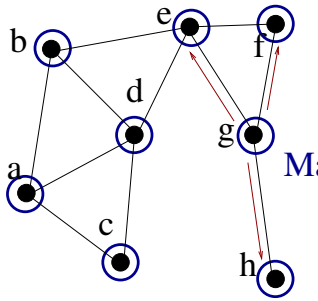
e est traité et supprimé.



File: ~~f, g, c~>

Marqués: {b, a, d, e, c, g, f}

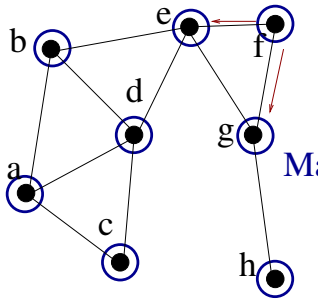
c est traité et supprimé.



File: ~~h, f, g~>

Marqués: {b, a, d, e, c, g, f, h}

g est traité et supprimé.

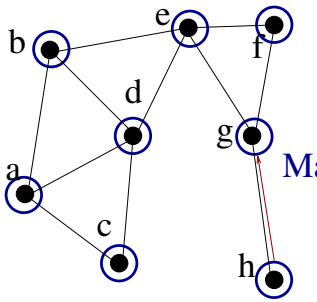


File: ~~h, f~>

Marqués: {b, a, d, e, c, g, f, h}

f est traité et supprimé.

Parcours en largeur (exemple)



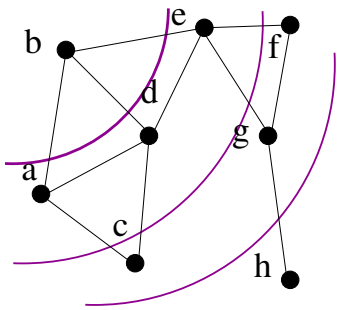
File:

h

Marqués: {b, a, d, e, c, g, f, h}

h est traité et supprimé.

La file est vide!



Ordre des traitements:

b; a, d, e; c, g, g; h.

Parcours en profondeur

Le principe de l'algorithme de **parcours en profondeur** est le suivant.

Entrée : Un graphe.
Sortie : Les sommets du graphe rangés dans l'ordre du parcours en profondeur.
On utilise une **pile** P. On empile le sommet de départ, on le marque.
tant que P n'est pas **vide faire**
    si le sommet de la pile P a au moins un voisin non marqué **alors**
        On sélectionne un voisin non marqué et on le marque.
        On l'empile.
    sinon
        On dépile.
    fin si
fin tant que

Parcours en profondeur

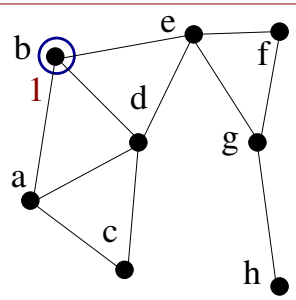
Le principe de l'algorithme de **parcours en profondeur** est le suivant.

Entrée : Un graphe.
Sortie : Les sommets du graphe rangés dans l'ordre du parcours en profondeur.
On utilise une **pile** P. On empile le sommet de départ, on le marque.
tant que P n'est pas **vide faire**
    si le sommet de la pile P a au moins un voisin non marqué **alors**
        On sélectionne un voisin non marqué et on le marque.
        On l'empile.
    sinon
        On dépile.
    fin si
fin tant que

Théorème (Complexités du parcours en profondeur de graphe)

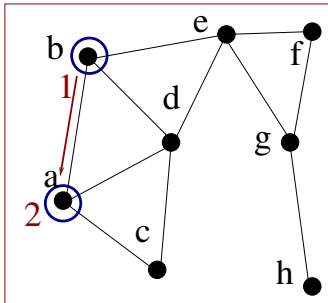
Dans le pire des cas, la **complexité en temps** est de $O(|V| + |E|)$, en traitant tous les sommets et toutes les arêtes d'un graphe $G = (V, E)$. La **complexité en espace** est de $O(|V|^2)$.

Parcours en profondeur (exemple)



Pile:

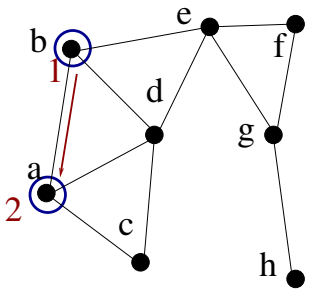
b	
1	



On prend un voisin non marqué du sommet de la pile et on l'empile pour le visiter.

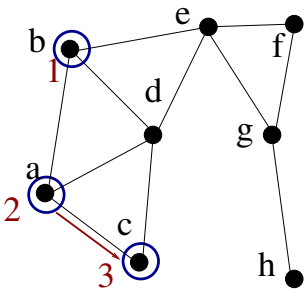
Pile:

b	a	
1	2	



Pile:

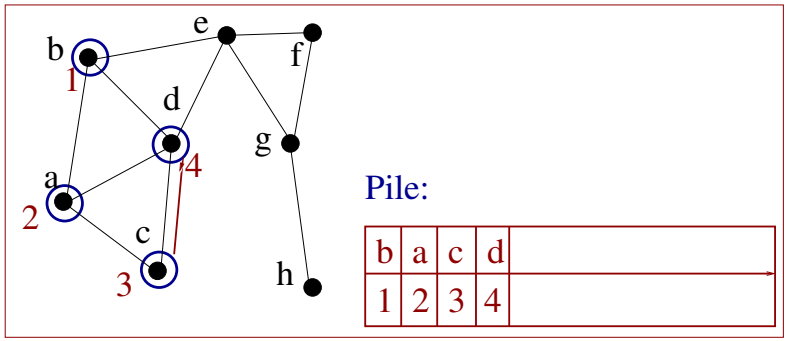
b	a	
1	2	



Pile:

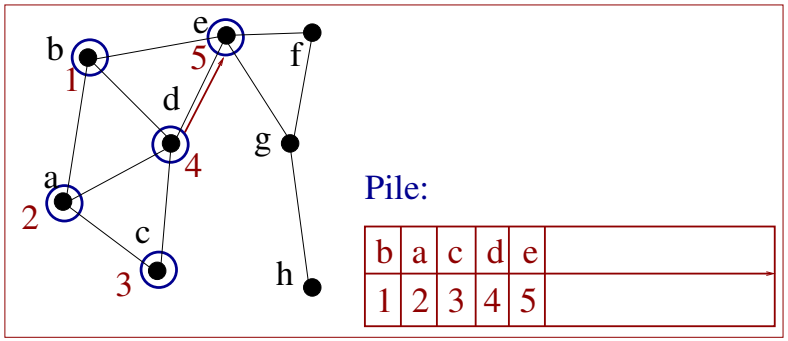
b	a	c	
1	2	3	

Parcours en profondeur (exemple)



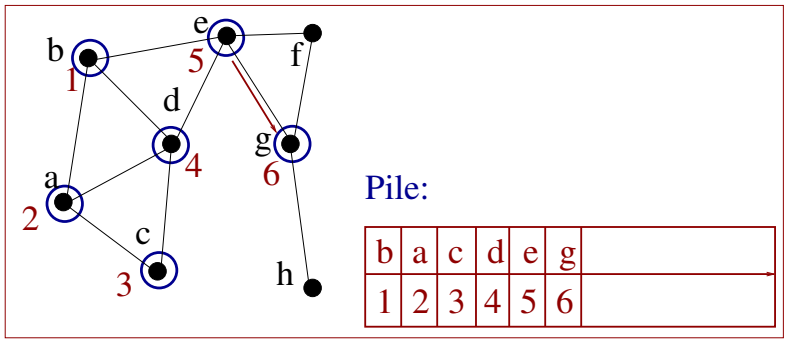
Pile:

b	a	c	d	
1	2	3	4	



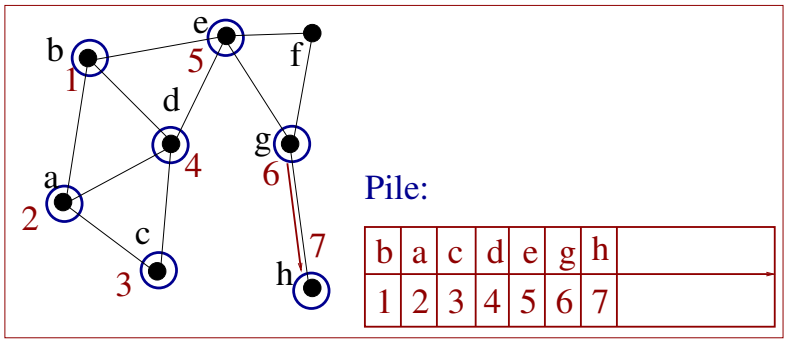
Pile:

b	a	c	d	e	
1	2	3	4	5	



Pile:

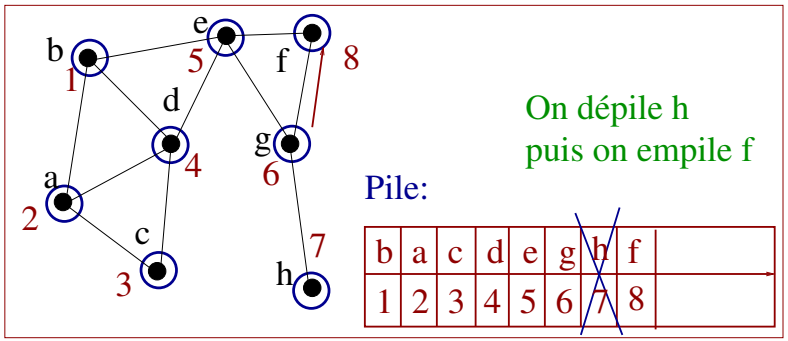
b	a	c	d	e	g	
1	2	3	4	5	6	

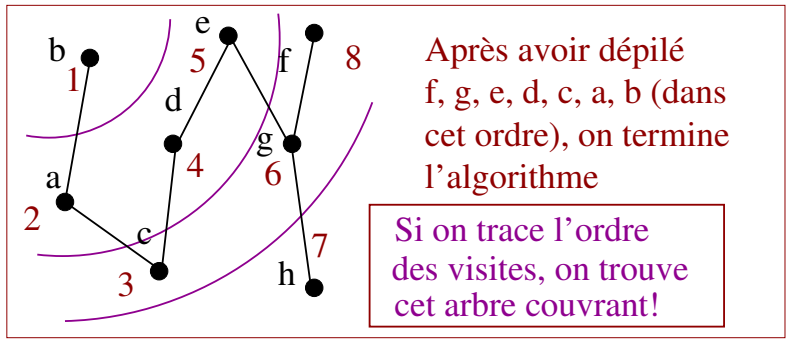


Pile:

b	a	c	d	e	g	h	
1	2	3	4	5	6	7	

Parcours en profondeur (exemple)





