Arbre couvrant de poids minimum et algorithme de Kruskal

Quentin Fortier

November 17, 2022

Arbre couvrant de poids minimum

Arbre couvrant

Soit G un graphe connexe et pondéré par w.
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 minimum

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

Arbre couvrant de poids minimum

Lemme

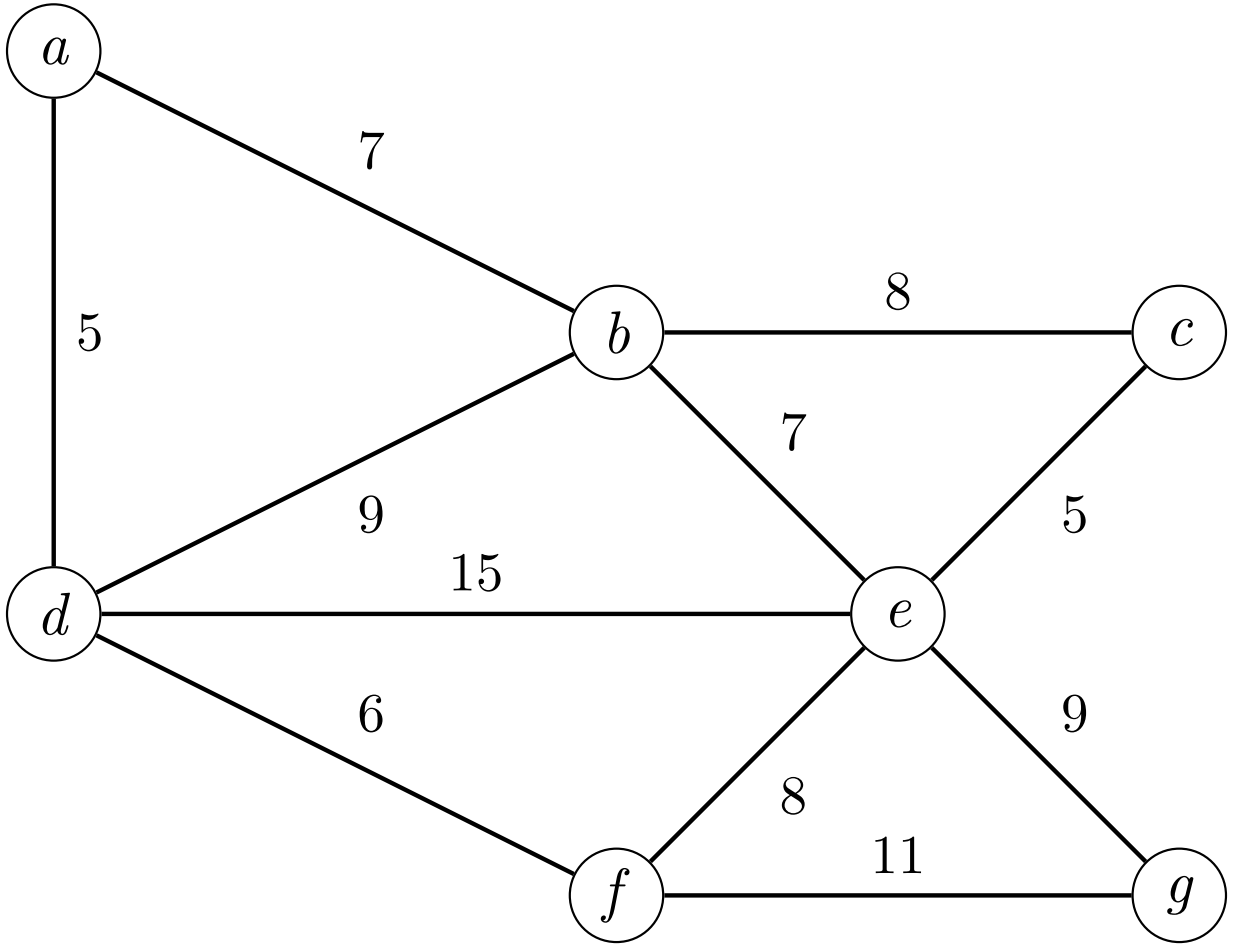
Soit G un graphe connexe et pondéré par w.
Alors G possède un arbre couvrant de poids minimum.

Preuve : Soit $E = \{w(T) \mid T \text{ est un arbre couvrant}\}$.

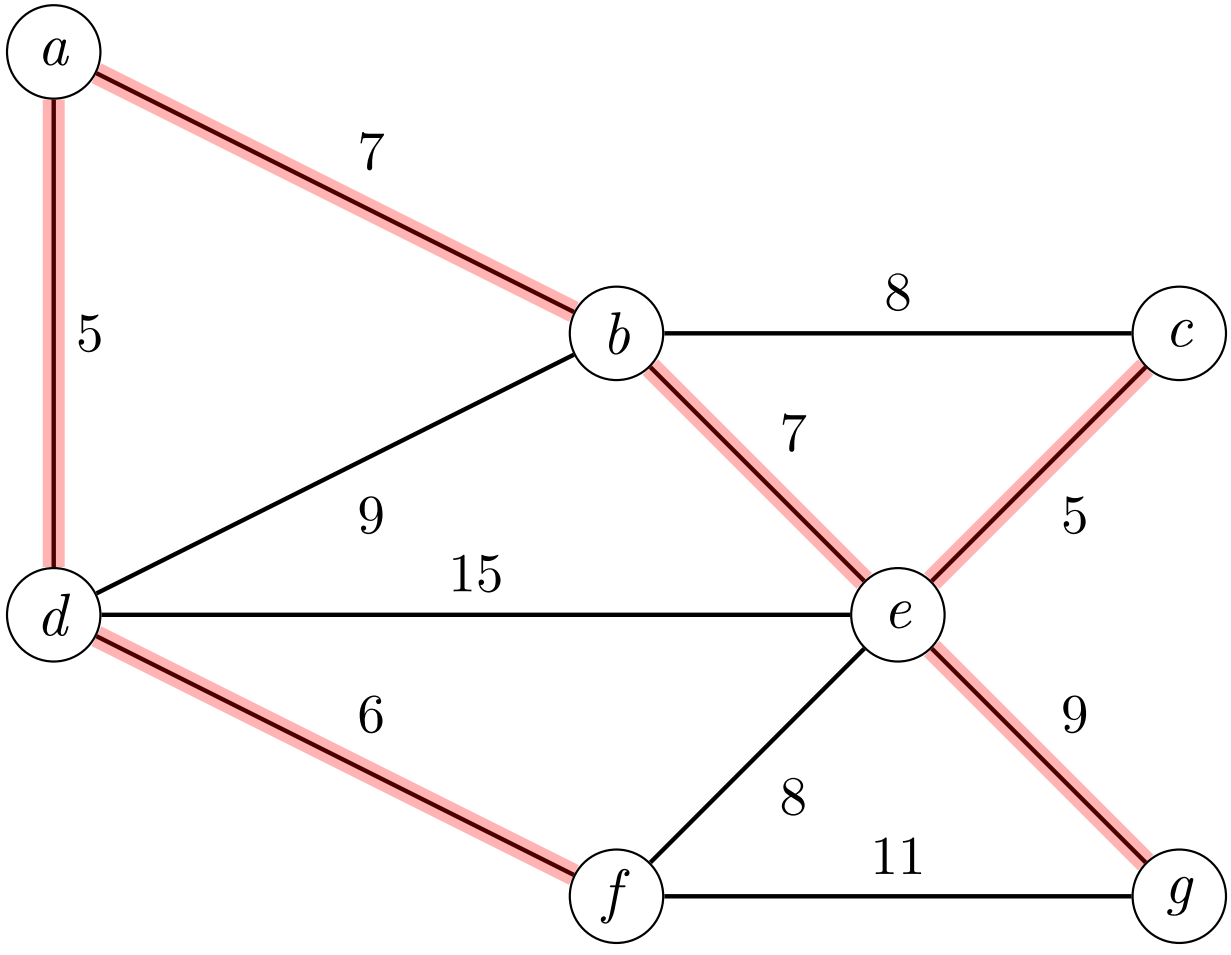
- $E \neq \emptyset$: l'ensemble des arêtes parcourues par un DFS (ou BFS) est un arbre couvrant de G, car G est connexe.
- $E \subseteq \mathbb{N}$

Donc E admet bien un minimum.

