Algorithmes gloutons

Quentin Fortier

July 2, 2022

Algorithme glouton

Un **algorithme glouton** effectue à chaque étape l'action qui semble la plus intéressante localement (sans revenir sur sa décision plus tard).

Algorithme glouton

Un **algorithme glouton** effectue à chaque étape l'action qui semble la plus intéressante localement (sans revenir sur sa décision plus tard).

Exemples :

- Algorithme glouton pour le sac à dos (non optimal)
- Algorithmes de Prim, Kruskal pour trouver un arbre couvrant de poids minimum
- Coloriage de graphe (non optimal a priori)

Sac à dos

Problème du sac à dos

On considère un sac à dos de capacité 10kg et les objets suivants :

poids (kg)	2	2	2	3	5	5	8
valeur (€)	1	1	1	7	10	10	13

Quelle est la valeur maximum que l'on peut mettre dans le sac ?

Sac à dos

Problème du sac à dos

On considère un sac à dos de capacité 10kg et les objets suivants :

poids (kg)	2	2	2	3	5	5	8
valeur (€)	1	1	1	7	10	10	13

Quelle est la valeur maximum que l'on peut mettre dans le sac ?

Algorithme glouton n°1 : ajouter les éléments dans l'ordre croissant de poids, tant que c'est possible

Sac à dos

Problème du sac à dos

On considère un sac à dos de capacité 10kg et les objets suivants :

poids (kg)	2	2	2	3	5	5	8
valeur (€)	1	1	1	7	10	10	13

Quelle est la valeur maximum que l'on peut mettre dans le sac ?

Algorithme glouton n°2 : ajouter les éléments dans l'ordre décroissant de valeur, tant que c'est possible

Sac à dos

Problème du sac à dos

On considère un sac à dos de capacité 10kg et les objets suivants :

poids (kg)	2	2	2	3	5	5	8
valeur (€)	1	1	1	7	10	10	13
valeur/poids	0.5	0.5	0.5	2.3	2	2	1.6

Quelle est la valeur maximum que l'on peut mettre dans le sac ?

Algorithme glouton n°3 : ajouter les éléments dans l'ordre décroissant de valeur/poids, tant que c'est possible

Sac à dos

Problème du sac à dos

On considère un sac à dos de capacité 10kg et les objets suivants :

poids (kg)	2	2	2	3	5	5	8
valeur (€)	1	1	1	7	10	10	13
valeur/poids	0.5	0.5	0.5	2.3	2	2	1.6

Quelle est la valeur maximum que l'on peut mettre dans le sac ?

Algorithme glouton n°3 : ajouter les éléments dans l'ordre décroissant de valeur/poids, tant que c'est possible

Ces méthodes gloutonnes ne sont pas optimales.

Sac à dos

On considère maintenant le problème du **sac à dos fractionnaire**, où il est possible de prendre une fraction d'un objet.

Sac à dos

On considère maintenant le problème du **sac à dos fractionnaire**, où il est possible de prendre une fraction d'un objet.

Théorème

L'algorithme glouton ajoutant les objets dans l'ordre décroissant de valeur/poids, et ajoutant une fraction du dernier objet (qui ne rentre pas en entier dans le sac) est optimal pour le problème du sac à dos fractionnaire.

Arbre couvrant de poids minimal

Arbre couvrant

Soit G un graphe pondéré (chaque arête e possède un poids $w(e)$).
Un arbre couvrant T de G est un ensemble d'arêtes de G qui forme un arbre et qui contient tous les sommets. Son poids $w(T)$ est la somme des poids des arêtes de l'arbre.