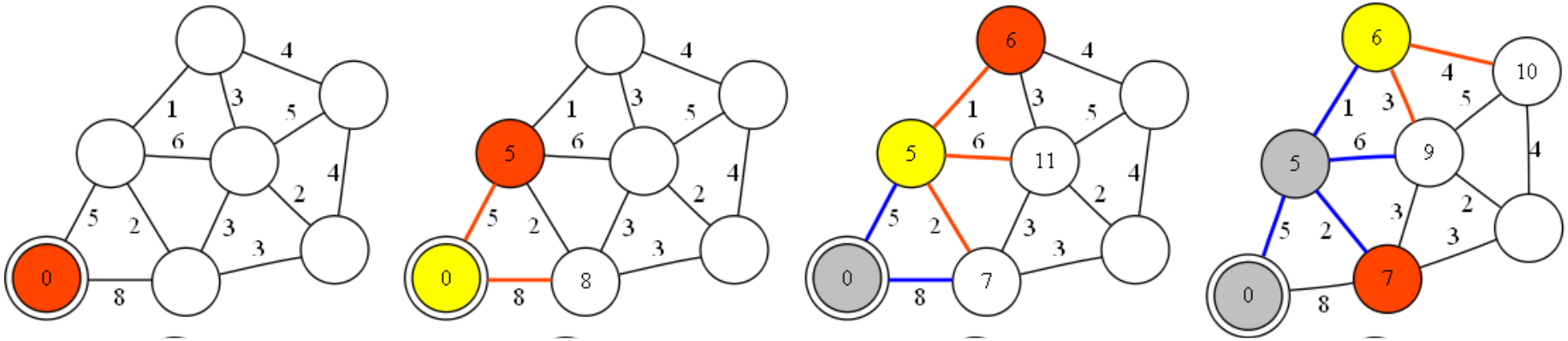
Remarques

Il est important de noter les choses suivantes sur les algorithmes de parcours :

- ils visitent **tous les sommets** et **toutes les arêtes** du graphe
- ils permettent de déterminer si le graphe est **connexe ou non**
- ils permettent de calculer les **composantes connexes** du graphe
- ils permettent de calculer les arbres couvrants des composantes connexes du graphe (la **forêt couvrante** du graphe)
- ils peuvent être **modifiés** pour trouver d'autres algorithmes comme :
  1. trouver et renvoyer **un chemin entre 2 sommets quelconques** (il suffit de renvoyer le contenu de la pile entre les deux sommets u et v)
  2. tester s'il y a un **cycle** dans un graphe (il suffit de regarder une composante connexe et le comparer à son arbre couvrant)

Un algorithme de propagation

Dans les figures suivantes, O est l'origine d'un feu (ou d'une maladie) qui peu se propager d'un sommet à un autre du graphe (les sommets sont des arbres ou des individus) en mettant le temps marqué par les arêtes du graphe. A chaque étape, on a exactement un sommet (appelé aussi nœud) qui est le plus proche de la zone contaminée (ou brûlée) qui le devient.



Au départ, le processus est en O. Le nœud (ou sommet) le plus proche (non "rongé" par le feu ou la maladie) est alors en **orange** et il est donc atteint à l'étape suivante. Le nœud en **jaune** est celui qui vient d'être "rongé". Les sommets en **gris** sont ceux qui ont été déjà "rongés".

Formalisation de l'algorithme

Définition Le problème de l'arbre enraciné de poids minimum.

Etant donné un graphe $G = (V, E)$ pondéré (i.e. chaque arête a un poids) et un sommet source, ce problème consiste à déterminer un arbre de **poids minimal** dont la racine est donné.

Formalisation de l'algorithme

Définition Le problème de l'arbre enraciné de poids minimum.

Etant donné un graphe $G = (V, E)$ pondéré (i.e. chaque arête a un poids) et un sommet source, ce problème consiste à déterminer un arbre de **poids minimal** dont la racine est donné.

La stratégie de cet algorithme consiste à construire un arbre de poids minimal à partir du sommet origine. La page précédente est l'illustration de ses premières étapes. L'algorithme est une modification de celui de **Dijkstra**.

Entrée : Un graphe G et un sommet o.
Sortie : L'arbre couvrant de poids minimal enraciné en o.
Initialisation : tout sommet v de G est marqué NON-RONGÉ, père(v) = INDÉFINI, $\delta(v) = +\infty$.
$\delta(o) = 0$, o est marqué RONGÉ.
tant que il existe un somment NON-RONGÉ **faire**
1) Trouver le sommet w (sommet **orange**) le plus proche des sommets déjà rongés.
2) Marqué w comme RONGÉ
pour chaque voisin v de w **faire**
si $\delta(w) + \text{distance}(v, w) < \delta(v)$ **alors**
$\delta(v) = \delta(w) + \text{distance}(v, w)$
père$(v) = w$ (**un seul parent par sommet : on a un arbre**)
fin si
fin pour
fin tant que

Formalisation de l'algorithme

Définition Le problème de l'arbre enraciné de poids minimum.

Etant donné un graphe $G = (V, E)$ pondéré (i.e. chaque arête a un poids) et un sommet source, ce problème consiste à déterminer un arbre de **poids minimal** dont la racine est donné.

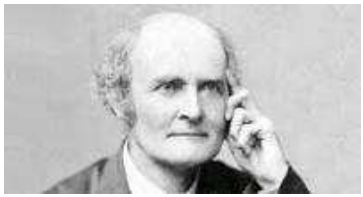
La stratégie de cet algorithme consiste à construire un arbre de poids minimal à partir du sommet origine. La page précédente est l'illustration de ses premières étapes. L'algorithme est une modification de celui de **Dijkstra**.

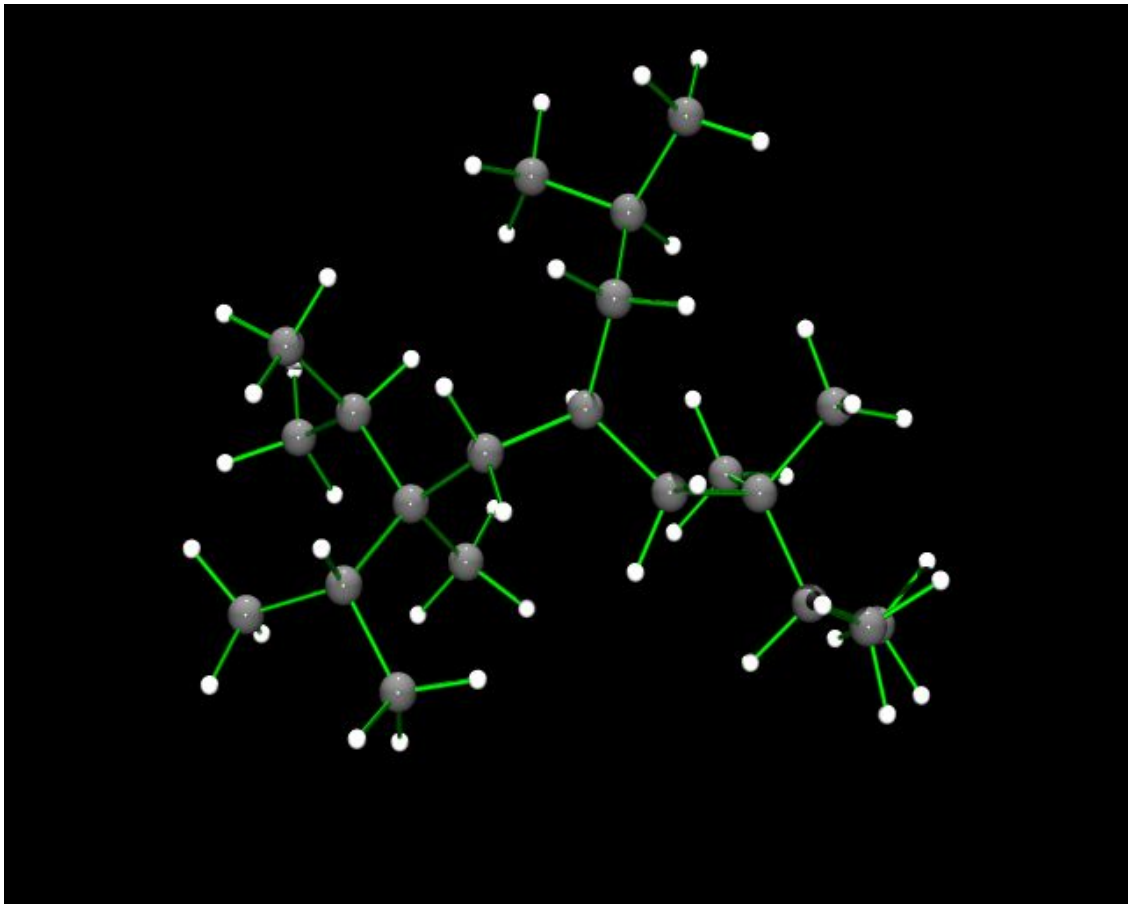
Entrée : Un graphe G et un sommet o.
Sortie : L'arbre couvrant de poids minimal enraciné en o.
Initialisation : tout sommet v de G est marqué NON-RONGÉ, père(v) = INDÉFINI, $\delta(v) = +\infty$.
$\delta(o) = 0$, o est marqué RONGÉ.
tant que il existe un somment NON-RONGÉ **faire**
1) Trouver le sommet w (sommet **orange**) le plus proche des sommets déjà rongés.
2) Marqué w comme RONGÉ
pour chaque voisin v de w **faire**
si $\delta(w) + \text{distance}(v, w) < \delta(v)$ **alors**
$\delta(v) = \delta(w) + \text{distance}(v, w)$
père$(v) = w$ (**un seul parent par sommet : on a un arbre**)
fin si
fin pour
fin tant que

Théorème

L'algorithme ci-dessus trouve l'arbre couvrant de poids minimum du graphe G enraciné en o.