Arbre couvrant de poids minimum



Arbre couvrant de poids minimum



Un arbre couvrant de poids minimum

Arbre couvrant de poids minimum

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** (HP) : ajoute la plus petite arête qui conserve la connexité

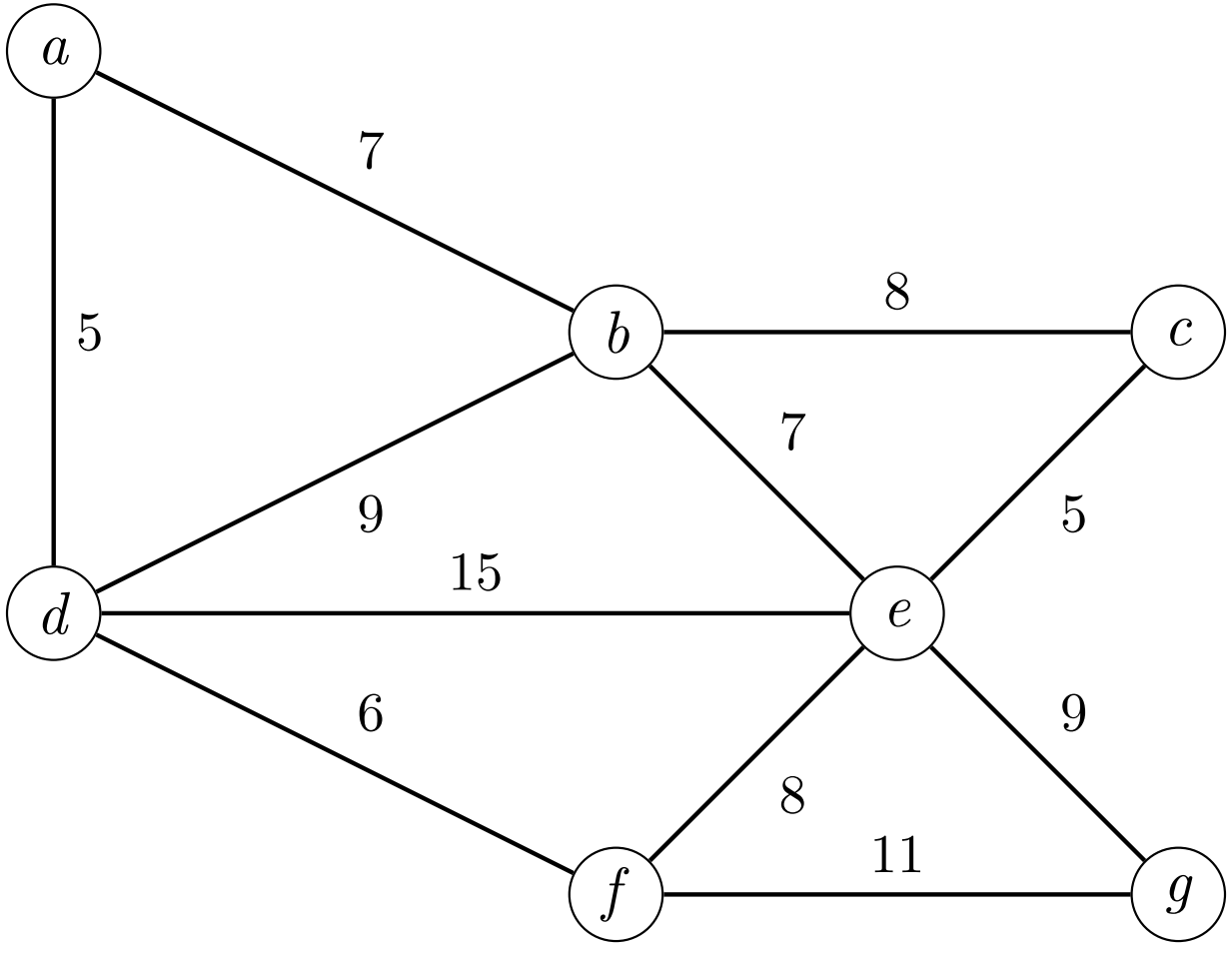
Kruskal

Algorithme de Kruskal sur un graphe connexe $G = (V, E)$:

