Arbre couvrant de poids minimum, algorithme de Kruskal et Union-Find

Quentin Fortier

October 18, 2024

Arbre couvrant de poids minimum

$G = (S, A)$ est un graphe non-orienté pondéré par $p : A \to \mathbb{R}$.

Arbre couvrant

On dit que $T = (S', A')$ est un arbre couvrant de G si :

- T est un sous-graphe de G, c'est-à-dire : $S' \subset S$ et $A' \subset A$.
- T est un arbre.
- T contient tous les sommets de G : $S' = S$.

Le poids $p(T)$ de T comme la somme des poids des arêtes de T.

Arbre couvrant de poids minimum

$G = (S, A)$ est un graphe non-orienté pondéré par $p : A \to \mathbb{R}$.

Arbre couvrant

On dit que $T = (S', A')$ est un arbre couvrant de G si :

- T est un sous-graphe de G, c'est-à-dire : $S' \subset S$ et $A' \subset A$.
- T est un arbre.
- T contient tous les sommets de G : $S' = S$.

Le poids $p(T)$ de T comme la somme des poids des arêtes de T.

Arbre couvrant de poids minimum

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

Exercice

Donner un graphe qui possède plusieurs arbres couvrants de poids minimum.

Arbre couvrant de poids minimum

Lemme

Tout graphe connexe possède un arbre couvrant de poids minimum.

Arbre couvrant de poids minimum

Lemme

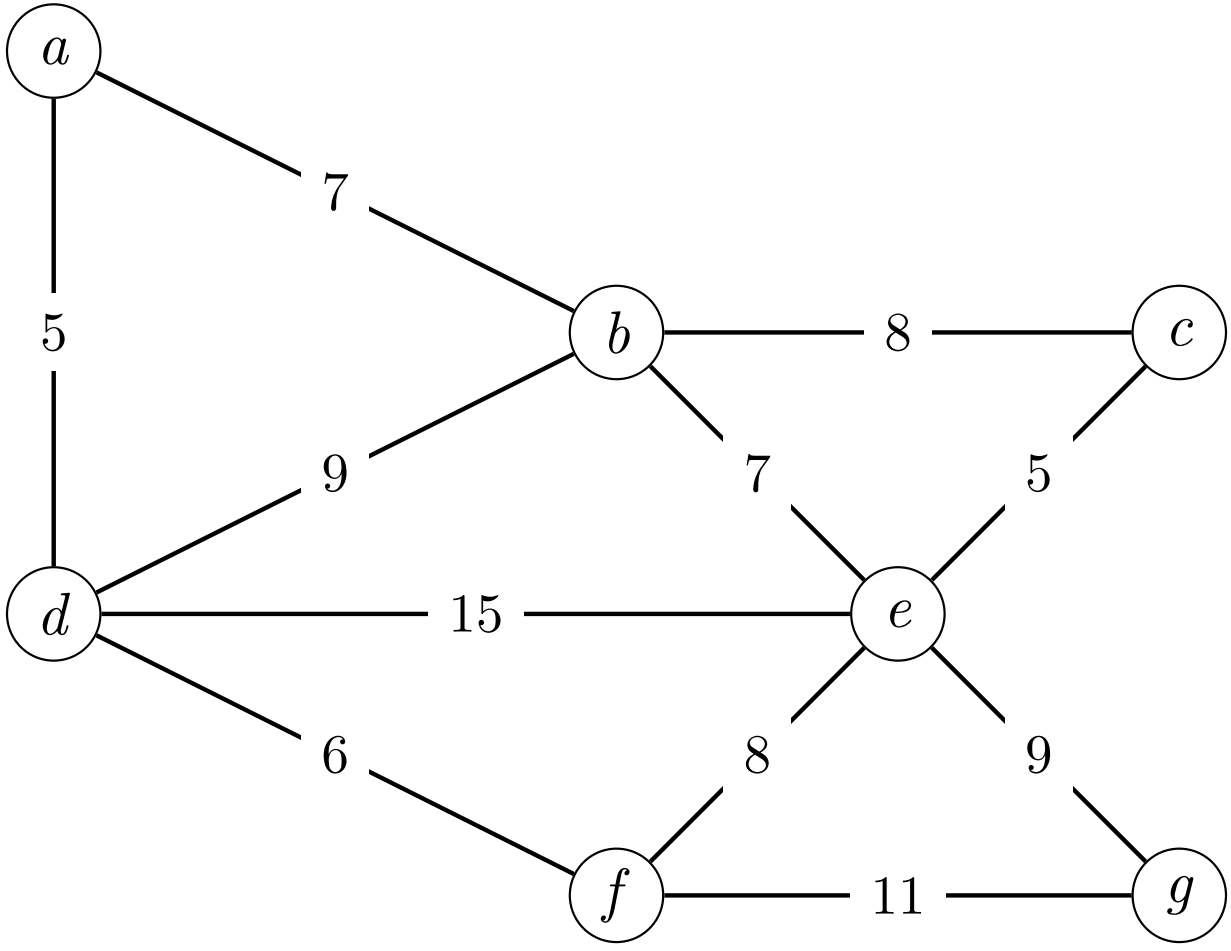
Tout graphe connexe possède un arbre couvrant de poids minimum.