Arbre couvrant de poids minimal

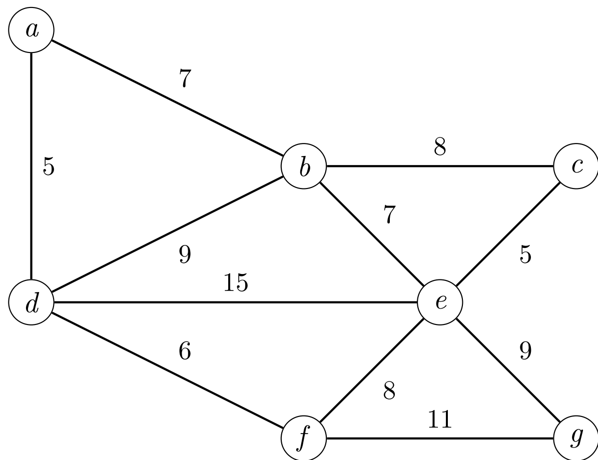
Arbre couvrant

Soit G un graphe pondéré (chaque arête e possède un poids $w(e)$).
Un arbre couvrant T de G est un ensemble d'arêtes de G qui forme un arbre et qui contient tous les sommets. Son poids $w(T)$ est la somme des poids des arêtes de l'arbre.

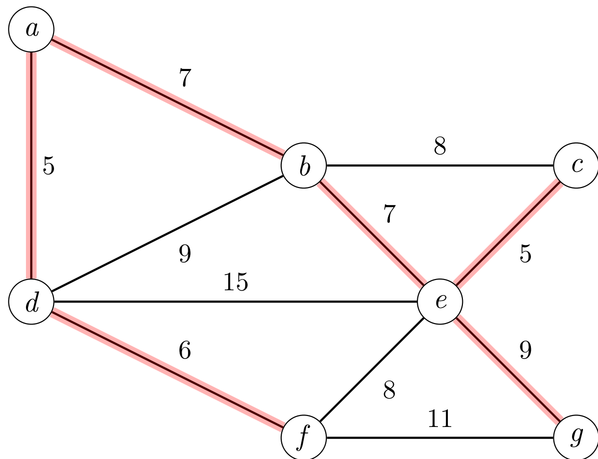
Arbre couvrant de poids minimal

Un arbre couvrant dont le poids est le plus petit possible est appelé un **arbre couvrant de poids minimal**.

Arbre couvrant de poids minimal



Arbre couvrant de poids minimal



Un arbre couvrant de poids minimum

Arbre couvrant de poids minimal

Deux algorithmes gloutons très connus permettent de trouver un arbre couvrant de poids minimum dans un graphe.
Ils ajoutent des arêtes une par une jusqu'à former un arbre couvrant de poids minimum.

Arbre couvrant de poids minimal

Deux algorithmes gloutons très connus permettent de trouver un arbre couvrant de poids minimum dans un graphe.
Ils ajoutent des arêtes une par une jusqu'à former un arbre couvrant de poids minimum.
Ils différent par le choix de l'arête à ajouter à chaque itération :

- **Kruskal** : ajoute la plus petite arête qui ne créé pas de cycle

Arbre couvrant de poids minimal

Deux algorithmes gloutons très connus permettent de trouver un arbre couvrant de poids minimum dans un graphe.
Ils ajoutent des arêtes une par une jusqu'à former un arbre couvrant de poids minimum.
Ils différent par le choix de l'arête à ajouter à chaque itération :

- **Kruskal** : ajoute la plus petite arête qui ne créé pas de cycle
- **Prim** : ajoute la plus petite arête qui conserve la connexité

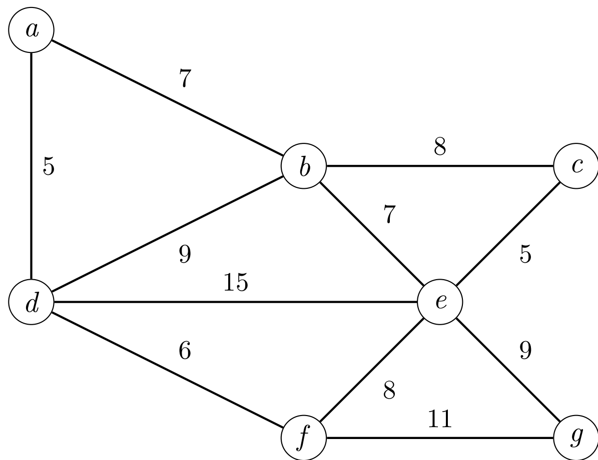
Kruskal

Algorithme de Kruskal :