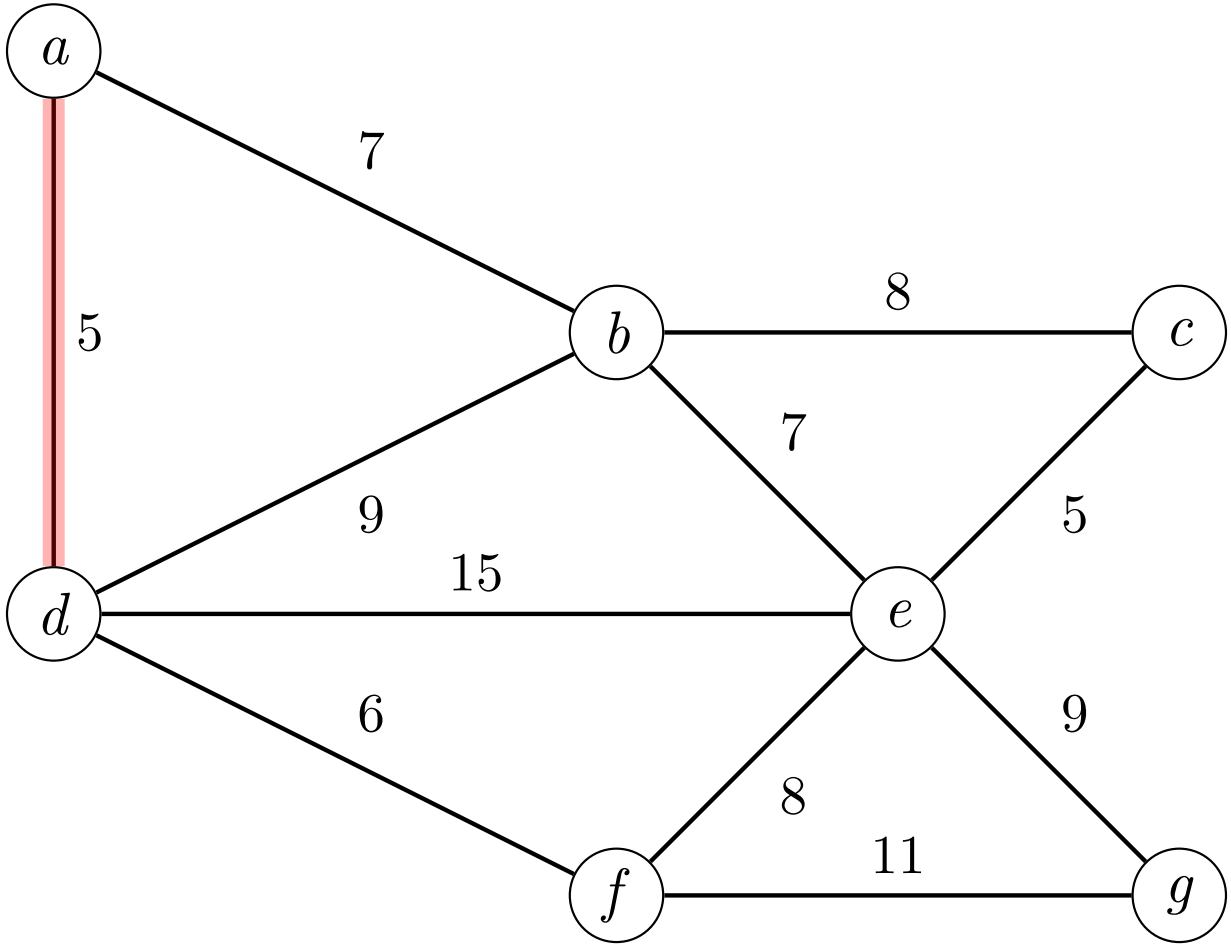
```
Trier les arêtes E par poids croissant.
T = arbre vide (aucune arête).

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

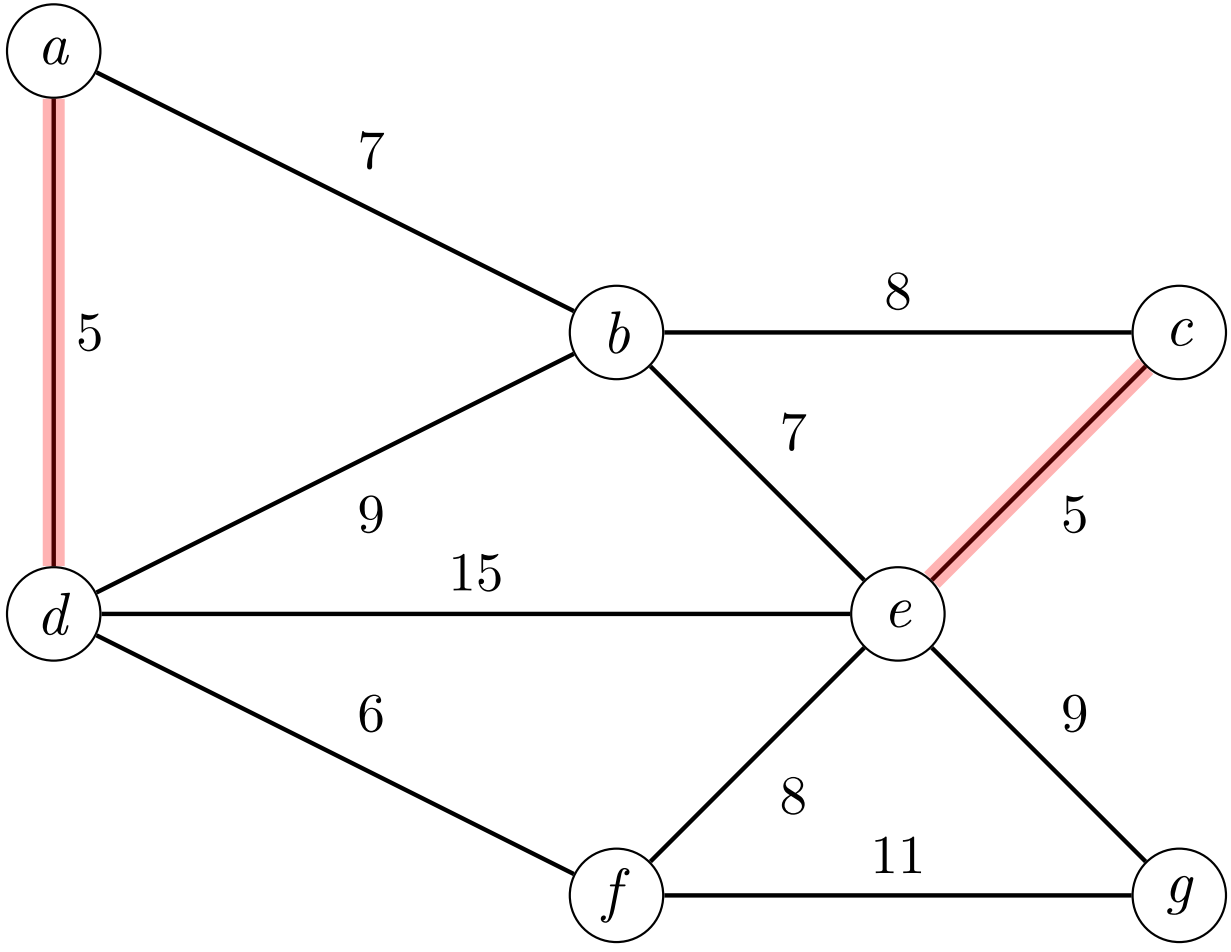
Kruskal



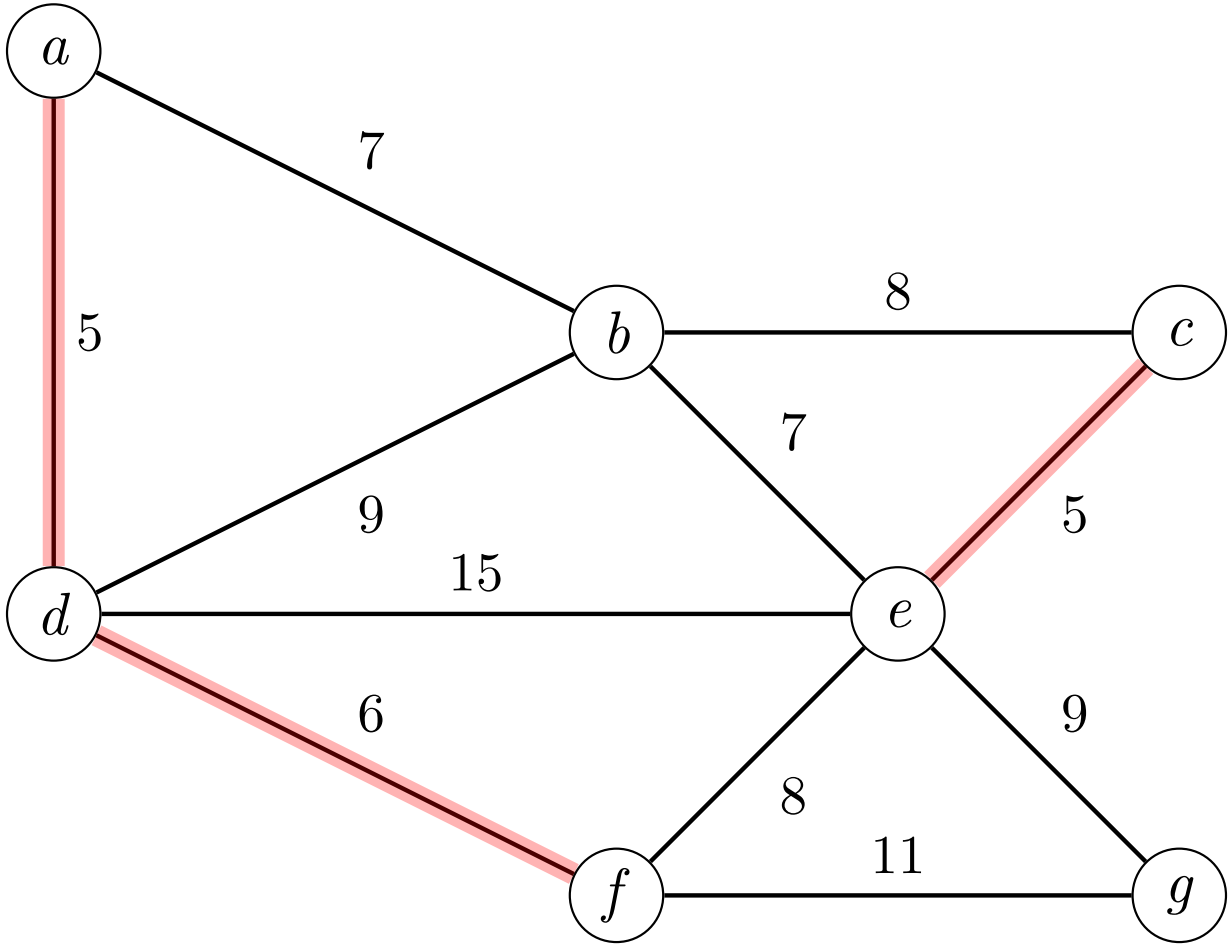
Kruskal



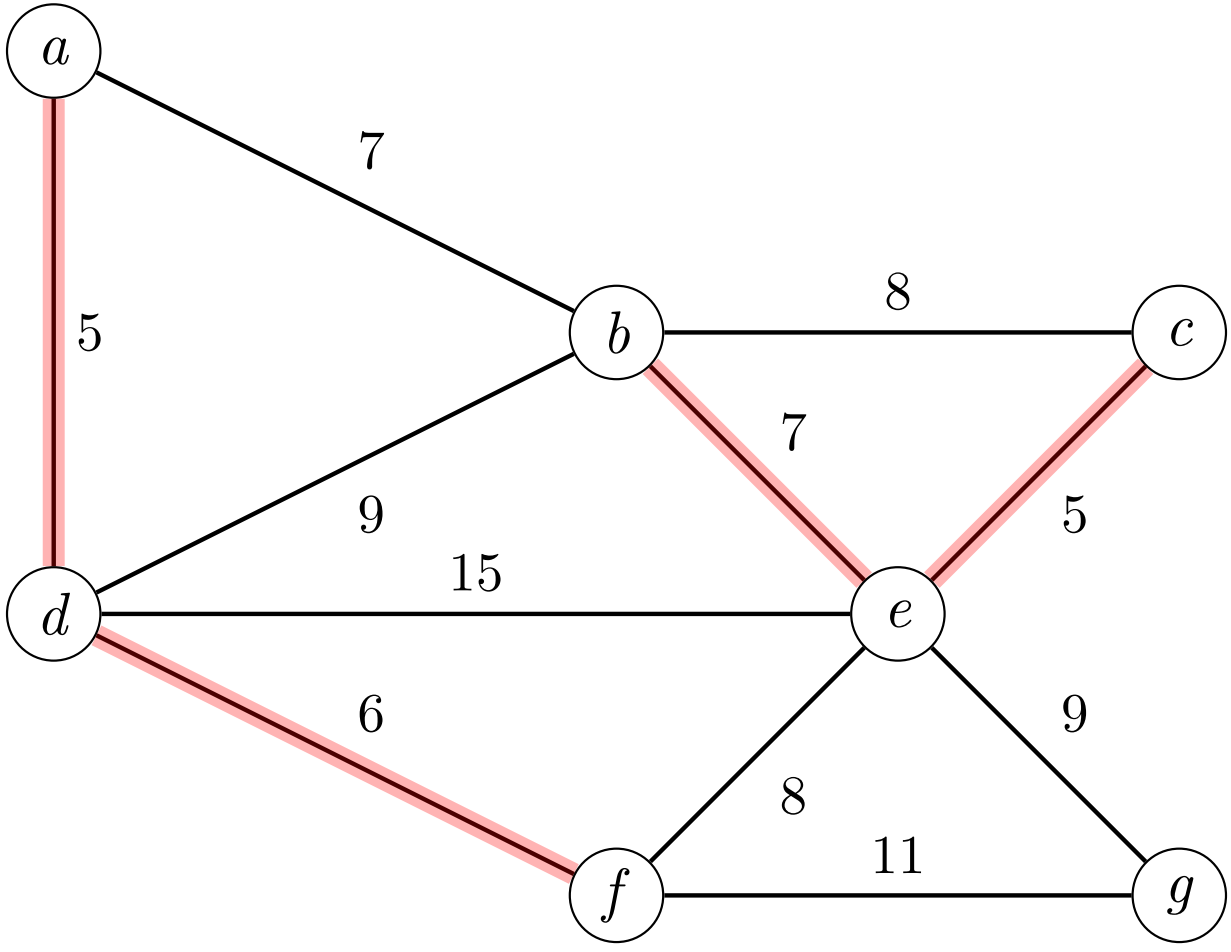
Kruskal



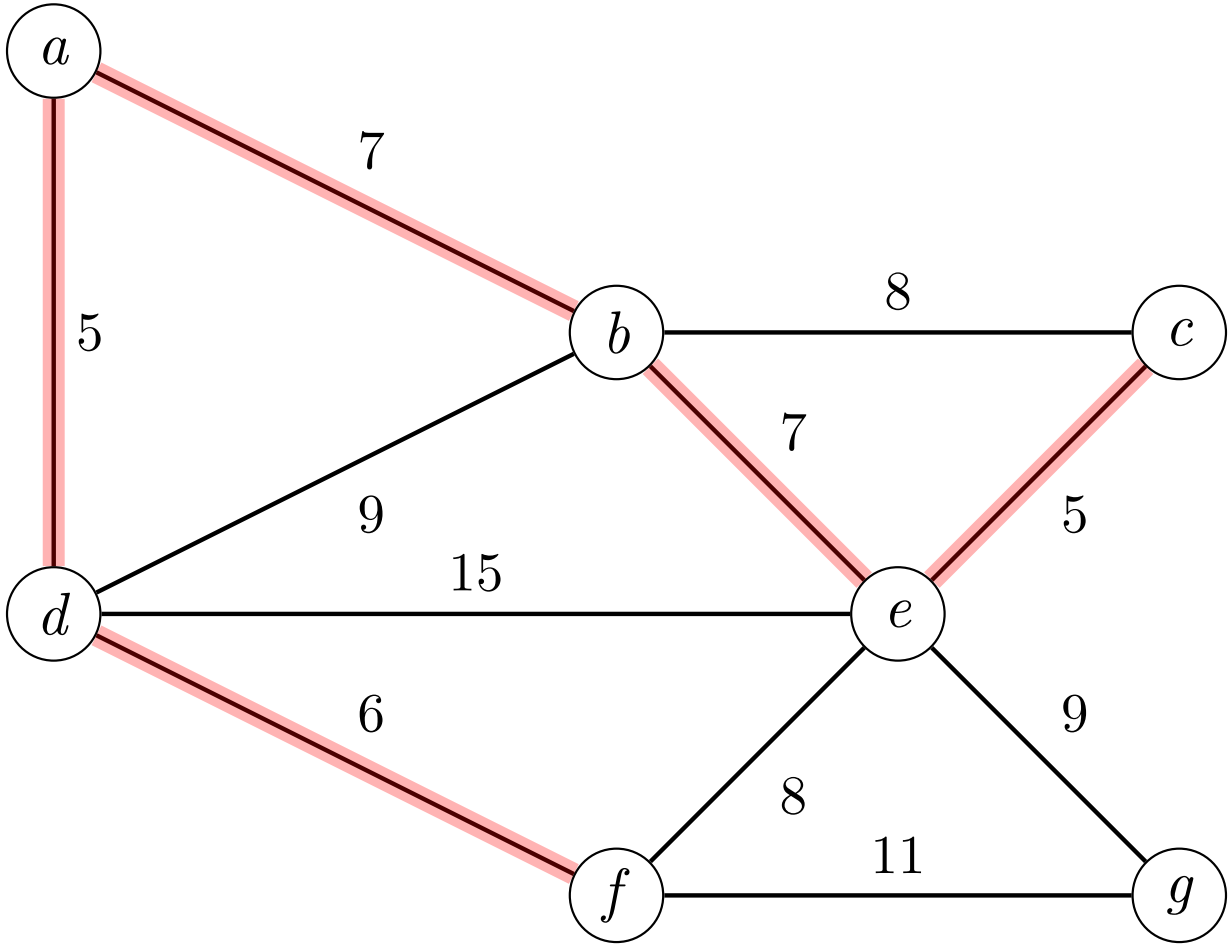
Kruskal



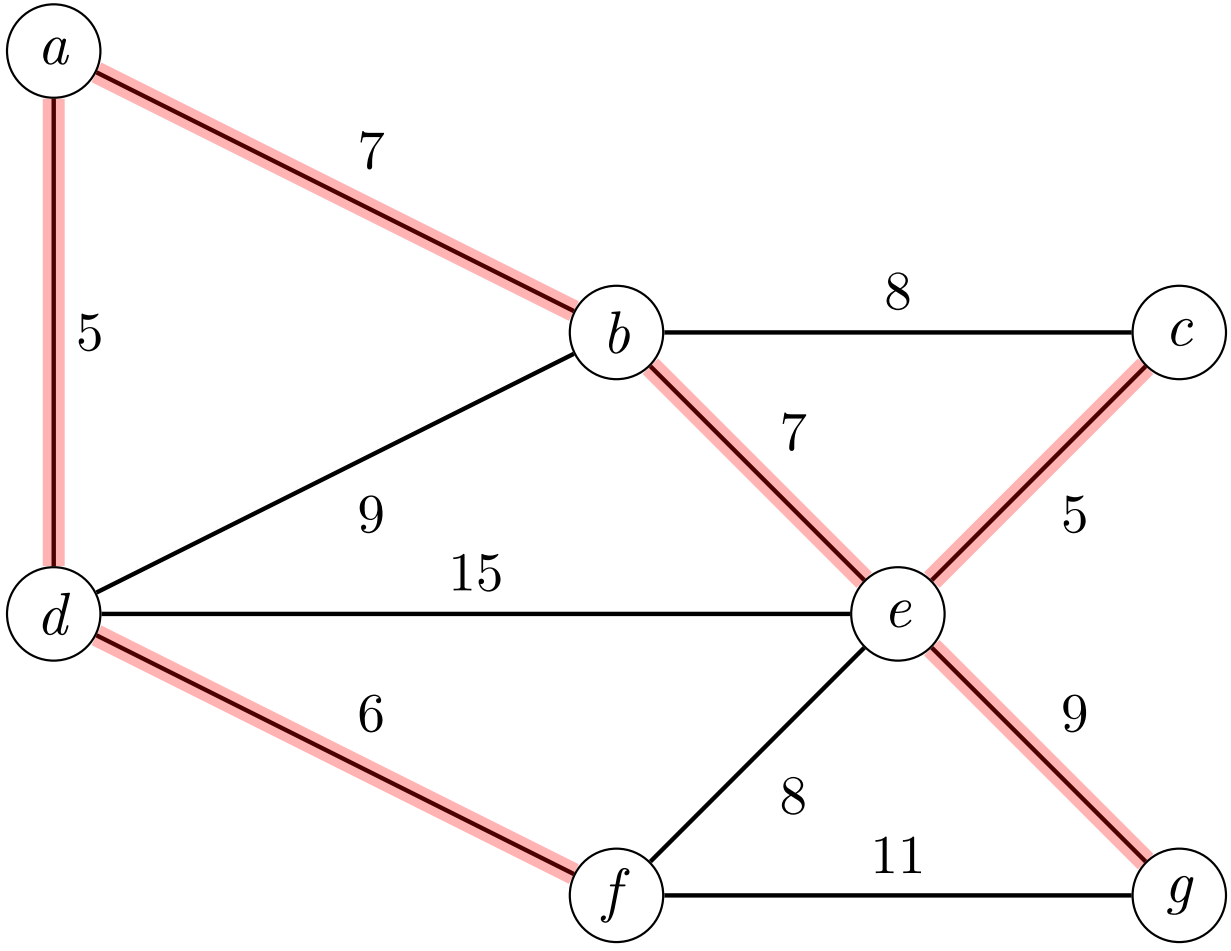
Kruskal



Kruskal



Kruskal



Kruskal

Théorème

L'algorithme de Kruskal sur un graphe connexe 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

Montrons que T est un arbre couvrant :

1. T est sans cycle :

Kruskal

Montrons que 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 $V \setminus U$.
   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 : 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 utiliser un graphe pondéré implémenté par matrice d'adjacence.

Kruskal : Implémentation et complexité

Exercice

Écrire une fonction `aretes : int list array -> (int*int) list` telle que `aretes g` renvoie la liste des arêtes du graphe `g` représenté par liste d'adjacence.

Kruskal : Implémentation et complexité

Exercice

Écrire une fonction `aretes : int list array -> (int*int) list` telle que `aretes g` renvoie la liste des arêtes du graphe g représenté par liste d'adjacence.