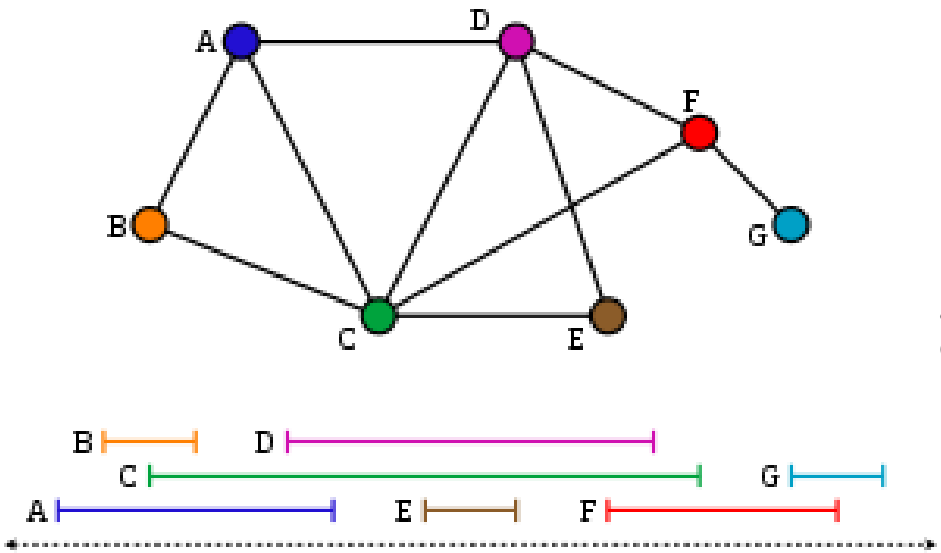
Arbres et hydrocarbones





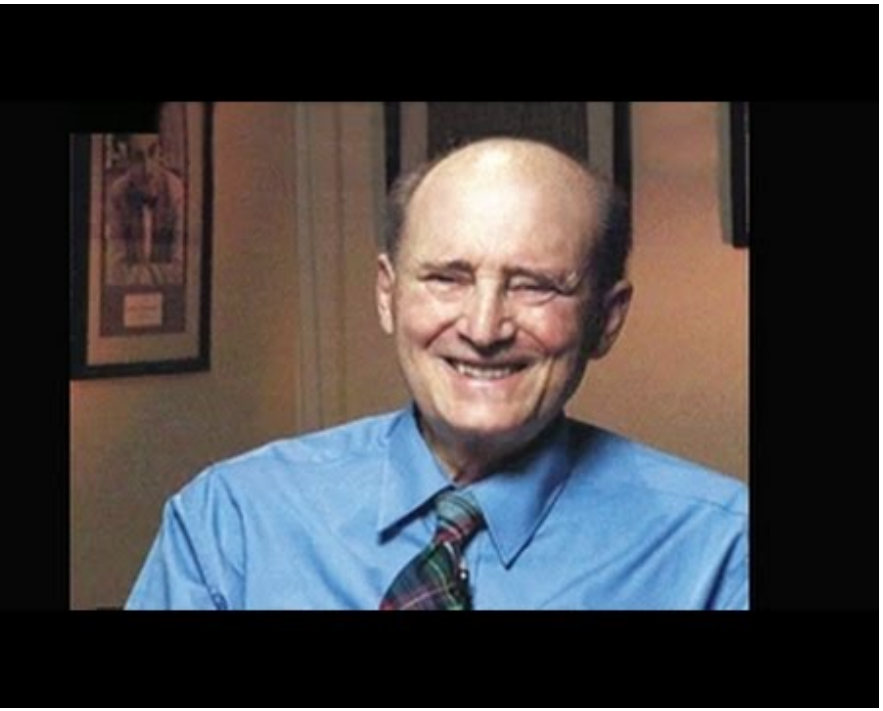
Arthur Cayley : utilisation des arbres pour énumérer les hydrocarbones (vers 1850).

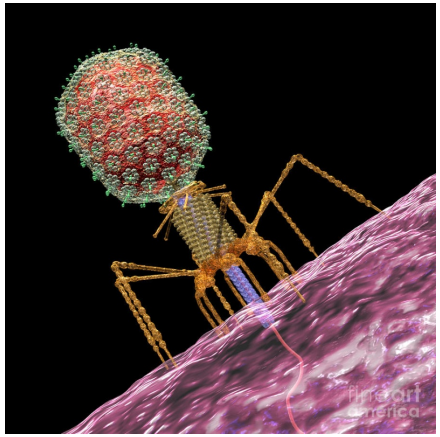
Graphe d'intervalles



En théorie des graphes, un graphe d'intervalle est le graphe d'intersection d'un ensemble d'intervalles de la droite réelle. Chaque sommet du graphe d'intervalle représente un intervalle de l'ensemble, et une arête relie deux sommets lorsque les deux intervalles correspondants s'intersectent.

Des graphes d'intervalles, des virus et des bacteries





Seymour Benzer : physicien, biologiste moléculaire et généticien.

Des graphes d'intervalles, des virus et des bacteries

Remarque

Les travaux de Seymour Benzer mettent en exergue l'utilisation des **graphes d'intervalles** (comme outils d'analyse) sur les relations entre les **virus T4** et les **bacteries *Escherichia coli***.

Des graphes d'intervalles, des virus et des bacteries

Remarque

Les travaux de Seymour Benzer mettent en exergue l'utilisation des **graphes d'intervalles** (comme outils d'analyse) sur les relations entre les **virus T4** et les **bacteries *Escherichia coli***.

- Normalement le T4 s'attaque à cette bactérie.
- Cependant si le T4 mute, il peut perdre cette aptitude.
- On étudie des bactéries infectées par 2 virus mutants (ayant tous les deux perdus l'aptitude de s'attaquer au E. Coli).
- Surprise : de manière indépendante les deux virus ne s'attaquent pas à la bactérie mais ensemble ils la tuent.
- C'est un des travaux fondamentaux de Seymour Benzer : la **preuve de la linéarité des gènes**.

Modélisation et preuve utilisant des graphes d'intervalles

Idée

- Fil conducteur : la bactérie infectée avec des T4 mutants.
- Chaque T4 a un intervalle (**inconnu**) supprimé de son génome.
- Si les deux intervalles se superposent : la bactérie survit car les deux mutants ne se "complètent" pas.
- Si les deux intervalles ne se superposent pas : la paire est complémentaire et ils tuent la bactérie.

Modélisation et preuve utilisant des graphes d'intervalles

Idée

- Fil conducteur : la bactérie infectée avec des T4 mutants.
- Chaque T4 a un intervalle (**inconnu**) supprimé de son génome.
- Si les deux intervalles se superposent : la bactérie survit car les deux mutants ne se "complètent" pas.
- Si les deux intervalles ne se superposent pas : la paire est complémentaire et ils tuent la bactérie.

***LA* modélisation.** On construit un **graphe d'intervalles** comme suit : chaque T4 mutant est un **sommet**, on dessine une **arête** entre une paire de mutants quand la bactérie inféctée par ses deux derniers survit.

Modélisation et preuve utilisant des graphes d'intervalles

Idée

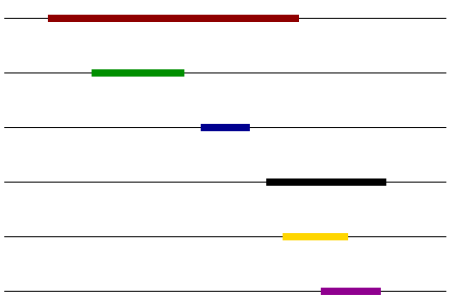
- Fil conducteur : la bactérie infectée avec des T4 mutants.
- Chaque T4 a un intervalle (**inconnu**) supprimé de son génome.
- Si les deux intervalles se superposent : la bactérie survit car les deux mutants ne se "complètent" pas.
- Si les deux intervalles ne se superposent pas : la paire est complémentaire et ils tuent la bactérie.

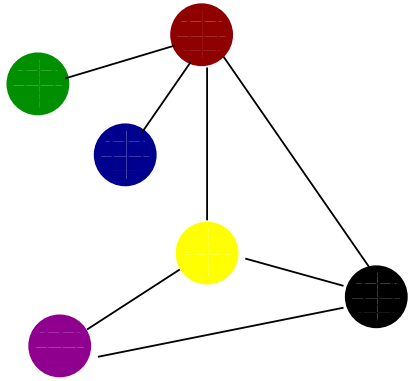
***LA* modélisation.** On construit un **graphe d'intervalles** comme suit : chaque T4 mutant est un **sommet**, on dessine une **arête** entre une paire de mutants quand la bactérie infféctée par ses deux derniers survit.

But : analyser la structure des graphes d'intervalles pour étudier la structure de l'ADN : **linéaire** ou avec des **branches**.

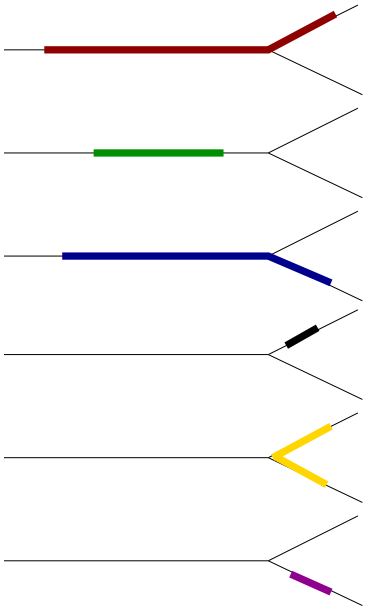
La découverte de Seymour Benzer

Elle se résume avec ses figures :

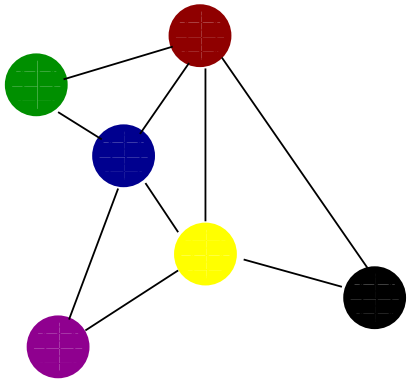




Génome linéaire



"avec des branchements"



L'algo de Dijkstra (1959)

Cet algorithme sert à trouver le **plus court chemin** d'un sommet s à un sommet t mais il est **plus général** car on peut demander à avoir la liste des plus courts chemins de s à tous les sommets du graphe. A noter que le graphe peut être **orienté**.

- En entrée, nous avons un graphe orienté pondéré par des réels positifs et un sommet source.
- On construit progressivement un **sous-graphe** dans lequel sont classés les différents sommets par ordre croissant de **distance** par rapport à s. Cette distance est la **somme** des poids des arcs empruntés.
- Au début, les distances sont toutes infinies sauf pour s qui est à distance 0. Le sous-graphe est vide.
- A chaque itération, on choisit un sommet **de distance minimale n'appartenant pas encore au sous-graphe**. On le rajoute au sous-graphe. On met alors à jour les distances des sommets voisins de celui qui vient d'être rajouté : c'est le minimum entre la distance d'avant et celle obtenue en rajoutant le poids de l'arc entre le sommet voisin et le sommet ajouté.

Pseudo-code Dijkstra

Dijkstra(graphe $G = (V, E)$, sommet s)
Entrée : Un graphe G et un sommet s de ce graphe.
Sortie : La liste des distances de tous les sommets de G à partir de s.
$\text{dist}[s] = 0$
pour tout sommet v dans $V - \{s\}$ **faire**
    $\text{dist}[s] = +\infty$
fin pour
$S := \emptyset$; $Q := V$ **($\star$ S est l'ensemble des sommets déjà visité(s) $\star$)**
tant que $Q \neq \emptyset$ **faire**
    $u := \text{mindistance}(Q, \text{dist})$ **($\star$ u est l'élément de Q de plus petite distance $\star$)**
    $S := S \bigcup \{u\}$
    pour tout voisin v de u **faire**
        si $\text{dist}[v] > \text{dist}[u] + \text{poids}(u, v)$ **alors**
            $\text{dist}[v] = \text{dist}[u] + \text{poids}(u, v)$
        fin si
    fin pour
fin tant que
retourner dist

Complexité et applications de Dijkstra

En considérant Q (voir algo.) comme une **file de priorité** implémenter en **tas**, la complexité de l'algorithme est en $O((|A| + |S|) \times \log(|S|))$.