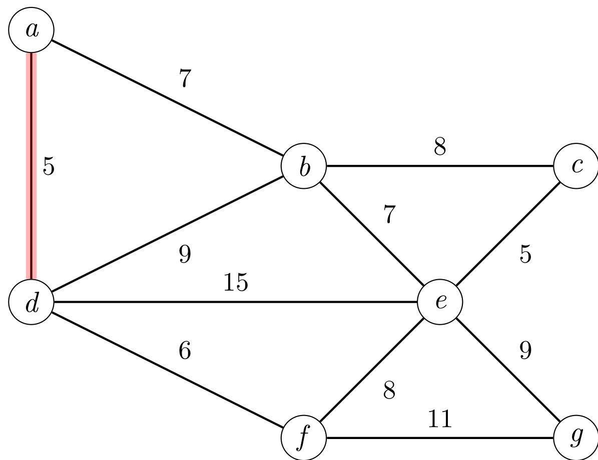
```
Trier les arêtes par poids croissant.
Commencer avec un arbre T vide (aucune arête).

Pour chaque arête e par poids croissant:
    Si l'ajout de e ne créé pas de cycle dans T:
        Ajouter e à T
```

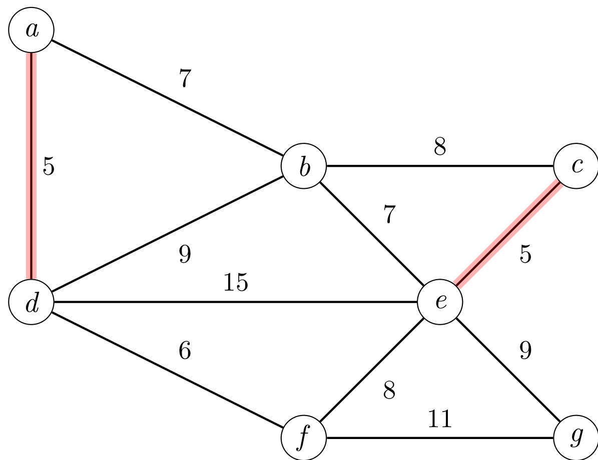
Kruskal



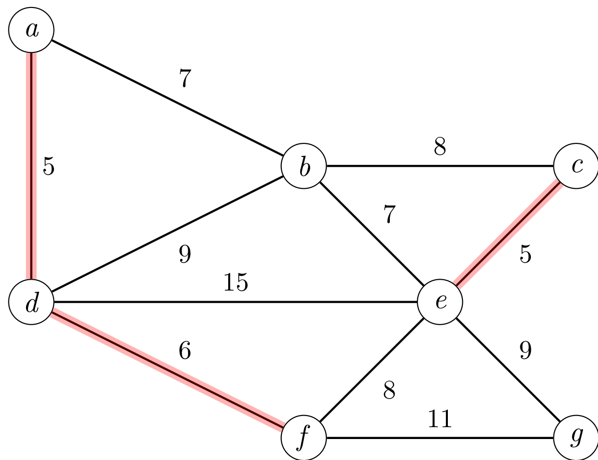
Kruskal



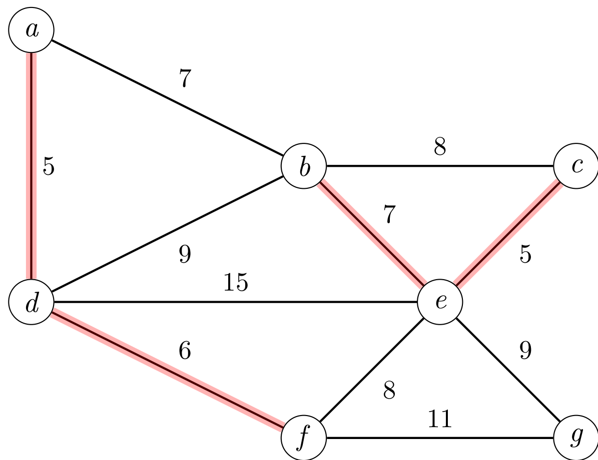
Kruskal



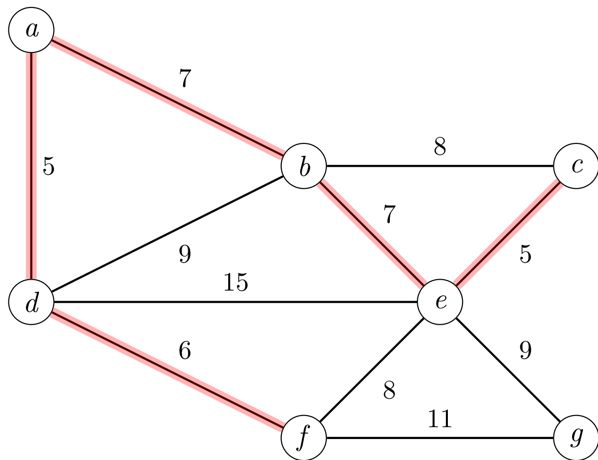
Kruskal



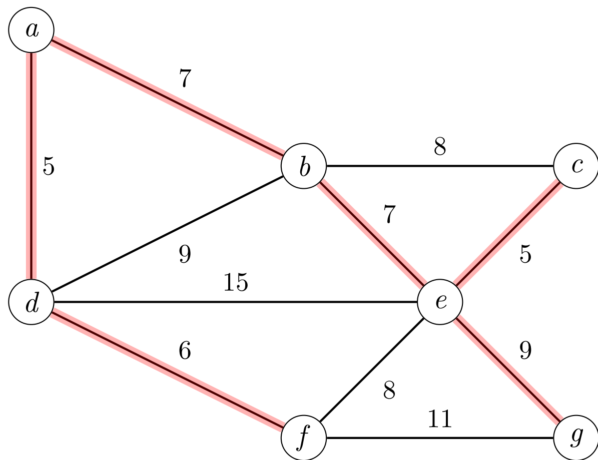
Kruskal



Kruskal



Kruskal



Kruskal

Théorème

L'algorithme de Kruskal sur un graphe G donne bien un arbre couvrant de poids minimum.

Preuve : Soit T l'arbre obtenu par Kruskal. Il faut montrer que :

1. T est un arbre couvrant.
2. T est de poids minimum.

Kruskal

T est un arbre couvrant :

1. $\underline{T \text{ est sans cycle}}$:

Kruskal

T est un arbre couvrant :

1. T est sans cycle : car l'algorithme ne créé pas de cycle
2. T est connexe (et couvrant) :

Kruskal

T est un arbre couvrant :

1. T est sans cycle : car l'algorithme ne créé pas de cycle
2. T est connexe (et couvrant) :