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

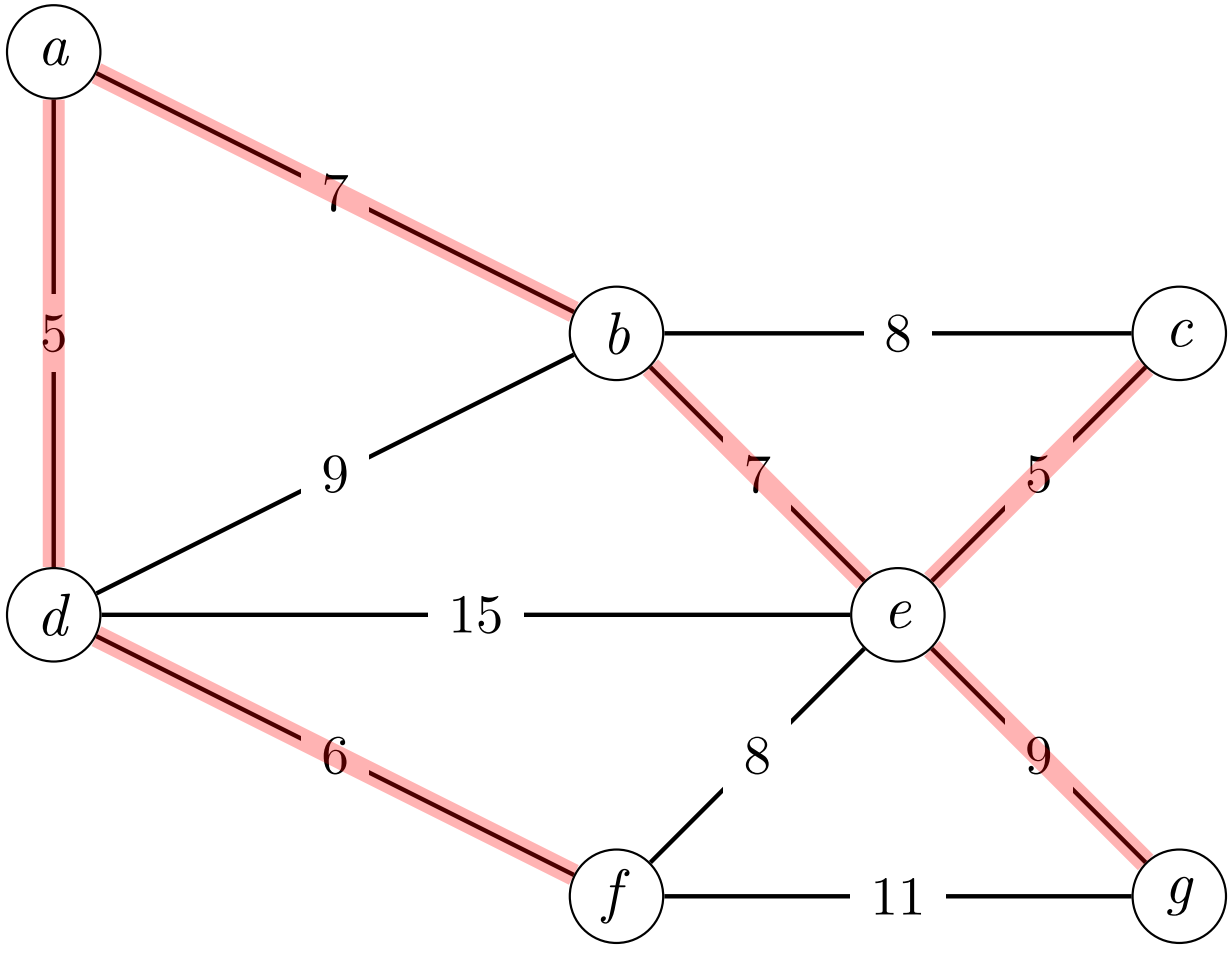
- $E \neq \emptyset$: l'ensemble des arêtes parcourues par un DFS (ou BFS) est un arbre couvrant de G car G est connexe.
- E est fini.