```
let rec couples i l = match l with
    | [] -> []
    | e::q -> (i, e)::couples q

let aretes g =
    let rec aux i =
        if i = Array.length g then []
        else (couples i g.(i))::aux (i + 1) in
    aux 0
```

Kruskal : Implémentation et complexité

Exercice

Écrire une fonction `tri_aretes g w` renvoyant la liste triée par poids `w` croissant des arêtes de `g`.
On utilisera `List.sort f l` qui trie `l` utilisant l'ordre
$a \leq b \iff f(a, b) \leq 0$.

Kruskal : Implémentation et complexité

Exercice

Écrire une fonction `tri_aretes g w` renvoyant la liste triée par poids `w` croissant des arêtes de `g`.
On utilisera `List.sort f l` qui trie `l` utilisant l'ordre
$a \leq b \iff f(a,b) \leq 0$.

```
let tri_aretes g w =
    List.sort (fun e1 e2 -> w e1 - w e2) (aretes g)
```

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 savoir si l'ajout de $\{u, v\}$ créé un cycle :

- Parcours de graphe (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 : $O(p \log(p))$
2. Pour chaque arête $\{u, v\}$, déterminer si l'ajout de cette arête créé un cycle.

Pour savoir si l'ajout de $\{u, v\}$ créé un cycle :

- Parcours de graphe (DFS/BFS) en $O(n + p)$
  $\longrightarrow$ Complexité $O(p(n + p))$ pour Kruskal
- Union-Find (comme dans le DS, HP) en $O(1)$
  $\longrightarrow$ Complexité $\boxed{O(p \log(p))}$ pour Kruskal

Kruskal : Détection de cycle avec DFS

Exercice

Écrire une fonction `chemin g u v` qui détermine s'il y a un chemin de `u` à `v` dans `g` (représenté par liste d'adjacence).

Kruskal : Détection de cycle avec DFS

Exercice

En déduire une fonction `kruskal : g w` renvoyant un arbre couvrant de poids minimum d'un graphe connexe `g` pondéré par `w`.

```
let ajout_arete g u v =
    g.(u) <- v::g.(u);
    g.(v) <- u::g.(v)

let kruskal g w =
    let n = Array.length g in
    let t = Array.make n [] in
    let rec aux l = match l with
        | [] -> ()
        | (u, v)::q ->
            if chemin g u v then ajout_arete t u v;
            aux q in
    aux (tri_aretes g w)
```

Kruskal : Détection de cycle avec Union-Find (HP)

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 : Détection de cycle avec Union-Find (HP)

On suppose disposer des fonction d'Union-Find :

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

Kruskal : Détection de cycle avec Union-Find (HP)

On suppose disposer des fonction d'Union-Find :

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

Kruskal : Détection de cycle avec Union-Find (HP)

On suppose disposer des fonction d'Union-Find :

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

Kruskal : Détection de cycle avec Union-Find (HP)

On suppose disposer des fonction d'Union-Find :

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

Kruskal : Détection de cycle avec Union-Find (HP)

On suppose disposer des fonction d'Union-Find :

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

Implémentation efficace (réalisé dans le DS 1) : utiliser une forêt, avec un arbre pour chaque classe, enraciné en le représentant.

Kruskal : Détection de cycle avec Union-Find (HP)

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 : Détection de cycle avec Union-Find (HP)

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 : Détection de cycle avec Union-Find (HP)

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 : Détection de cycle avec Union-Find (HP)