   Soit u et v deux sommets de G. Soit U l'ensemble des sommets accessibles depuis u dans T.

Kruskal

T est un arbre couvrant :

1. T est sans cycle : car l'algorithme ne créé pas de cycle

2. T est connexe (et couvrant) :
   Soit u et v deux sommets de G. Soit U l'ensemble des sommets accessibles depuis u dans T.
   Supposons $v \notin U$. Comme G est connexe, il existe une arête de G entre U et $V \setminus U$.

Kruskal

T est un arbre couvrant :

1. <u>T est sans cycle</u> : car l'algorithme ne créé pas de cycle

2. <u>T est connexe (et couvrant)</u> :
   Soit u et v deux sommets de G. Soit U l'ensemble des sommets accessibles depuis u dans T.
   Supposons $v \notin U$. Comme G est connexe, il existe une arête de G entre U et $V \setminus U$.
   Cette arête aurait dû être ajoutée à T, puisqu'elle ne créée pas de cycle.
   Contradiction : v est donc accessible depuis u dans T.
   Comme c'est vrai pour tout u, v, T est connexe.

Kruskal

Théorème

L'algorithme de Kruskal sur un graphe G donne un arbre couvrant de poids minimum.

Preuve :

Kruskal

Théorème

L'algorithme de Kruskal sur un graphe G donne un arbre couvrant de poids minimum.