Donc E admet bien un minimum.

Arbre couvrant de poids minimum



Arbre couvrant de poids minimum



Un arbre couvrant de poids minimum

Kruskal

L'algorithme de Kruskal permet de trouver un arbre couvrant de poids minimum sur un graphe connexe $G = (S, A)$:

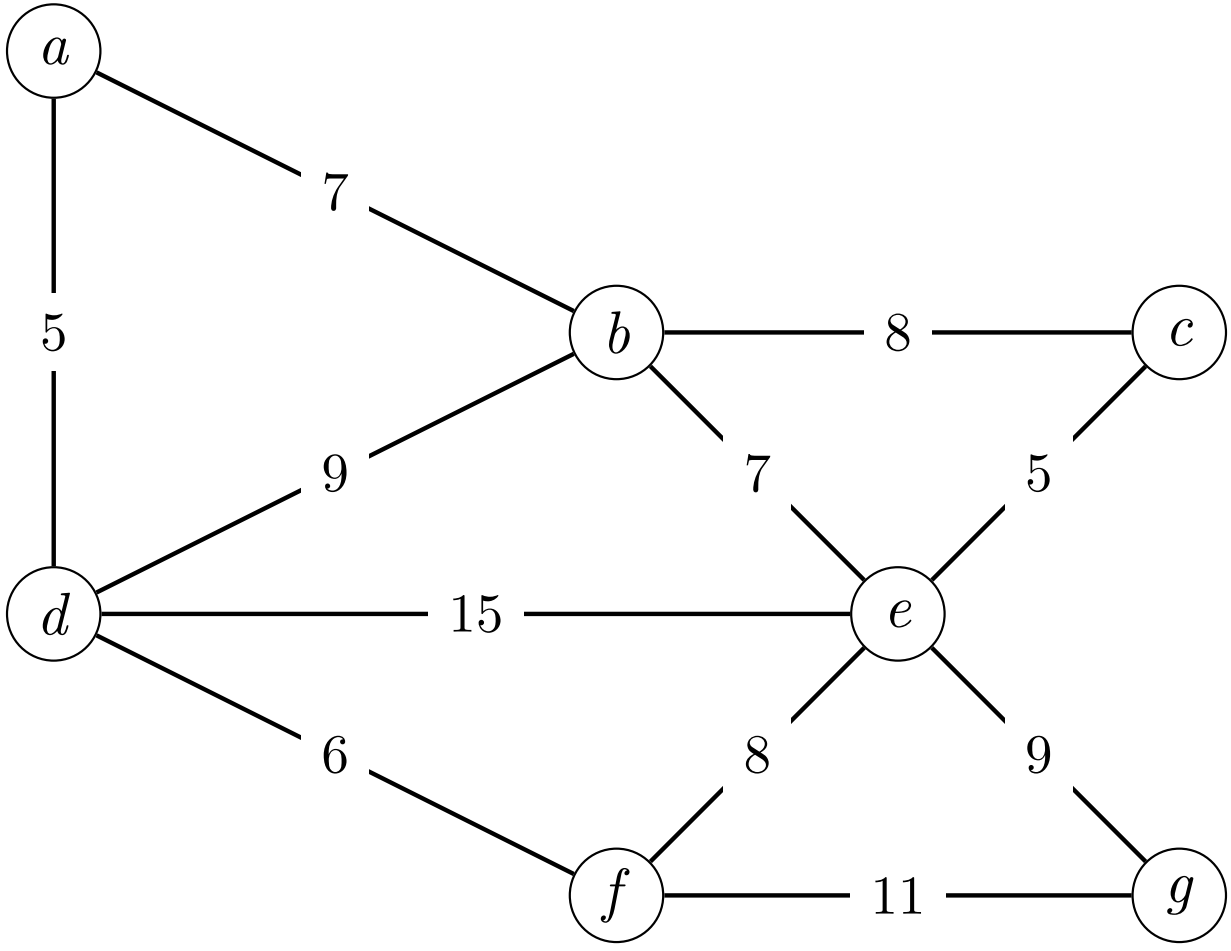
Algorithme de Kruskal