```
let kruskal g w =
    let uf = uf_make () in
    let n = Array.length g in
    let t = Array.make n [] in
    let rec aux = function
        | [] -> ()
        | (u, v)::q -> if uf_find uf u <> uf_find uf v then (
                uf_union uf u v;
                ajout_arete t u v
            );
            aux q
    aux (tri_aretes g w);
    t
```

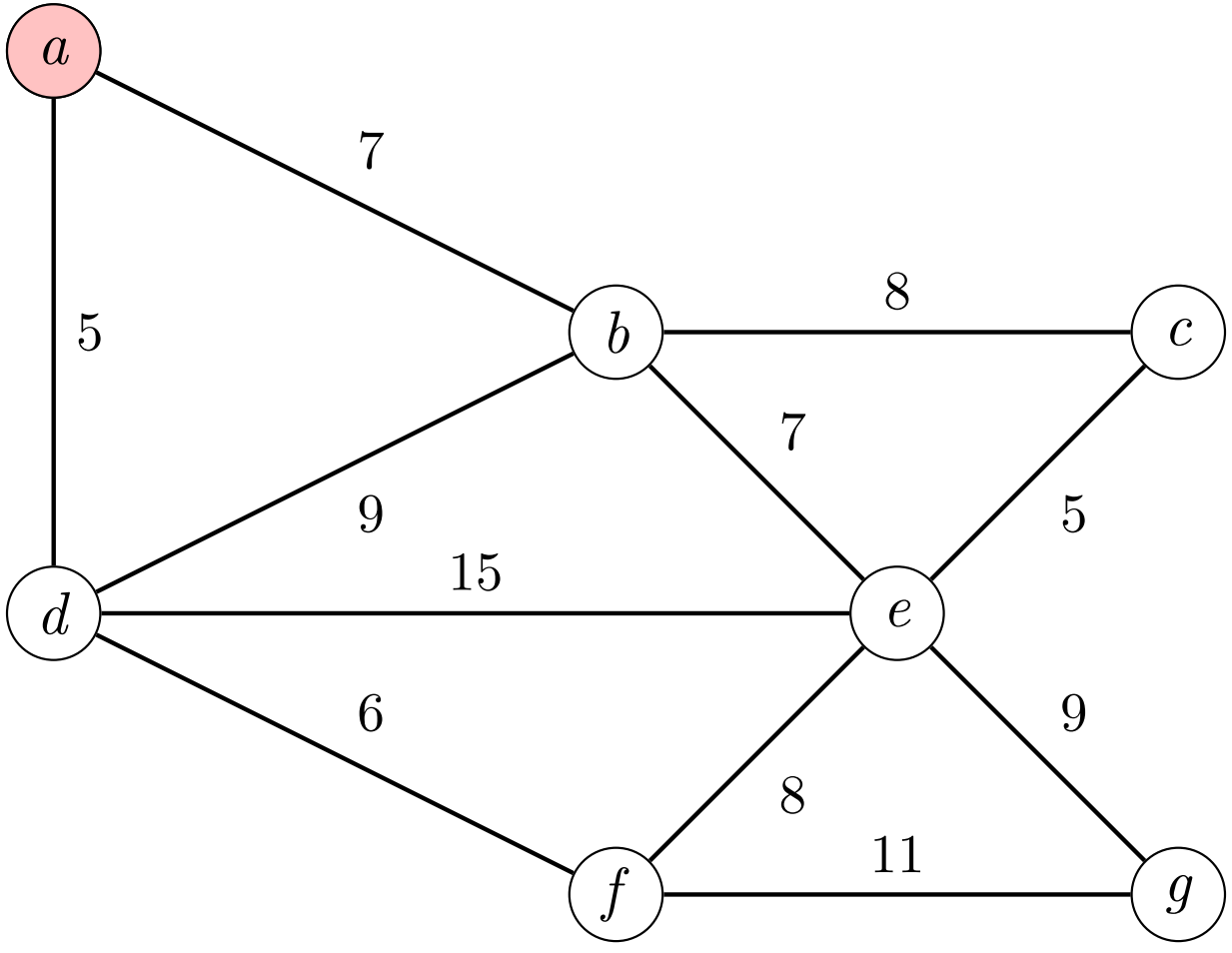
Prim (HP)

Algorithme de Prim :

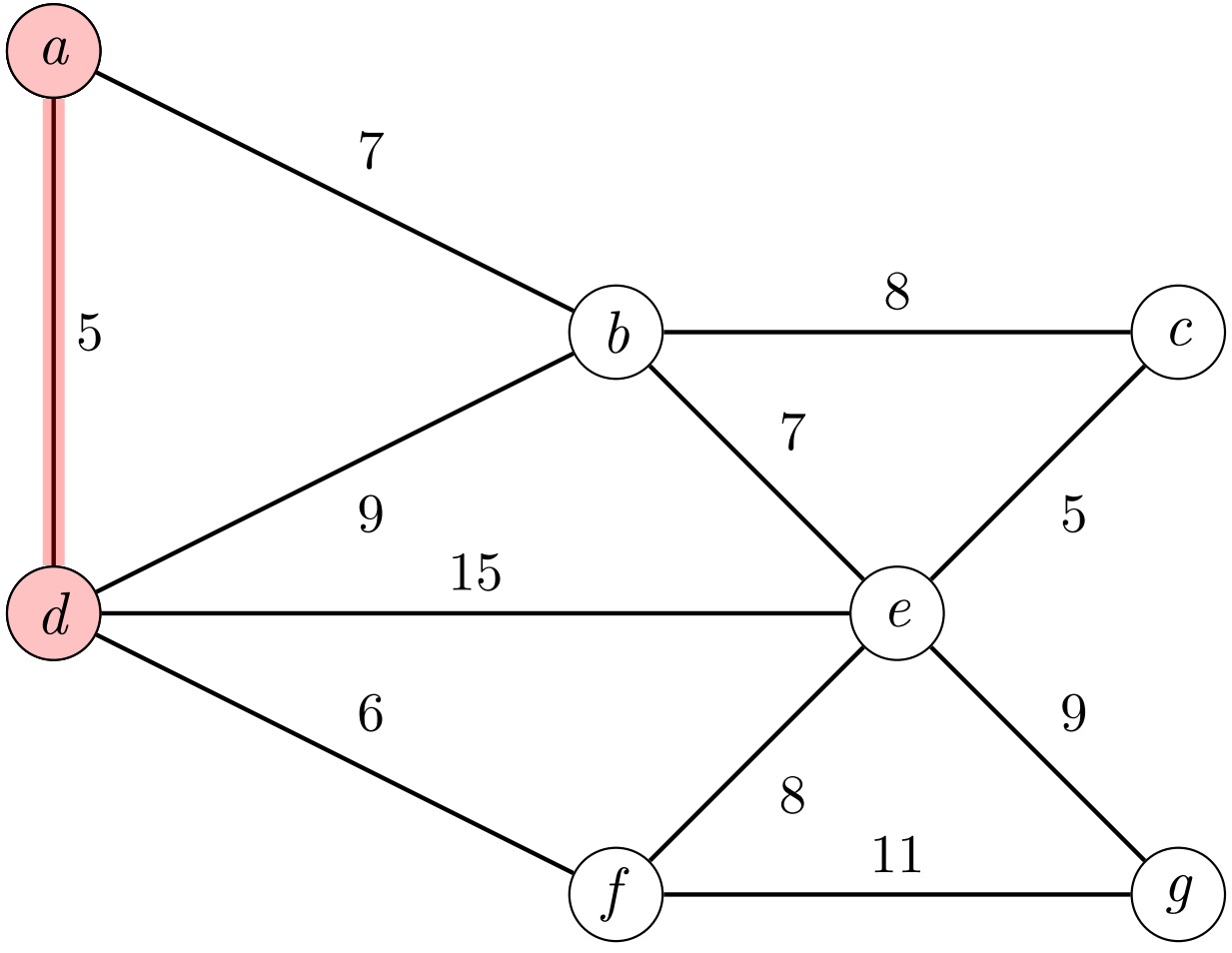
```
Commencer avec un arbre T contenant un seul sommet.

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

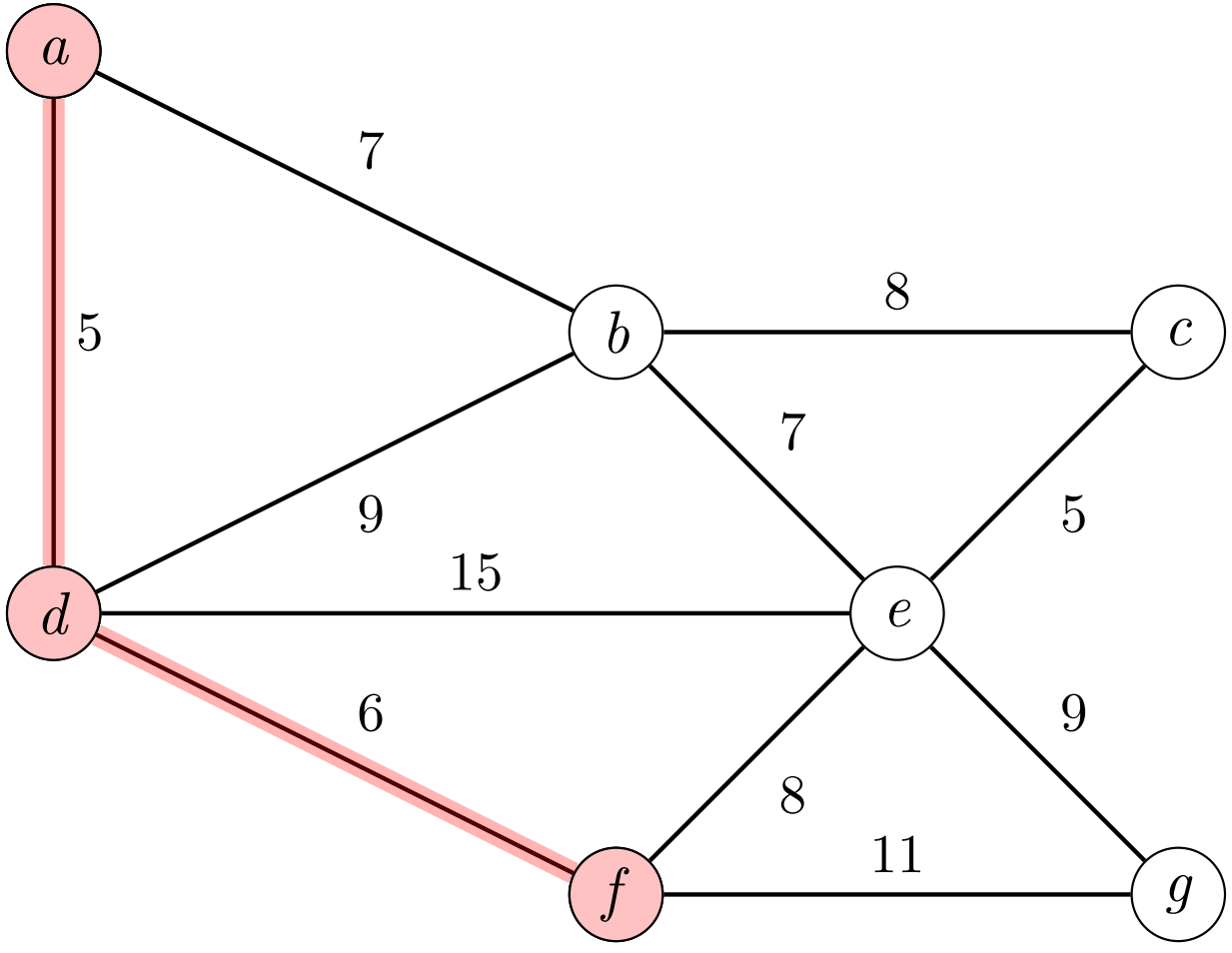
Prim (HP)