Preuve : Soient T l'arbre obtenu par Kruskal et T^* un arbre de poids minimum.

Si $T = T^*$, le théorème est démontré.

Kruskal

Théorème

L'algorithme de Kruskal sur un graphe G donne un arbre couvrant de poids minimum.

Preuve : Soient T l'arbre obtenu par Kruskal et T^* un arbre de poids minimum.
Si $T = T^*$, le théorème est démontré.
Sinon, soit $e^* \in T^*$ une arête de poids min n'appartenant pas à T.

Kruskal

Théorème

L'algorithme de Kruskal sur un graphe G donne un arbre couvrant de poids minimum.

Preuve : Soient T l'arbre obtenu par Kruskal et T^* un arbre de poids minimum.
Si $T = T^*$, le théorème est démontré.
Sinon, soit $e^* \in T^*$ une arête de poids min n'appartenant pas à T.
Comme T est connexe, il existe un chemin C dans T reliant les extrémités de e^*.

Kruskal

Théorème

L'algorithme de Kruskal sur un graphe G donne un arbre couvrant de poids minimum.

Preuve : Soient T l'arbre obtenu par Kruskal et T^* un arbre de poids minimum.
Si $T = T^*$, le théorème est démontré.
Sinon, soit $e^* \in T^*$ une arête de poids min n'appartenant pas à T.
Comme T est connexe, il existe un chemin C dans T reliant les extrémités de e^*.

- Il existe une arête e de C qui n'est pas dans T^*

Kruskal

Théorème

L'algorithme de Kruskal sur un graphe G donne un arbre couvrant de poids minimum.

Preuve : Soient T l'arbre obtenu par Kruskal et T^* un arbre de poids minimum.
Si $T = T^*$, le théorème est démontré.
Sinon, soit $e^* \in T^*$ une arête de poids min n'appartenant pas à T.
Comme T est connexe, il existe un chemin C dans T reliant les extrémités de e^*.

- Il existe une arête e de C qui n'est pas dans T^* car T^* ne peut pas contenir de cycle
- $w(e) \leq w(e^*)$ (sinon Kruskal aurait ajouté e^* à T)

Kruskal

Considérons $T_2 = T \setminus \{e\} \cup \{e^*\}$.

Kruskal

Considérons $T_2 = T \setminus \{e\} \cup \{e^*\}$.

- T_2 est un arbre couvrant

Kruskal

Considérons $T_2 = T \setminus \{e\} \cup \{e^*\}$.

- T_2 est un arbre couvrant
- $w(T) \leq w(T_2)$

Kruskal

Considérons $T_2 = T \setminus \{e\} \cup \{e^*\}$.

- T_2 est un arbre couvrant
- $w(T) \leq w(T_2)$

On répète le même processus sur T_2, ce qui nous donne T_3, T_4... jusqu'à obtenir T^* :

$$w(T) \leq w(T_2) \leq w(T_3) \leq ... \leq w(T^*)$$

Comme T est un arbre couvrant et $w(T) \leq w(T^*)$, on a en fait $w(T) = w(T^*)$ et T **est un arbre couvrant de poids minimum**.

Kruskal : Implémentation et complexité

On va avoir besoin d'un graphe pondéré :

```
type ('v, 'w) graph = {
    n : int; (* nombre de sommets *)
    add_edge : 'v -> 'v -> 'w -> unit;
    del_edge : 'v -> 'v -> unit;
    edges : unit -> ('v*'v) list; (* toutes les arêtes *)
    w : 'v -> 'v -> 'w option; (* poids d'une arête *)
    adj : 'v -> 'v list (* liste des sommets adjacents *)
}
```

On suppose de plus avoir une fonction créant un nouveau graphe à n sommets et sans arête :

```
create_graph : int -> ('v, 'w) graph
```

Kruskal : Implémentation et complexité

Complexité de Kruskal sur un graphe à n sommets et p arêtes :

1. Trier les arêtes par poids croissants :

Kruskal : Implémentation et complexité

Complexité de Kruskal sur un graphe à n sommets et p arêtes :

1. Trier les arêtes par poids croissants : $O(p\log(p))$
2. Pour chaque arête $\{u, v\}$, déterminer si l'ajout de cette arête créé un cycle.