Application

- Calcul d'intinéraires :
  arcs = distance, temps, coût, consommation carburant, prix, ...
- protocoles de routages (recherche de parcours efficaces, ...)

Remarque

- L'algorithme de Dijkstra est basé sur un parcours en largeur.
- L'algorithme ne s'applique pas aux graphes avec des arcs de **poids négatifs**.

L'algo de Floyd-Warshall (1959)

Décrit par Bernard Roy en 1959 :) c'est un algorithme qui determine les distances des plus courts chemins entre toutes les paires de sommets dans un **graphe orienté et pondéré** **avec des poids négatifs mais sans que le graphe ne possède de circuit de poids strictement négatif.** La complexité de l'algorithme est en temps $O(n^3)$ pour n sommets.

- On note $W^k_{i,j}$ le poids minimal d'un **chemin du sommet i au sommet j** n'empruntant que des sommets intermédiaires dans $\{1, 2, 3, \cdots, k\}$ s'il en existe sinon c'est $+\infty$.
- On note W^k la matrice $W^k = [W^k_{i,j}]$, W^0 étant la **matrice d'adjacence** du graphe de départ.
- Si p est un chemin entre i et j de poids minimal dont les sommets intermédiaires sont dans $\{1, \cdots, k\}$ alors :
  1. soit p n'emprunte pas k
  2. soit p emprunte k exactement une fois et donc p est la concaténation de deux chemins de i à k puis de k à j avec des sommets intermédiaires dans $\{1, 2, \cdots, k-1\}$ d'où la récurrence :

$$W^k_{i,j} = \min(W^{k-1}_{i,j}, W^{k-1}_{i,k} + W^{k-1}_{k,j})$$

Pseudo-code de Floyd-Warshall

FloydWarshall(graphe G)
Entrée : (graphe G donné par sa matrice d'adjacence M)
Sortie : W^n la matrice des distances entre tous les sommets.
W^0 := matrice d'adjacence de G
pour k de 1 à n **faire**
    pour i de 1 à n **faire**
        pour j de 1 à n **faire**
            $W^k_{i,j} := \min(W^{k-1}_{i,j},\ W^{k-1}_{i,k} + W^{k-1}_{k,j})$
        fin pour
    fin pour
fin pour
retourner W^n

Complexité et applications de Floyd-Warshall

On lit directement sa complexité en temps : $O(n^3)$. On peut optimiser l'espace en ne mémorisant qu'une matrice $n \times n$ (i.e. W^k est remplacé par W) : complexité en espace en $O(n^2)$.

- Plus courts chemins notamment dans les graphes orientés non pondérés (poids unitaire pour chaque arc
- Calcul de la fermeture transitive d'un graphe orienté (on rajoute un arc entre x et y ssi il existe un chemin orienté de x à y) : en remplaçant

$$W_{i,j} := \min(W_{i,j},\ W_{i,k} + W_{k,j})$$

par

$$C[i,j] = C[i,j] \text{ OR } (C[i,k] \text{ AND } C[k,j])\,.$$

Remarque

Attention : Les arcs du graphe peuvent avoir des poids négatifs, mais le graphe ne doit pas posséder de circuit de poids strictement négatif !

Algo de Bellman-Ford

Découvert en 1956 (Ford) redécouvert en 1958 (Bellman) puis en 1959 (Moore), cet algorithme calcule les plus courts chemins depuis un sommet source donné dans un graphe orienté pondéré. Il a la **particularité** d'autoriser la présence d'arcs de poids négatif et permet de détecter l'existence d'un **circuit absorbant**, c'est-à-dire de poids total strictement négatif, accessible depuis le sommet source.

- On note $d[t,\, k]$ la distance du sommet source s à t avec un chemin d'au plus k arcs.
- On a donc $d[t,\, 0] = +\infty$ pour $t \neq s$ et $d[s,\, 0] = 0$
- Puis

$$d[t, k] = \min(d[t, k-1], \min_{\text{arc}(u,t)} (d[u, k-1] + \text{poids}(u, t)))$$

Pseudo-code de Bellman-Ford

BellmanFord(graphe G, sommet s)
Entrée : (graphe G et sommet de départ s)
Sortie : distance de s à tous les sommets de G.
pour chaque sommet u de G **faire**
    $d[u] := +\infty$
fin pour
$d[s] := 0$
pour k de 1 à $n-1$ **faire**
    pour chaque arc $(u,\ t)$ de G **faire**
        $d[t] := \min(d[t],\ d[u] + \text{poids}(u,\ t))$
    fin pour
fin pour
retourner d

Comme il y a un cycle de poids négatif ssi un nouveau tour de boucle fera diminuer la distance alors la détection d'un cycle de poids négatif se fait à la fin de l'algo comme suit :

pour chaque arc $(u,\ t)$ de G **faire**
    si $d[u] + \text{poids}(u,\ t) < d[t]$ **alors**
        retourner VRAI
    fin si
fin pour
retourner FAUX

Algorithme de Borůvka (1926)

Définition 1.

Etant donné un **graphe non orienté connexe** dont les arêtes sont pondérées, un **arbre couvrant de poids minimal** de ce graphe est un arbre couvrant (sous-graphe qui est un arbre et qui couvre tous les sommets) dont la somme des poids des arêtes est minimale.

L'algorithme de Borůvka est axé sur les principes suivants.
Dans la boucle principale, on effectue les opérations suivantes.

- On commence par dètruire les boucles du graphe initial G (de telles boucles peuvent apparaître après une itération).
- Pour tout sommet, on rajoute dans l'arbre résultat l'arête de poids minimum adjacente à ce sommet, puis on contracte cette arête.
- On construit ainsi une forêt couvrante dont les arbres vont fusionner.

Pseudo-code Borůvka

Borůvka
Entrée : graphe connexe G avec arêtes pondérées.
Sortie : un arbre couvrant de poids minimum.
On initialise $\mathcal{F}$ à tous les sommets mais **vide d'arêtes**.
tant que $\mathcal{F}$ a plus qu'une composante connexe **faire**
    Détruire les boucles de G
    Remplacer les arêtes multiples entre deux sommets par une seule de poids minimum
    pour tout sommet x de G **faire**
        Trouver l'arête $e(x)$ de poids minimum adjacente à x
        $\mathcal{F} := \mathcal{F} \bigcup \{e(x)\}$
        ($\star$ on rajoute l'arête dans la forêt $\star$)
        Contracter $e(x)$ en fusionnant ses deux extrémités.
    fin pour
fin tant que
retourner $\mathcal{F}$

Complexité et applications de Borůvka

La logique générale est la suivante tant que le calcul de la forêt n'est pas terminé :

- trouver les meilleures arêtes pour chaque sommet ou groupe
- fusionner les groupes de sommets
- enlever les arêtes utilisées et celles qui créent des cycles

A chaque étape, on a au plus $n/2$ groupes. Donc on aura du $O(\log |S|)$ étapes générales pour mettre tous les sommets dans le même groupe. A chaque étape, on va visiter toutes les arêtes pour trouver les arêtes minimales donc $O(|A|)$ par étape : on arrive à une **complexité** de $O(|A| \log |S|)$.

($\star$ $|S|$ = nbr. sommets et $|A|$ = nbr. arêtes $\star$)
C'est historiquement le premier algorithme de recherche d'arbre couvrant de poids minimal découvert en 1926 et il était destiné à rendre le réseau électrique de Moravie (aujourd'hui en Tchéquie) efficace. Il fut redécouvert à de nombreuses reprises par la suite3.

Algorithme de PRIM (du à Jarnik en 1930)

Remarque

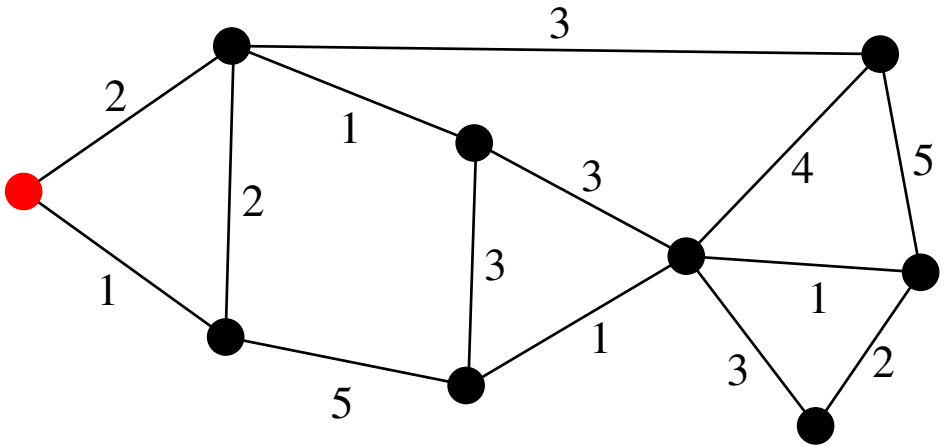
L'algorithme a été développé en 1930 par le mathématicien tchèque [Vojtech Jarnik] puis a été redécouvert et republié par [Robert C. Prim] et [Edsger W. Dijkstra] en 1959.

- On fait croître de manière gloutonne un arbre depuis un sommet.
- A chaque étape, on rajoute une arête de poids minimum ayant exactement une extrémité dans l'arbre (pour éviter un cycle)

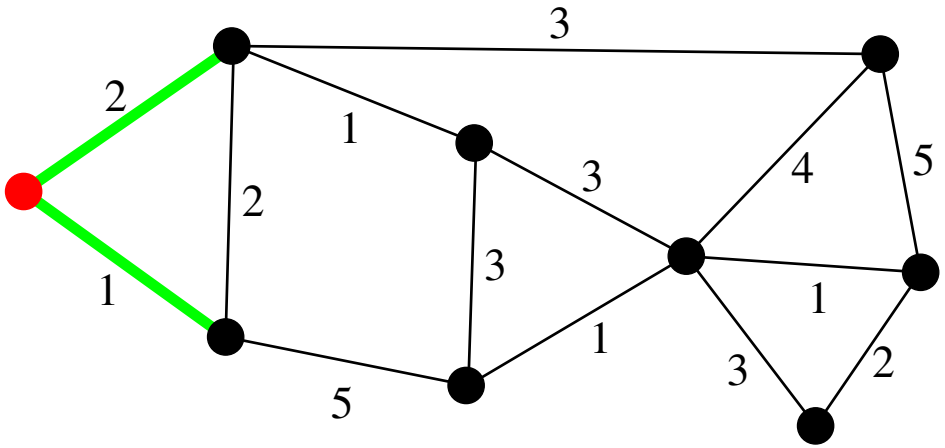
Prim
Entrée : graphe connexe G avec arêtes pondérées.
Sortie : un arbre couvrant de poids minimum.
On initialise : $\mathcal{F}$ avec un sommet racine et sans arêtes.
tant que Tous les sommets de G ne sont pas dans $\mathcal{F}$ **faire**
    Trouver toutes les arêtes de G qui relient $\mathcal{F}$ et un sommet extérieur à $\mathcal{F}$
    Choisir alors celle avec le plus petit poids.
fin tant que
retourner $\mathcal{F}$

L'algorithme est en $O(|S|^2)$ si le graphe est donné comme une matrice et en $O(|A| \log |S|)$ si on utilise des listes d'adjacences combiné avec un tas binaire (exercice !).