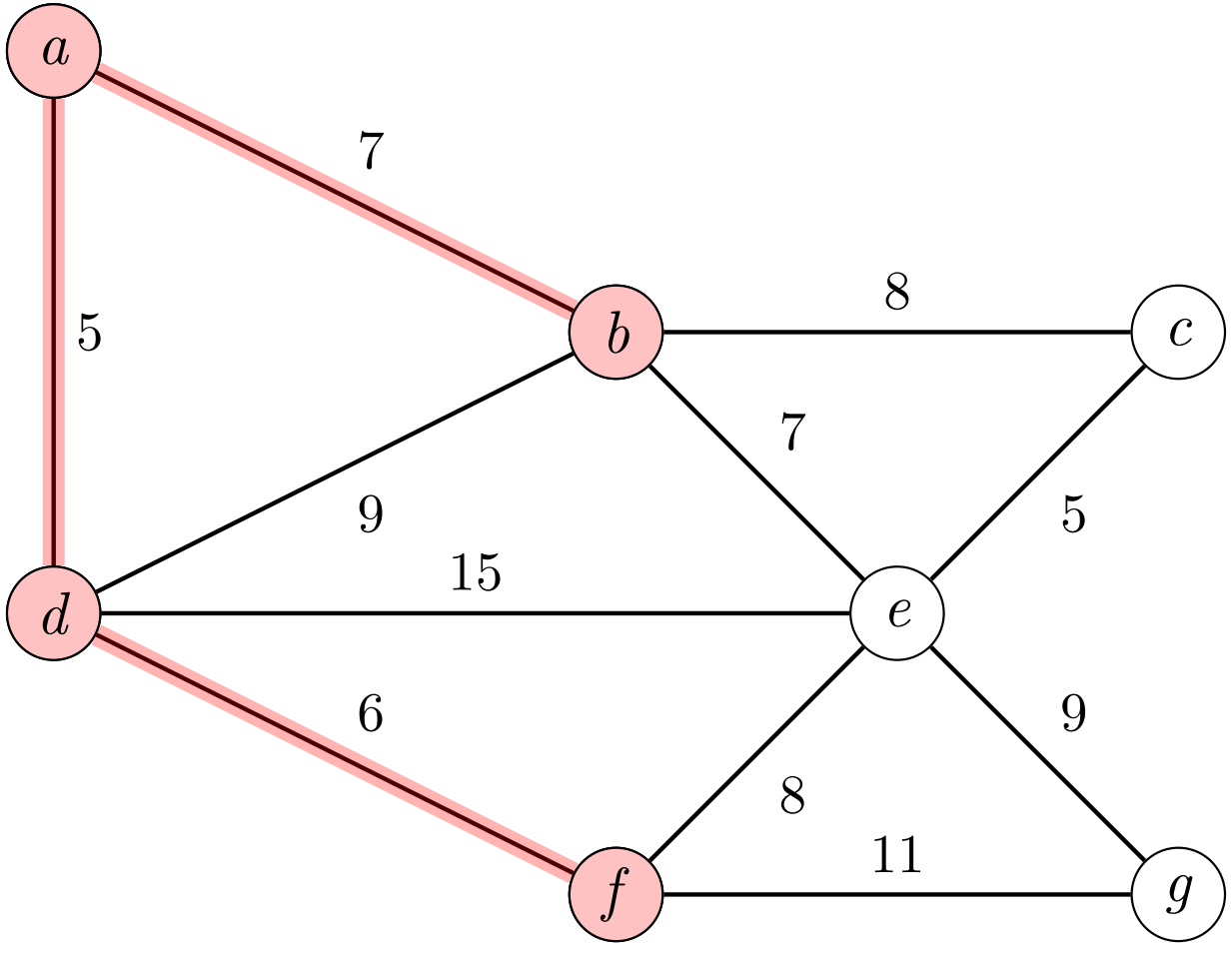
Prim (HP)



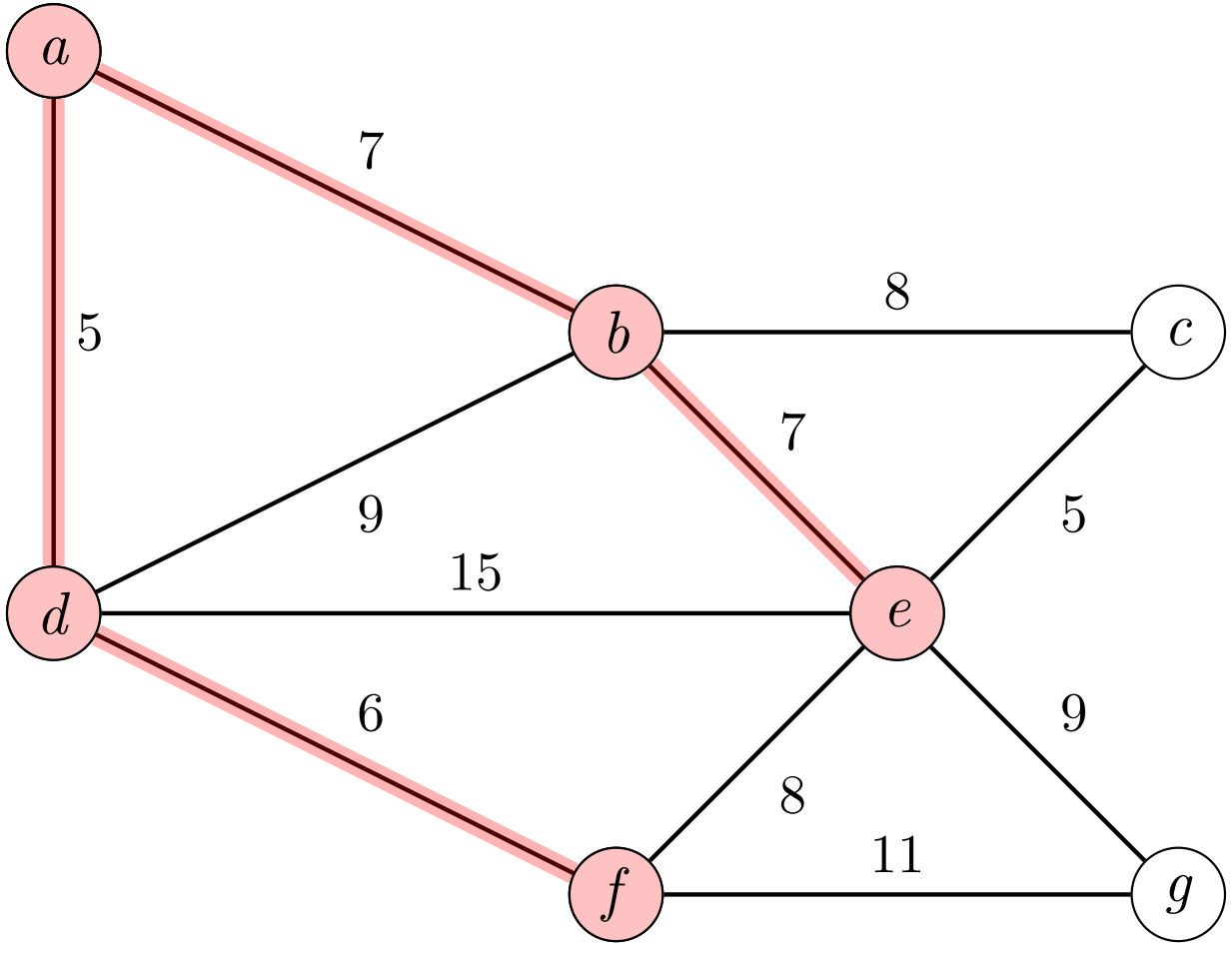
Prim (HP)