Pour détecter un cycle :

- DFS/BFS

Kruskal : Implémentation et complexité

Complexité de Kruskal sur un graphe à n sommets et p arêtes :

1. Trier les arêtes par poids croissants : $\mathrm{O}(p\log(p))$
2. Pour chaque arête $\{u, v\}$, déterminer si l'ajout de cette arête créé un cycle.

Pour détecter un cycle :

- DFS/BFS en $\mathrm{O}(n+p) \longrightarrow$ Complexité $\mathrm{O}(p(n+p))$ pour Kruskal
- **Union-Find** en $\mathrm{O}(1) \longrightarrow$ Complexité $\boxed{\mathrm{O}(p\log(p))}$ pour Kruskal

Kruskal : Union-Find

Union-Find est une structure de donnée permettant de représenter des classes (d'équivalences) :

- Chaque élément appartient à une classe
- Chaque classe possède un représentant, qui est un élément particulier de cette classe

Kruskal : Union-Find

On va utiliser les opérations suivantes d'Union-Find :

1. `create : int -> 'a unionfind`
   → `create n` renvoie une valeur de type `unionfind` avec `n` éléments, chaque élément étant seul dans sa classe

Kruskal : Union-Find

On va utiliser les opérations suivantes d'Union-Find :

1. `create : int -> 'a unionfind`
   → `create n` renvoie une valeur de type `unionfind` avec `n` éléments, chaque élément étant seul dans sa classe
2. `find : 'a unionfind -> 'a -> 'a`
   → `find uf v` donne le représentant de l'élément `v`

Kruskal : Union-Find

On va utiliser les opérations suivantes d'Union-Find :

1. `create : int -> 'a unionfind`
   → `create n` renvoie une valeur de type `unionfind` avec `n` éléments, chaque élément étant seul dans sa classe
2. `find : 'a unionfind -> 'a -> 'a`
   → `find uf v` donne le représentant de l'élément `v`
3. `union : 'a unionfind -> 'a -> 'a -> unit`
   → `union uf u v` fusionne les classes de `u` et `v` en une seule

Kruskal : Union-Find

On va utiliser les opérations suivantes d'Union-Find :

1. `create : int -> 'a unionfind`
   → `create n` renvoie une valeur de type `unionfind` avec `n` éléments, chaque élément étant seul dans sa classe
2. `find : 'a unionfind -> 'a -> 'a`
   → `find uf v` donne le représentant de l'élément `v`
3. `union : 'a unionfind -> 'a -> 'a -> unit`
   → `union uf u v` fusionne les classes de `u` et `v` en une seule

Kruskal : Union-Find

On va utiliser les opérations suivantes d'Union-Find :

1. `create : int -> 'a unionfind`
   → `create n` renvoie une valeur de type `unionfind` avec `n` éléments, chaque élément étant seul dans sa classe
2. `find : 'a unionfind -> 'a -> 'a`
   → `find uf v` donne le représentant de l'élément `v`
3. `union : 'a unionfind -> 'a -> 'a -> unit`
   → `union uf u v` fusionne les classes de `u` et `v` en une seule

Implémentation efficace (exercice à faire) : utiliser une forêt, avec un arbre pour chaque classe, enraciné en le représentant.

Kruskal : Union-Find

Utilisation d'un Union-Find pour construire un arbre T (qu'on construit en ajoutant les arêtes une par une) dans Kruskal :

- Chaque classe de l'Union-Find correspond à une composante connexe dans T.

Kruskal : Union-Find

Utilisation d'un Union-Find pour construire un arbre T (qu'on construit en ajoutant les arêtes une par une) dans Kruskal :

- Chaque classe de l'Union-Find correspond à une composante connexe dans T.
- Si u et v sont dans la même classe (`find uf u = find uf v`) alors l'ajout de l'arête $\{u, v\}$ à T créerait un cycle.