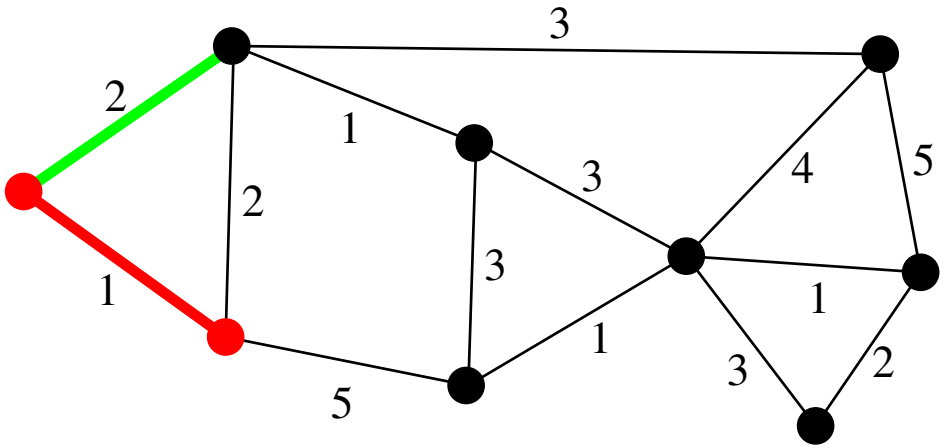
Exemple



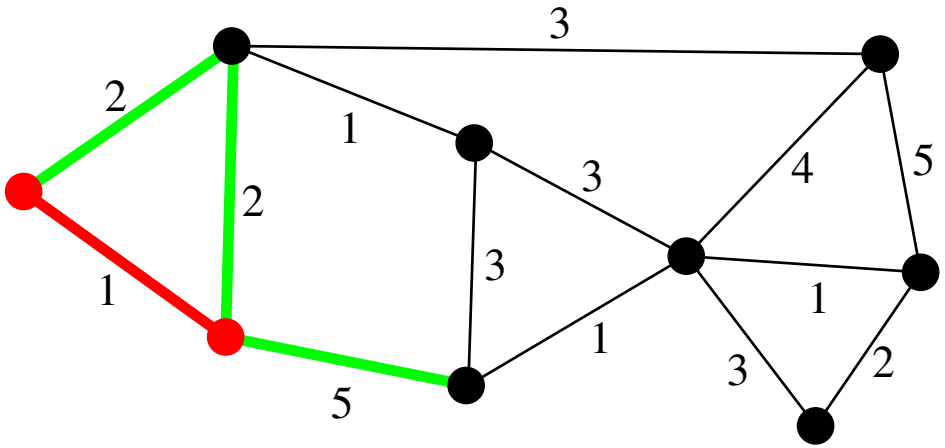
Exemple



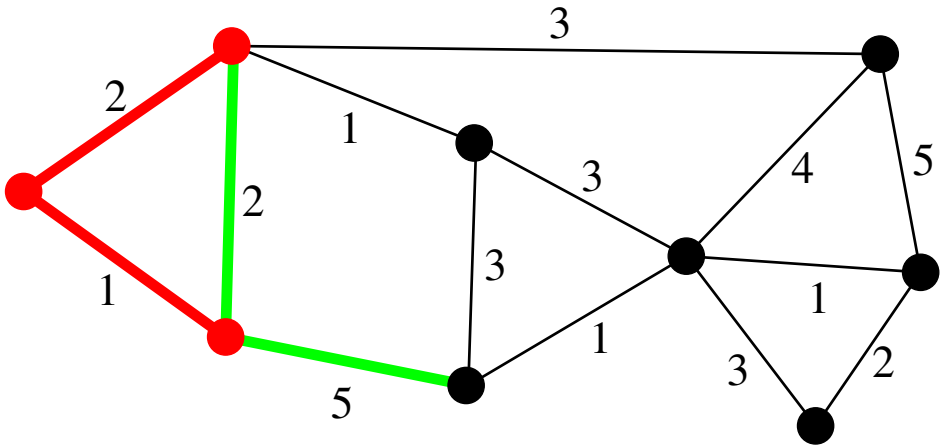
Exemple



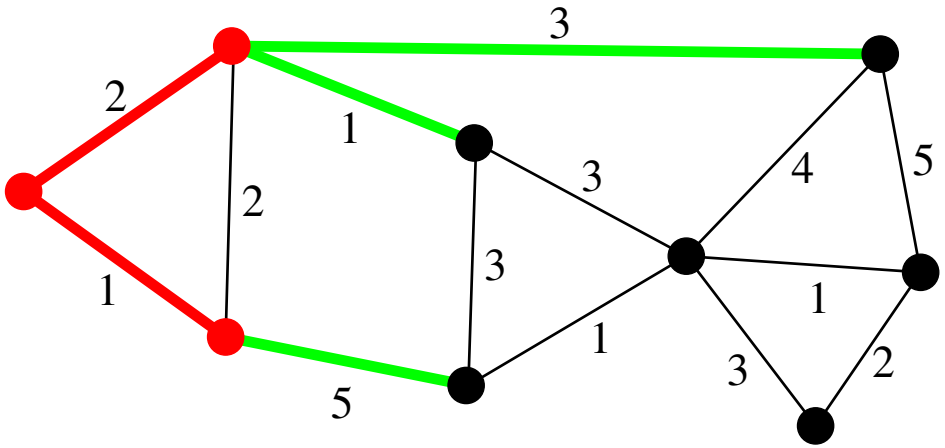
Exemple



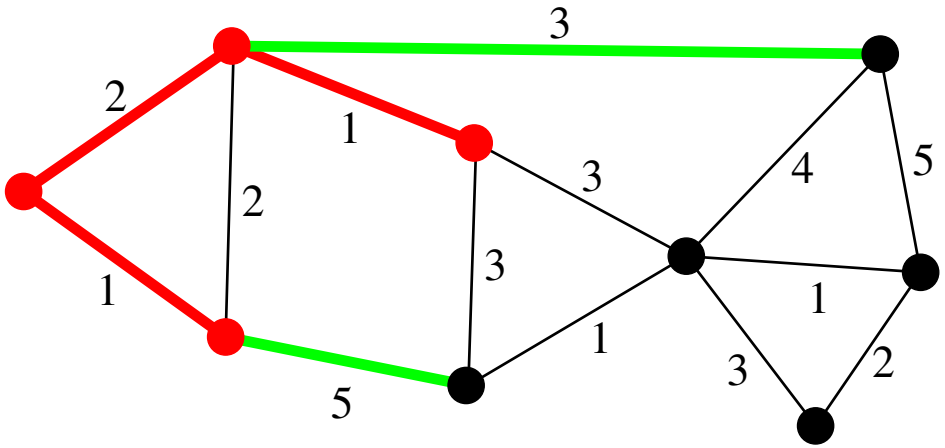
Exemple



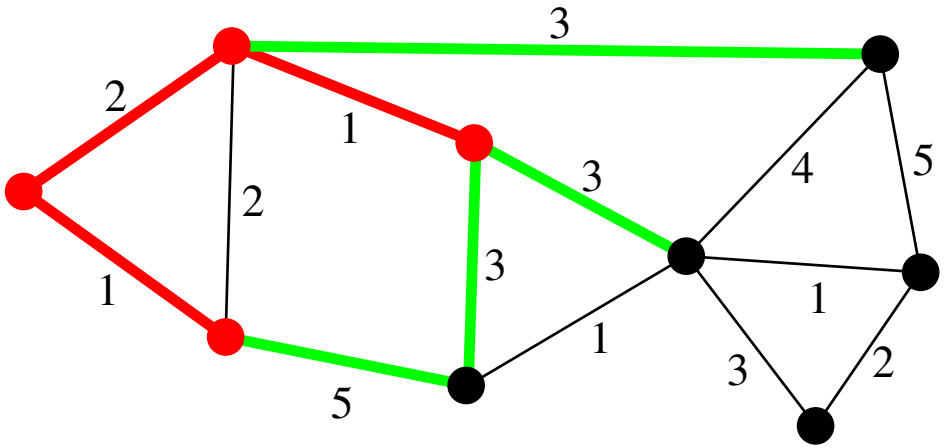
Exemple



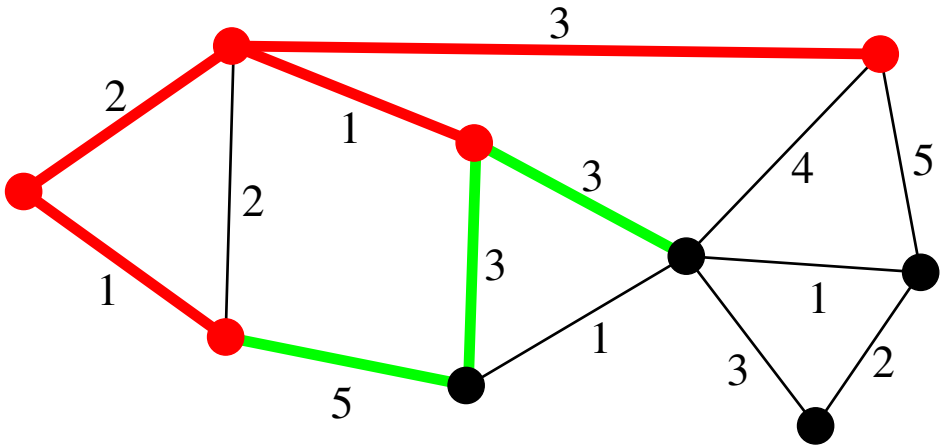
Exemple



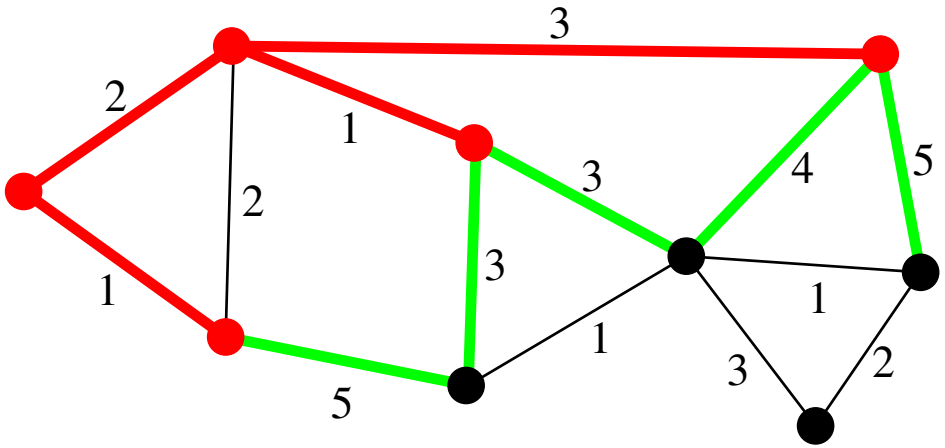
Exemple