Entrée : Un graphe connexe $G = (S, A)$
Sortie : Un arbre couvrant de poids minimum T

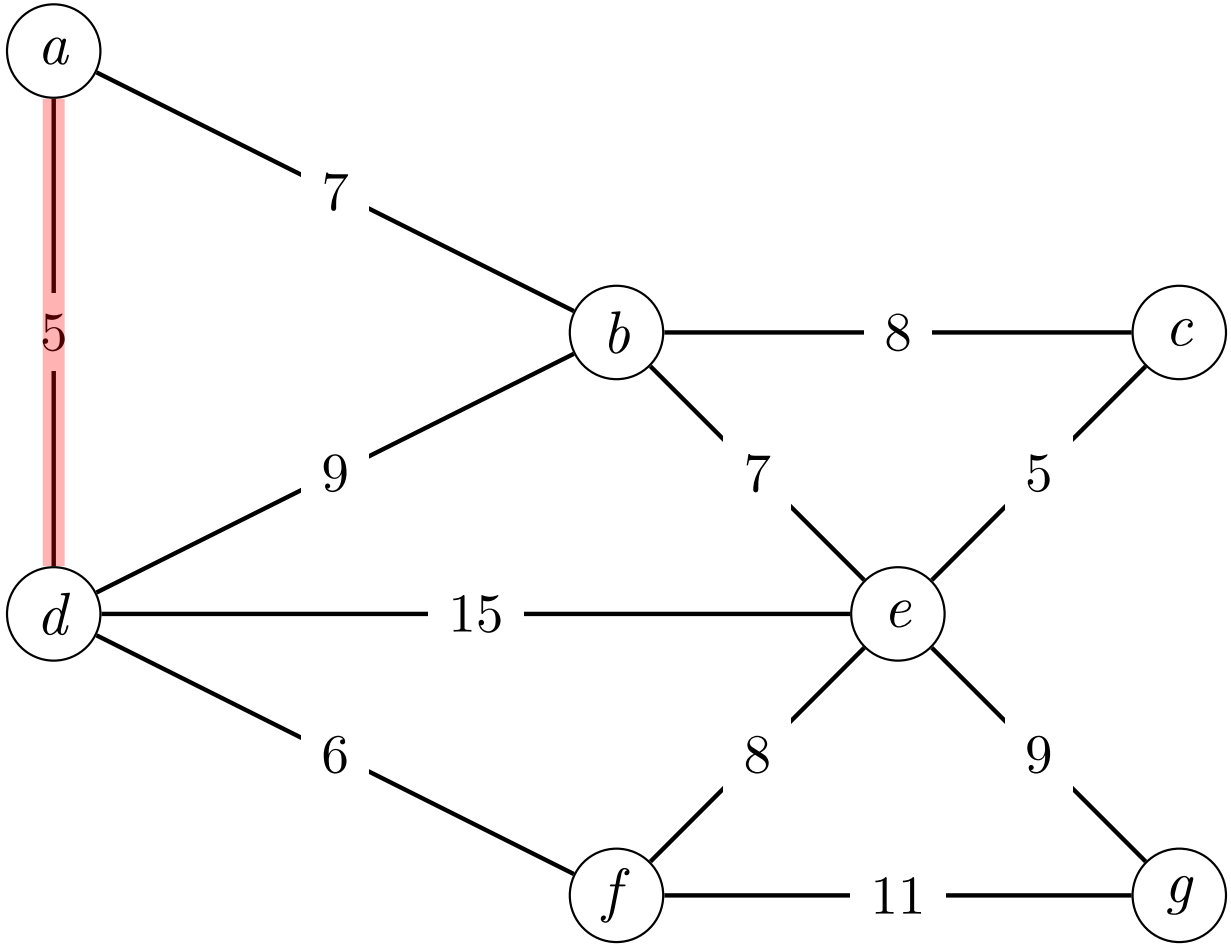
Trier les arêtes de A par poids croissant
$T \leftarrow$ arbre vide (aucune arête)
Pour chaque arête e par poids croissant **:**
    Si $T + e$ est acyclique **:**
        $T \leftarrow T + e$
Renvoyer T