Kruskal : Union-Find

Utilisation d'un Union-Find pour construire un arbre T (qu'on construit en ajoutant les arêtes une par une) dans Kruskal :

- Chaque classe de l'Union-Find correspond à une composante connexe dans T.
- Si u et v sont dans la même classe (`find uf u = find uf v`) alors l'ajout de l'arête $\{u, v\}$ à T créerait un cycle.
- Sinon, ajouter l'arête à T et fusionner les classes de u et v (`union uf u v`).

Kruskal : Union-Find

```
let kruskal g = (* renvoie la liste des arêtes d'un
                   arbre couvrant de poids min de g *)
    let uf = create () in
    let rec aux = function
        | [] -> []
        | (u, v)::q when find uf u = find uf v -> aux q
        | (u, v)::q -> (union uf u v; (u, v)::aux q) in
    g.edges ()
    |> List.sort (fun e f -> g.w e - g.w f)
    |> aux
```

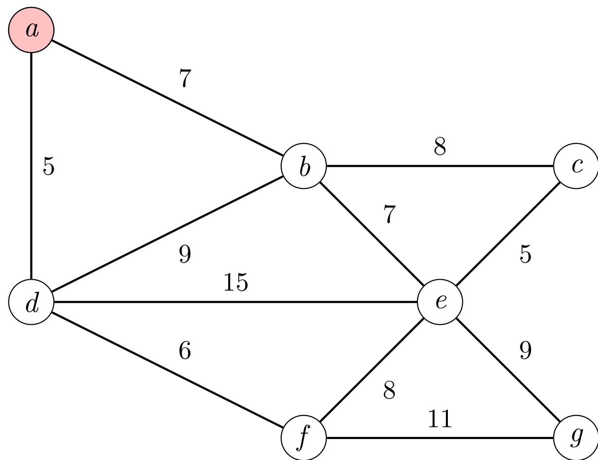
Prim

Algorithme de Prim :

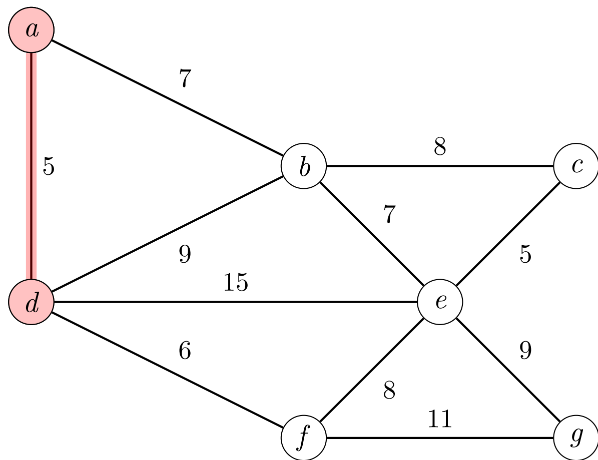
```
Commencer avec un arbre T contenant un seul sommet.

Tant que T ne contient pas tous les sommets:
    Ajouter l'arête sortante de T de poids minimum
```