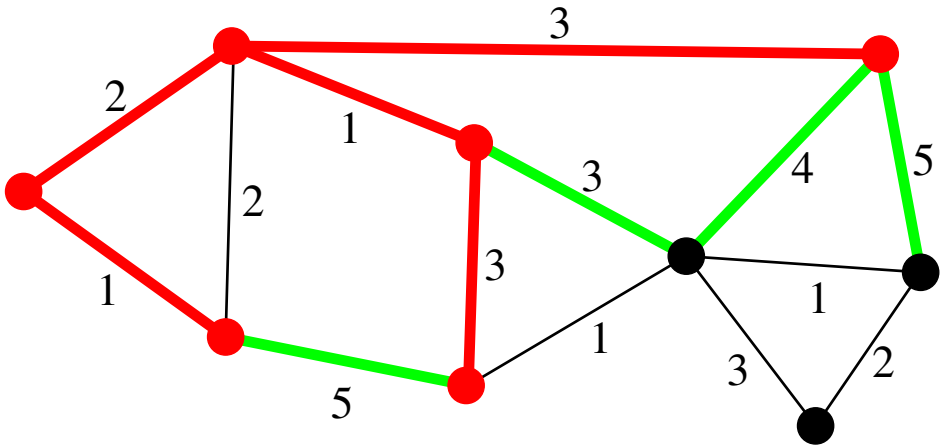
Exemple



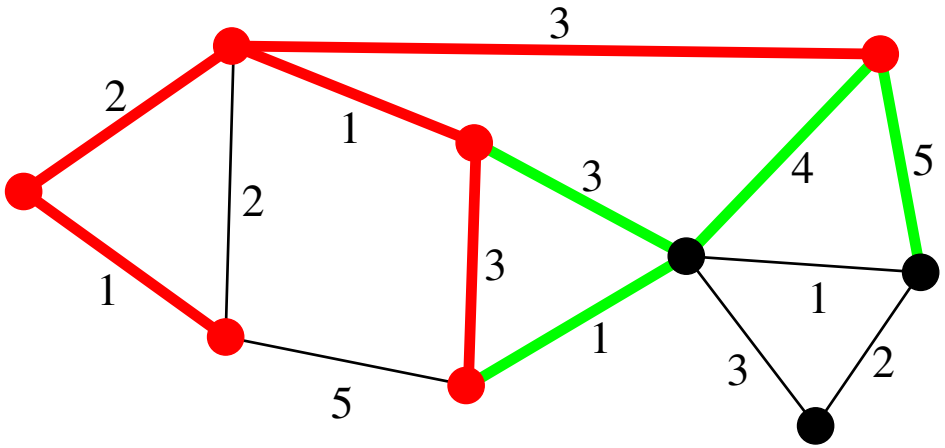
Exemple



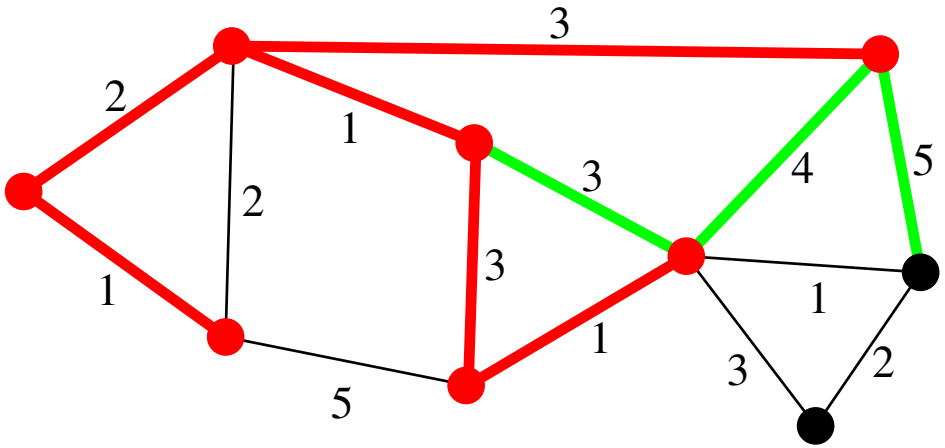
Exemple



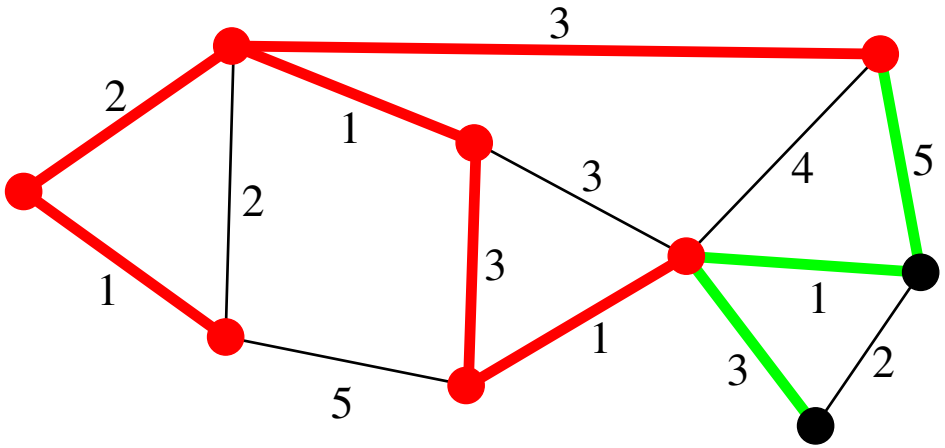
Exemple



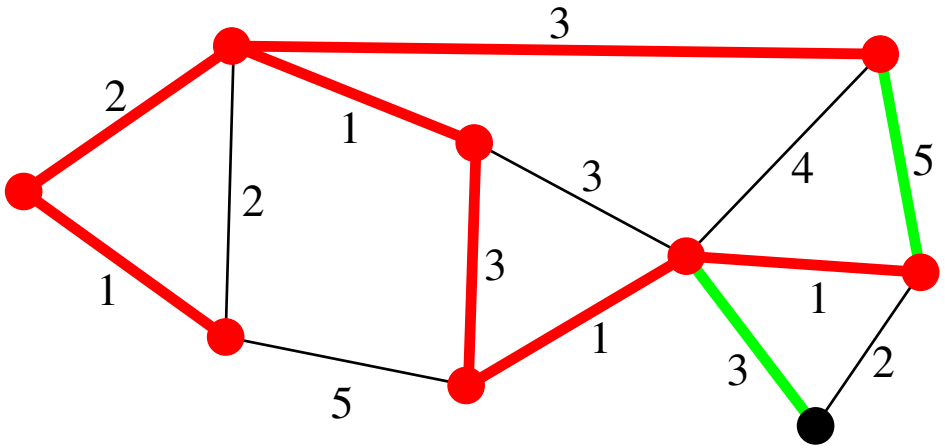
Exemple



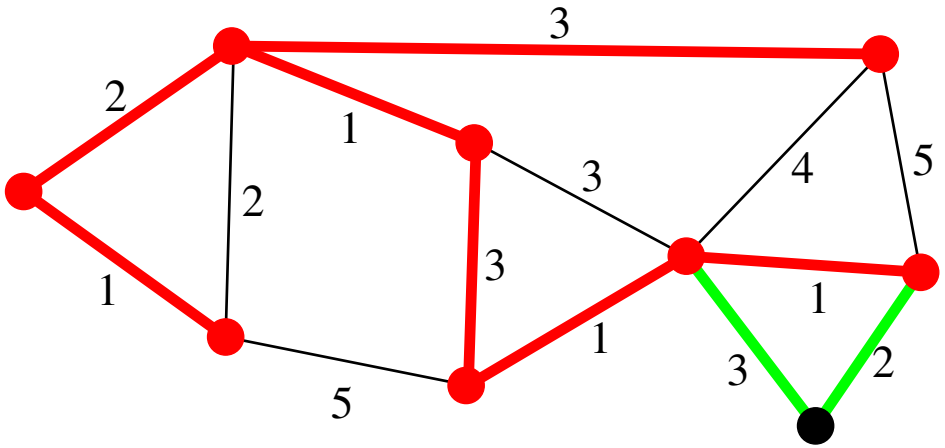
Exemple



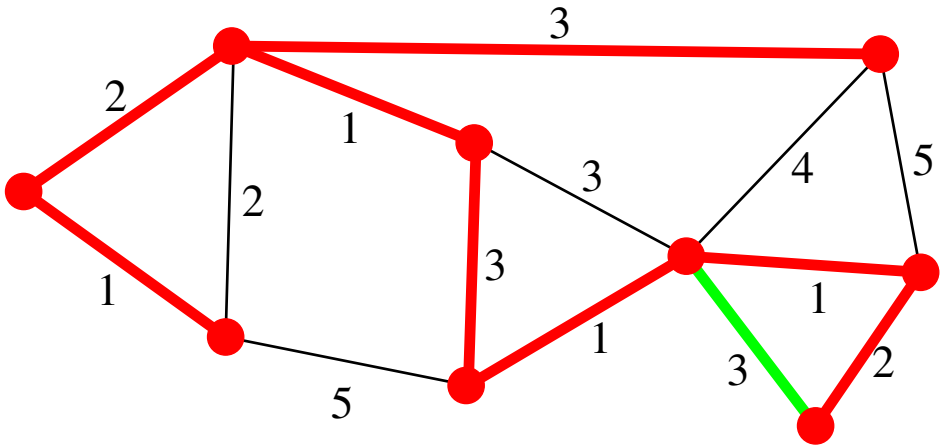
Exemple



Exemple



Exemple

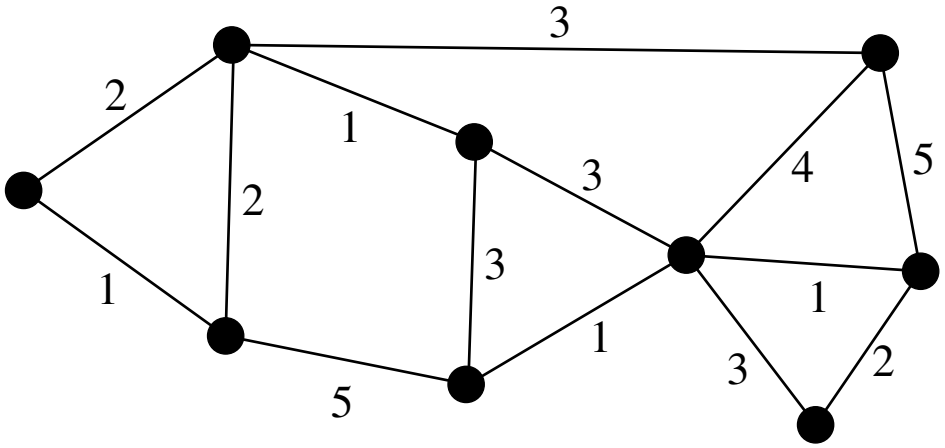


Algorithme de Kruskal (1956)