« T est acyclique » est un invariant de boucle.

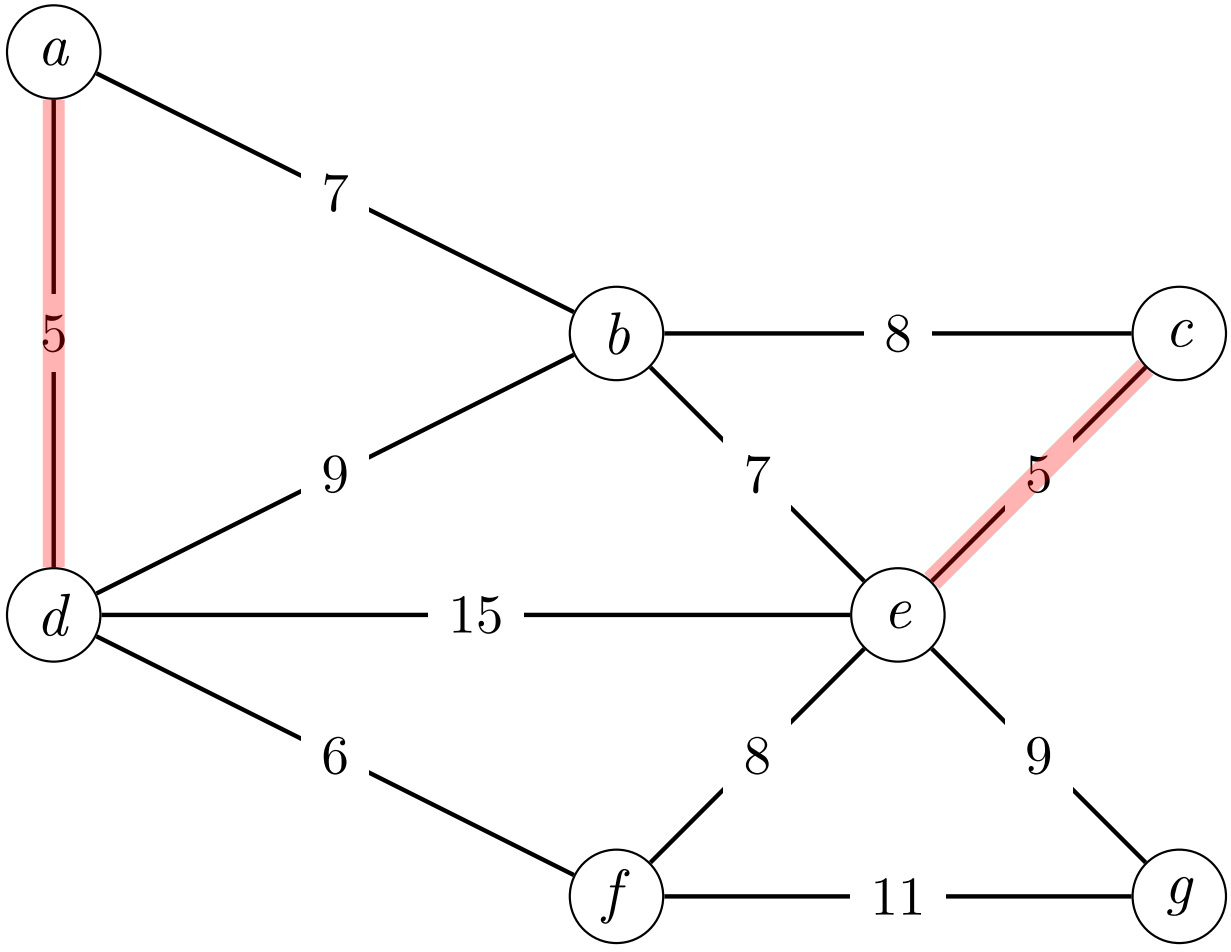
Kruskal



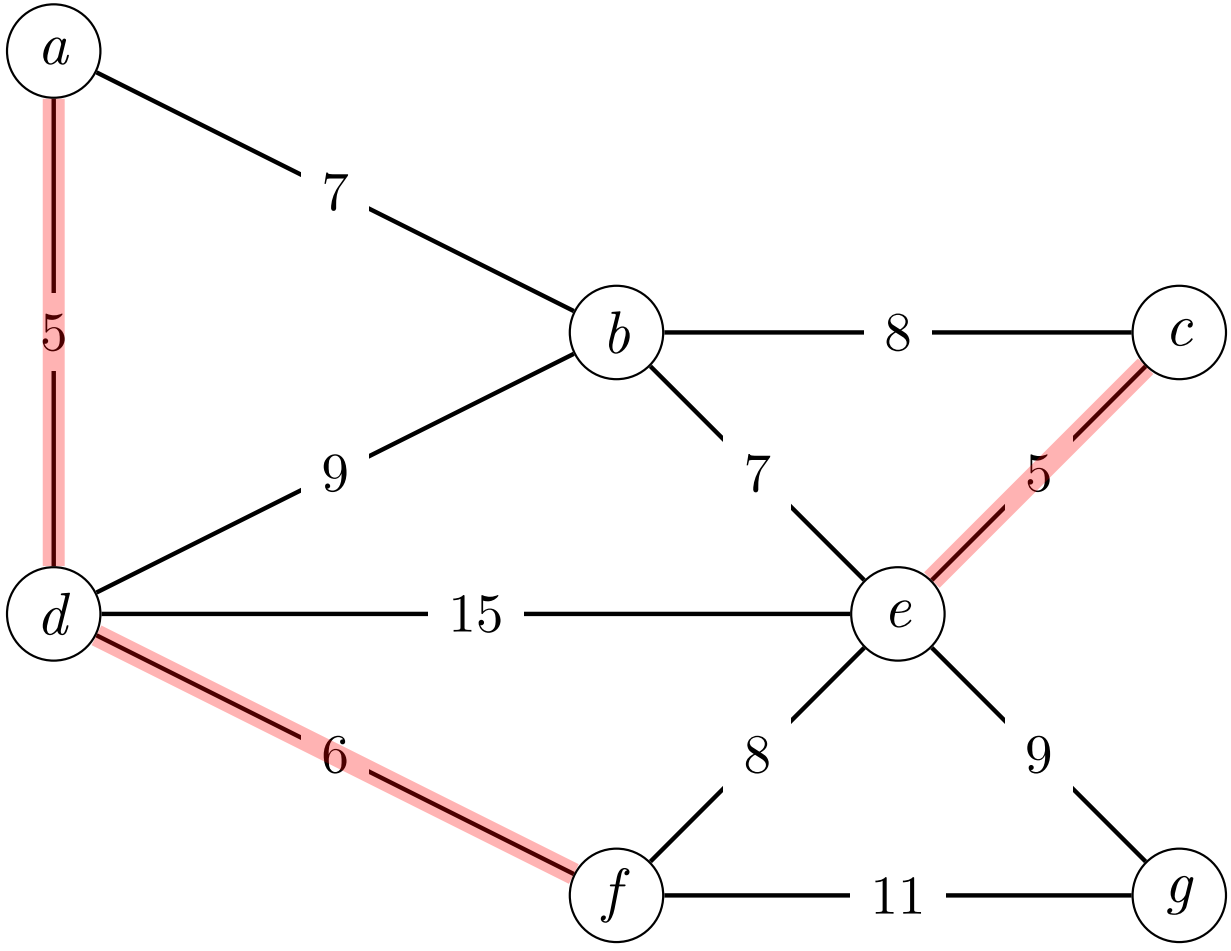