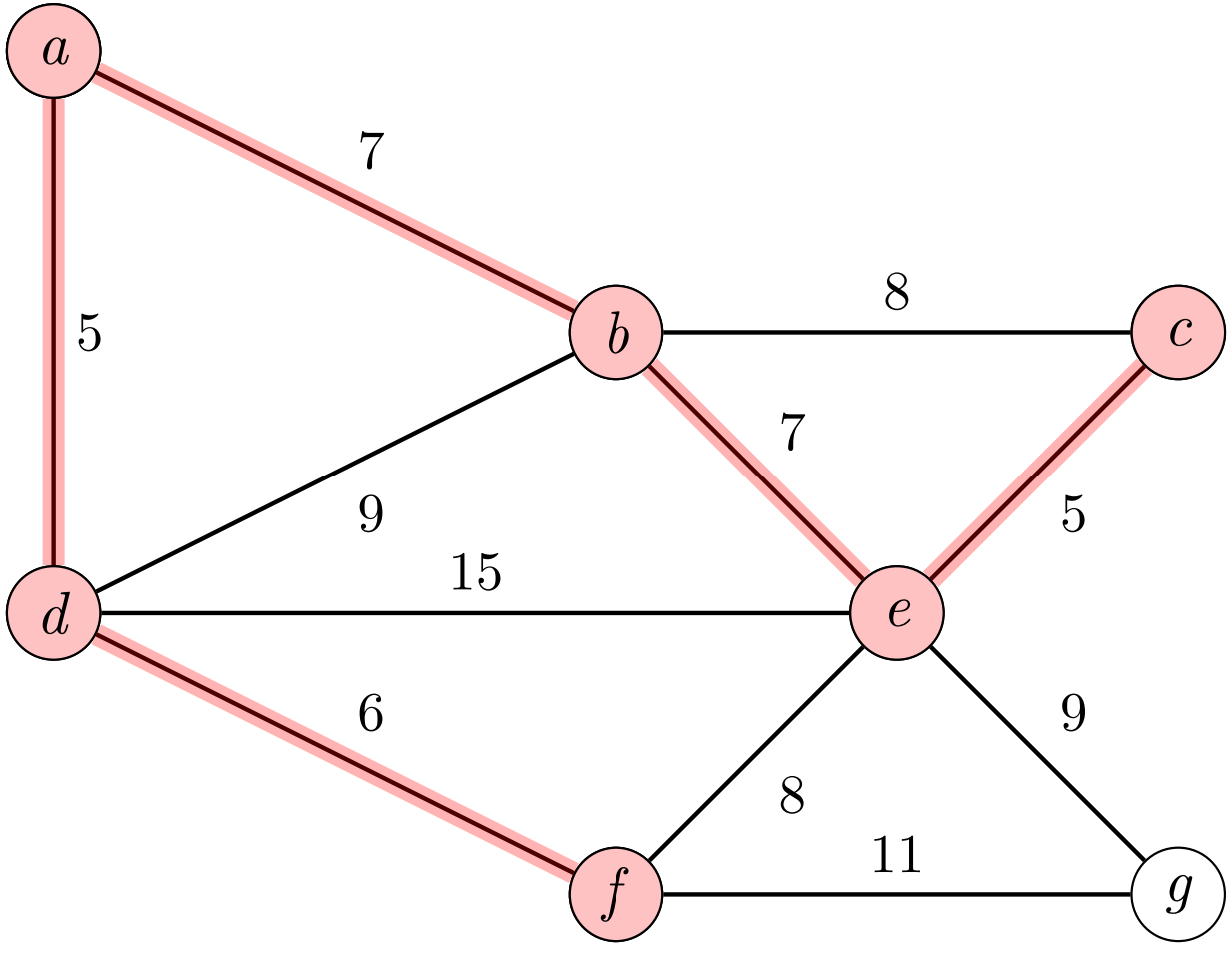
Prim (HP)



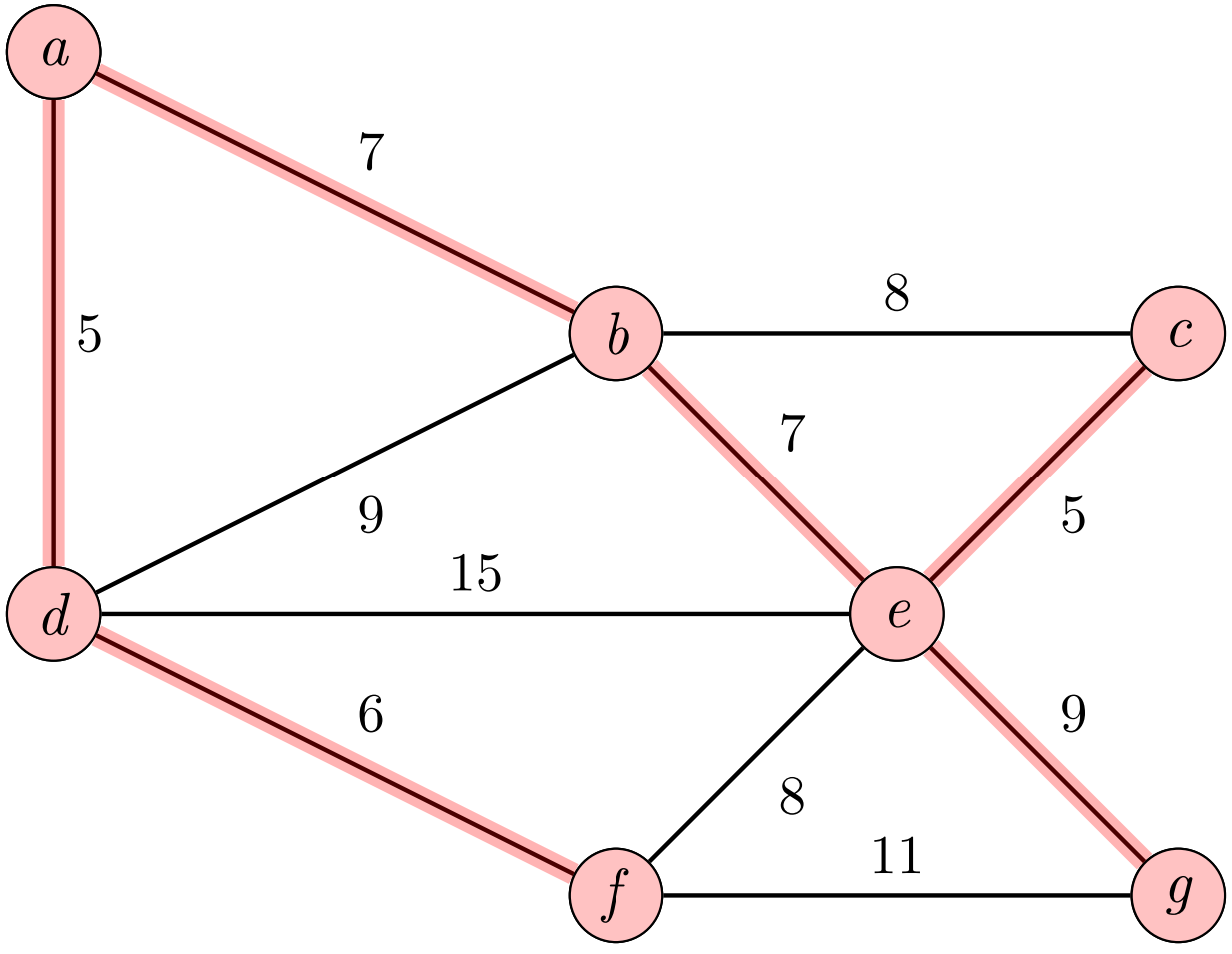
Prim (HP)



Prim (HP)



Prim (HP)



Prim (HP)

Théorème

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

Preuve : Laissée en exercice.

Prim (HP) : Implémentation

Exercice

Implémenter l'algorithme de Prim en OCaml.