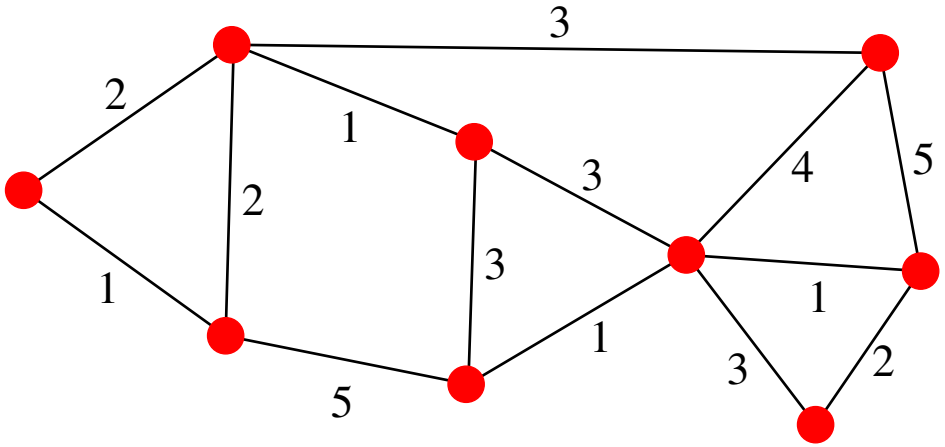
- On commence avec une forêt d'arbres constitués de chacun des sommets isolés.
- On itère en rajoutant à la forêt l'arête de poids le plus faible en faisant attention de ne pas créer de cycles.
- On stoppe quand on a examiné toutes les arêtes.

Kruskal
Entrée : graphe connexe G avec arêtes pondérées.
Sortie : un arbre couvrant de poids minimum.
On initialise : $\mathcal{F}$ avec tous les sommets et sans arêtes.
tant que il existe une arête à traiter **faire**
    On traite les arêtes de G l'une après l'autre par poids croissant
    si Une arête e permet de connecter deux composantes de $\mathcal{F}$ **alors**
        $\mathcal{F} := \mathcal{F} \bigcup \{e\}$
    fin si
fin tant que
retourner $\mathcal{F}$