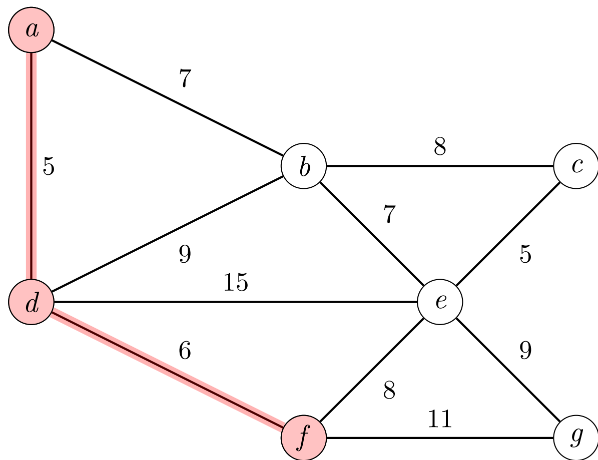
Prim



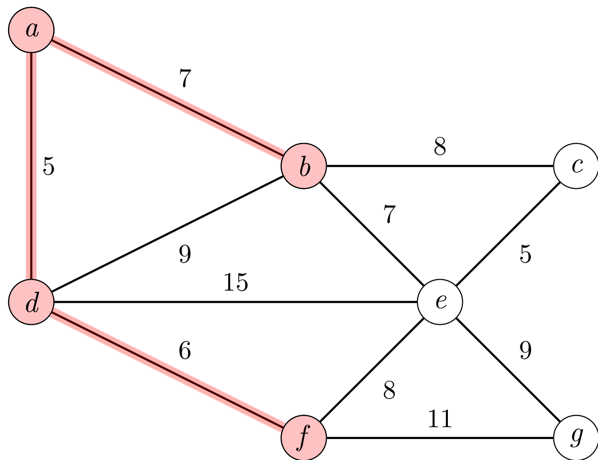
Prim



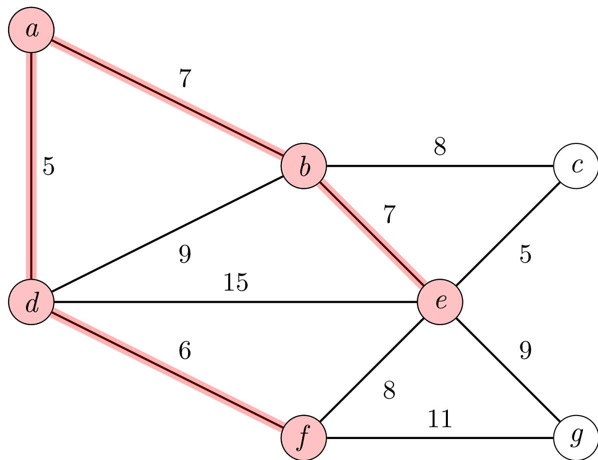
Prim



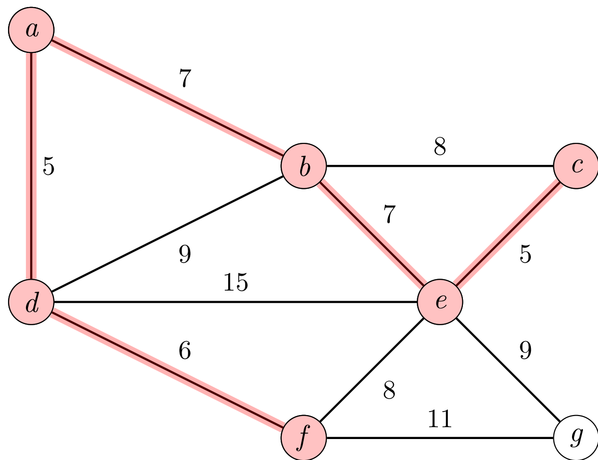
Prim



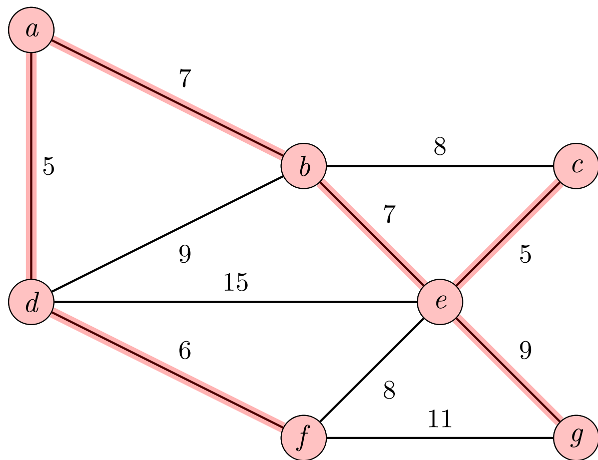
Prim



Prim



Prim



Prim

Théorème

L'algorithme de Prim sur un graphe G donne bien un arbre couvrant de poids minimum.

Preuve : Laissée en exercice.

Prim : Implémentation

Pour implémenter Prim, on utilise une **file de priorité** `pq` :

- `pq` contient les arêtes sortantes de T
- les éléments de `pq` sont ordonnés par poids croissants : `take_min pq` renvoie donc l'arête sortante de T de poids minimum