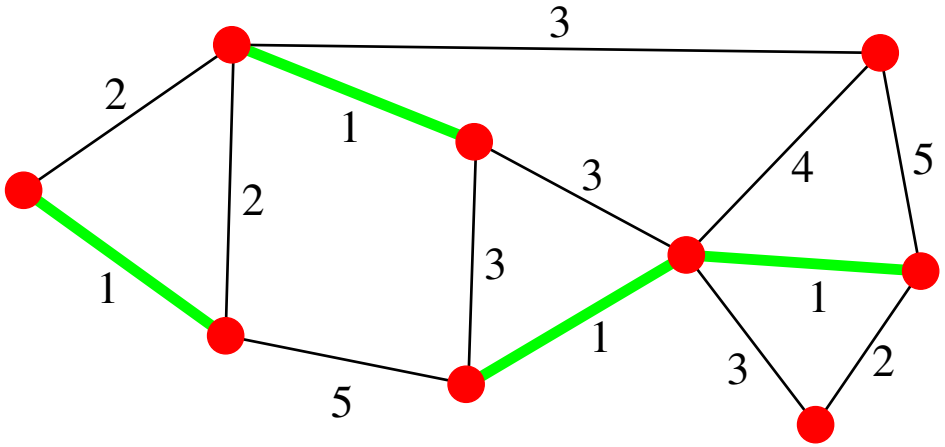
Exemple



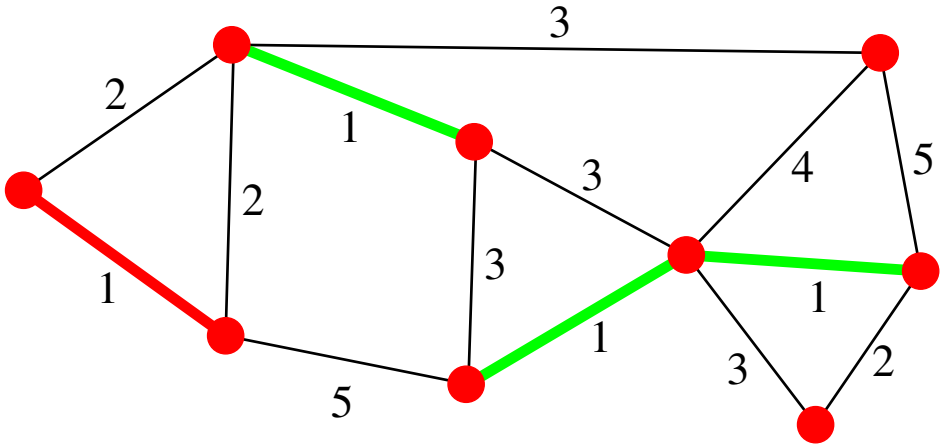
Exemple



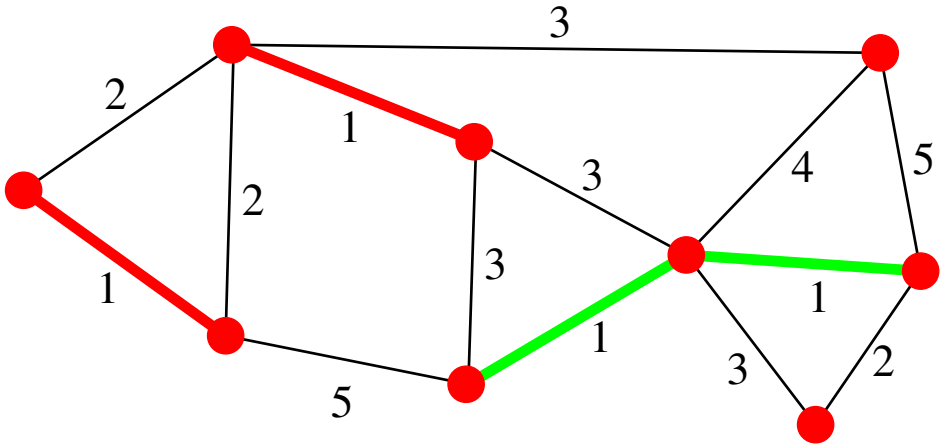
Exemple



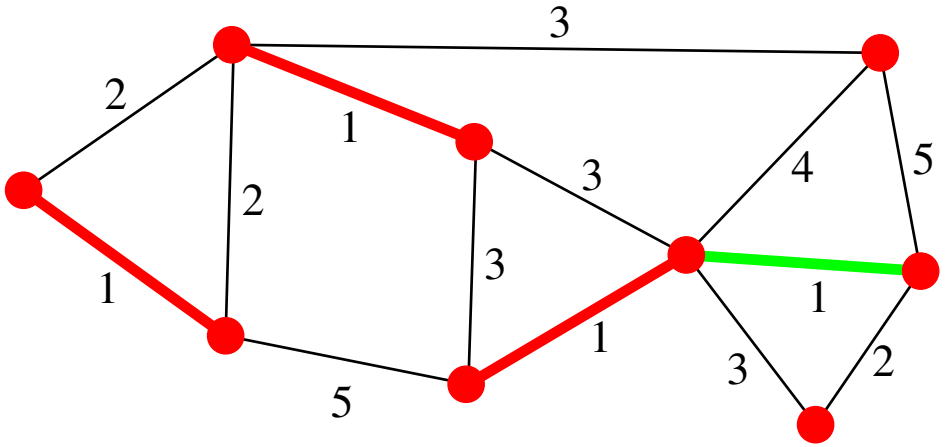
Exemple



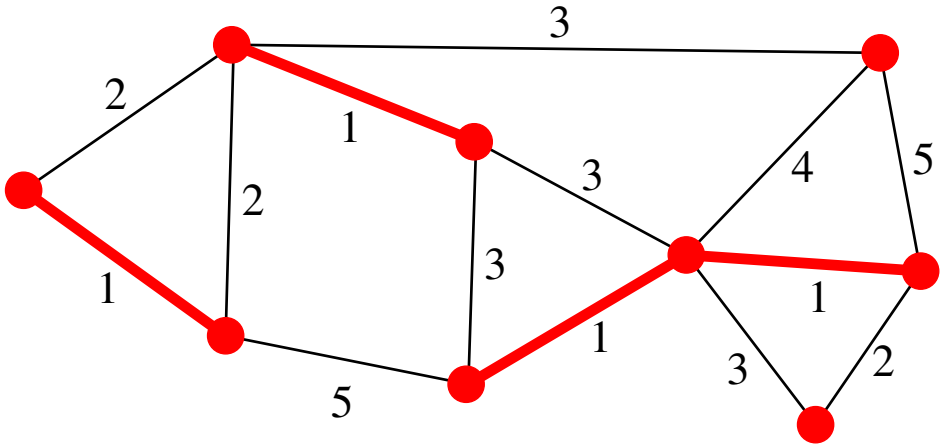
Exemple



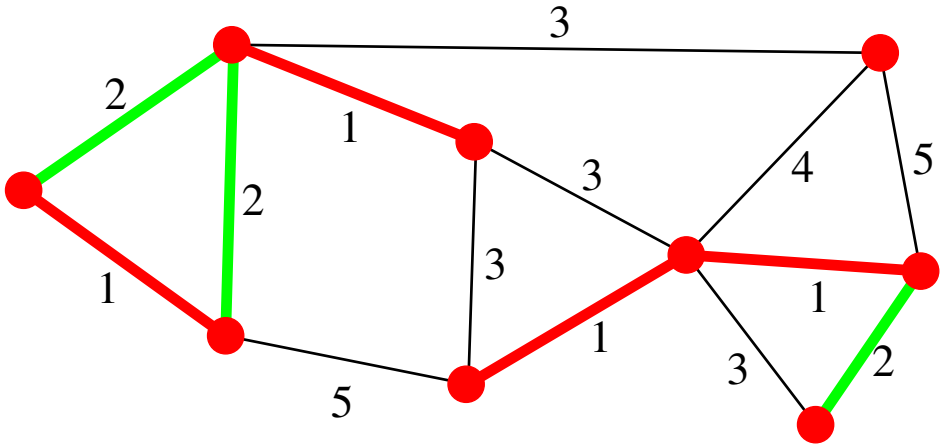
Exemple



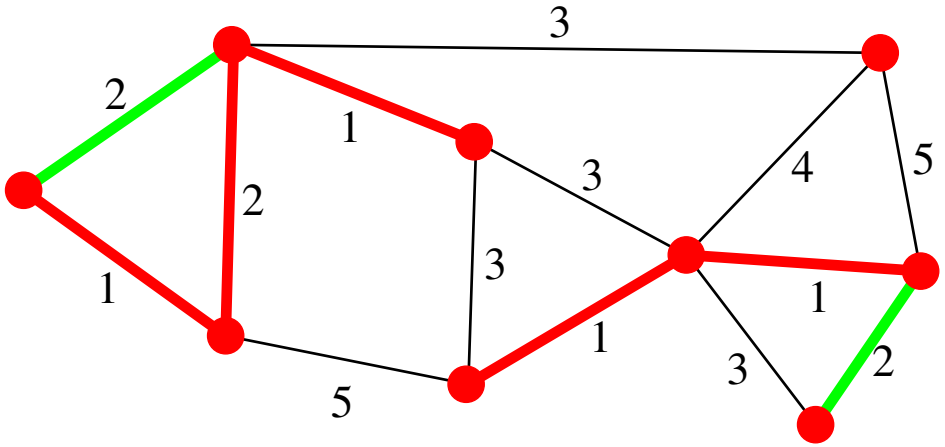
Exemple



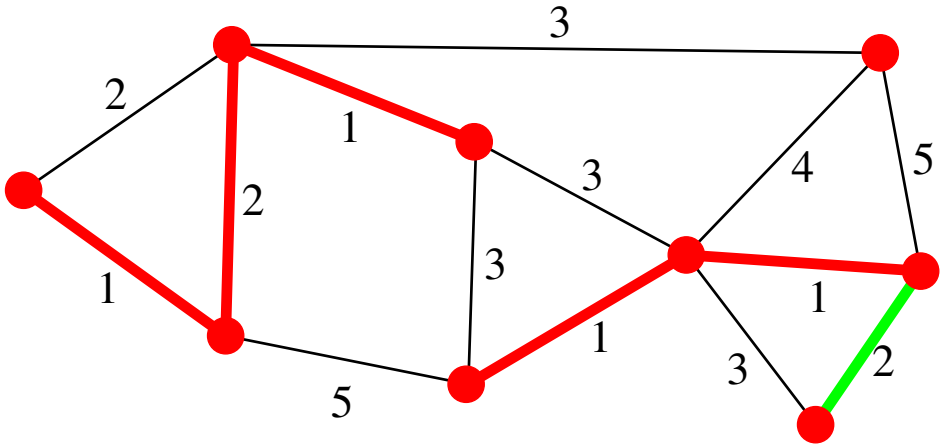
Exemple



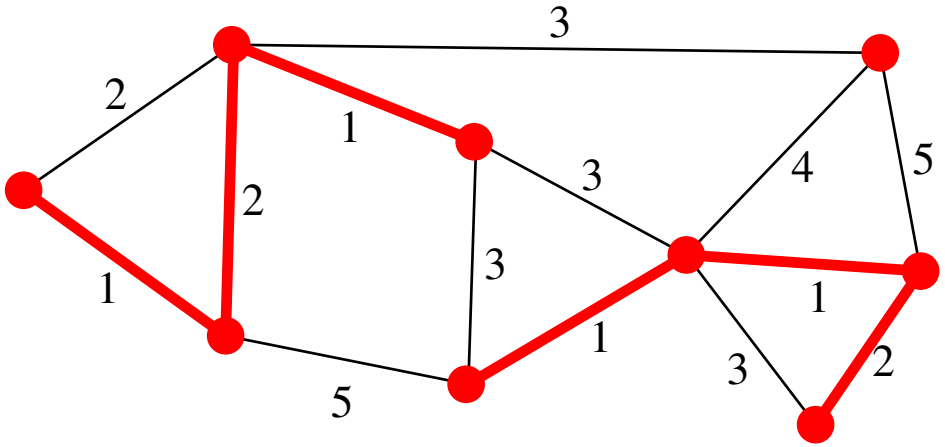
Exemple



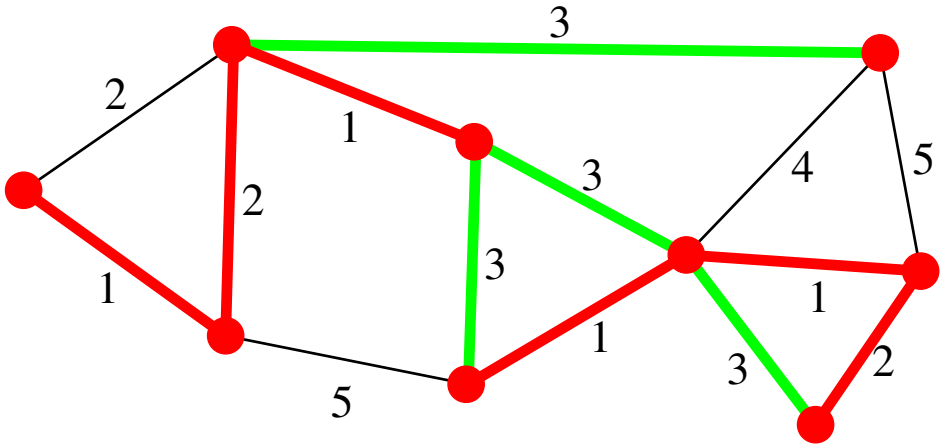
Exemple



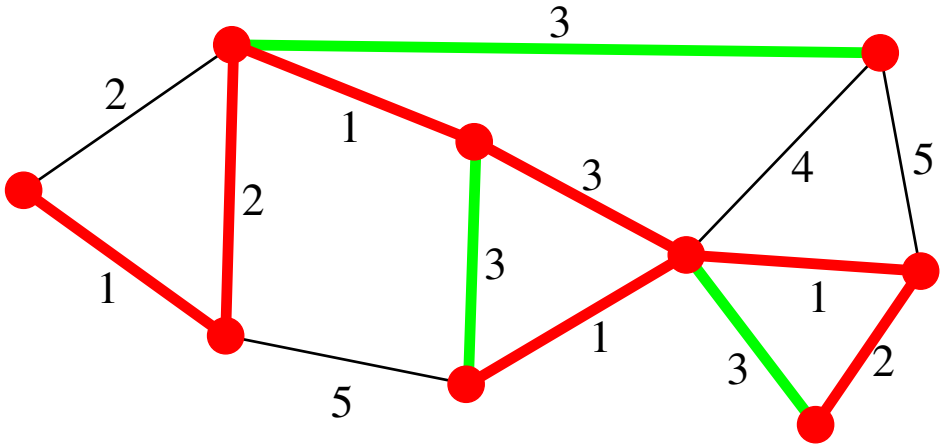
Exemple



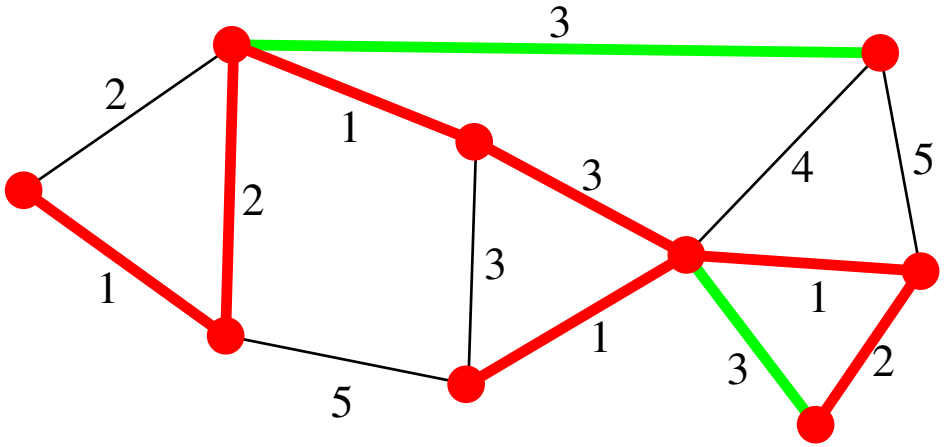
Exemple



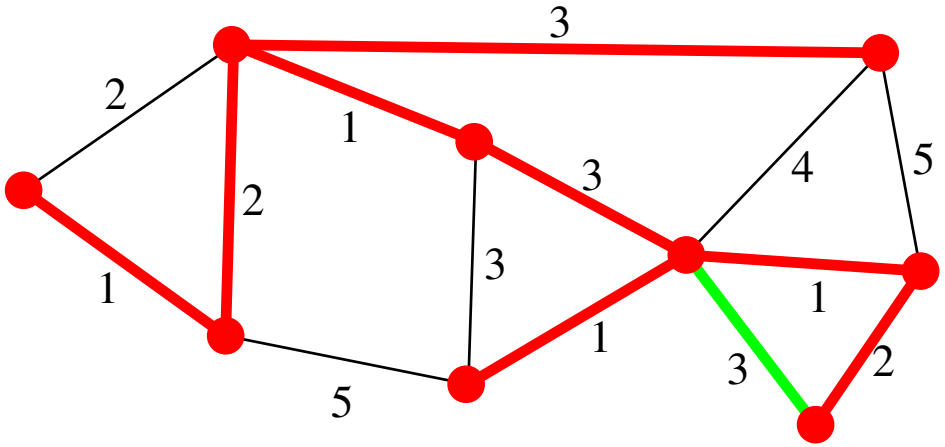
Exemple



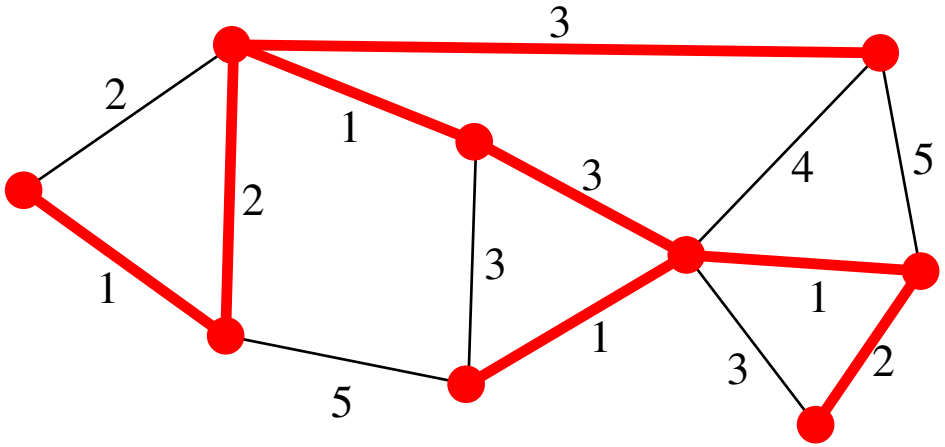
Exemple



Exemple



Exemple



A l'envers ?

Que fait l'algorithme "Devinette" suivant ?

Devinette
Entrée : graphe connexe G avec arêtes pondérées.
Sortie : $\mathcal{F}$?
On initialise : $\mathcal{F}$ avec toutes les arêtes de G
tant que $|F| > |S| - 1$ **faire**
    Choisir un cycle c
    Soit e l'arête de poids maximal de c
    $\mathcal{F} := \mathcal{F} - \{e\}$
fin tant que
retourner $\mathcal{F}$