Kruskal



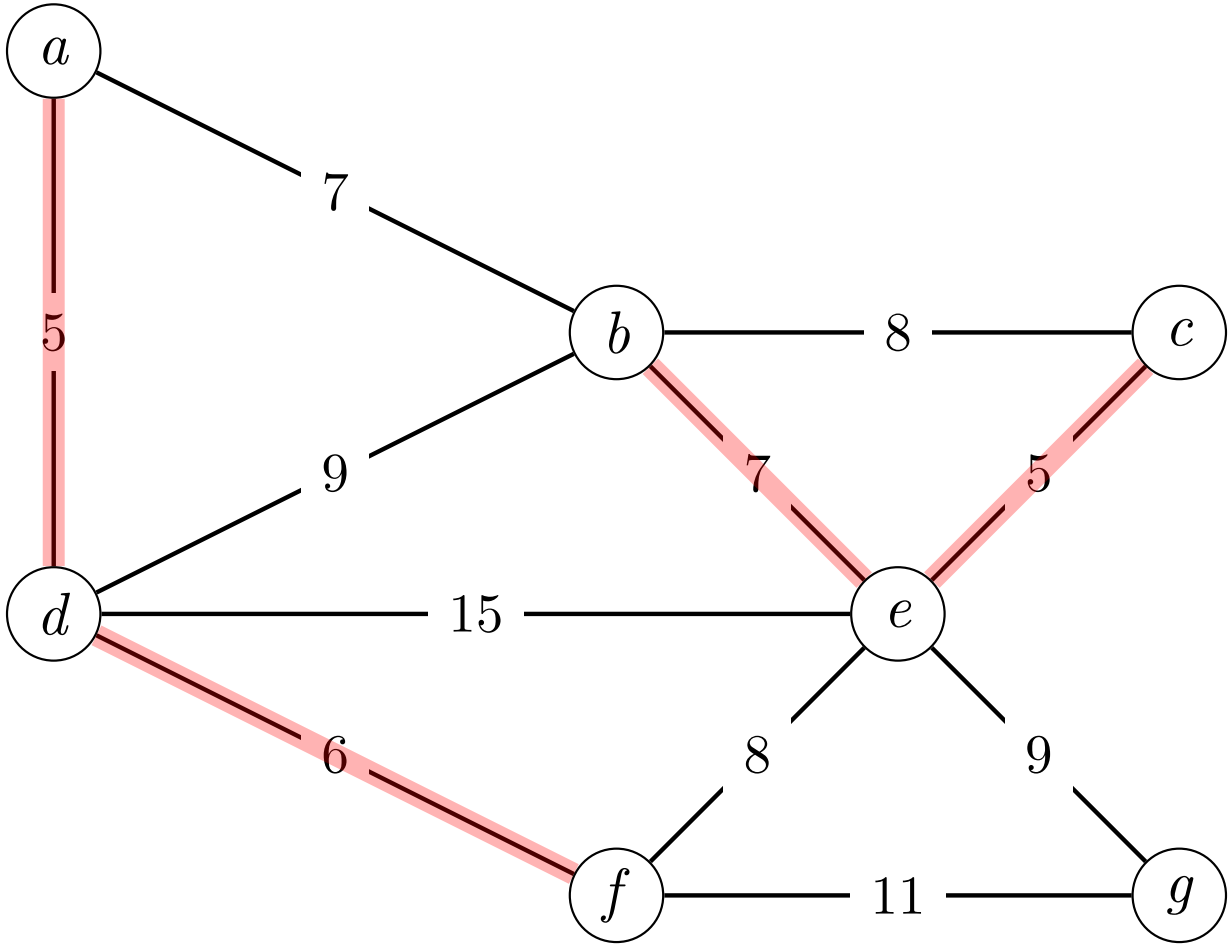
Kruskal



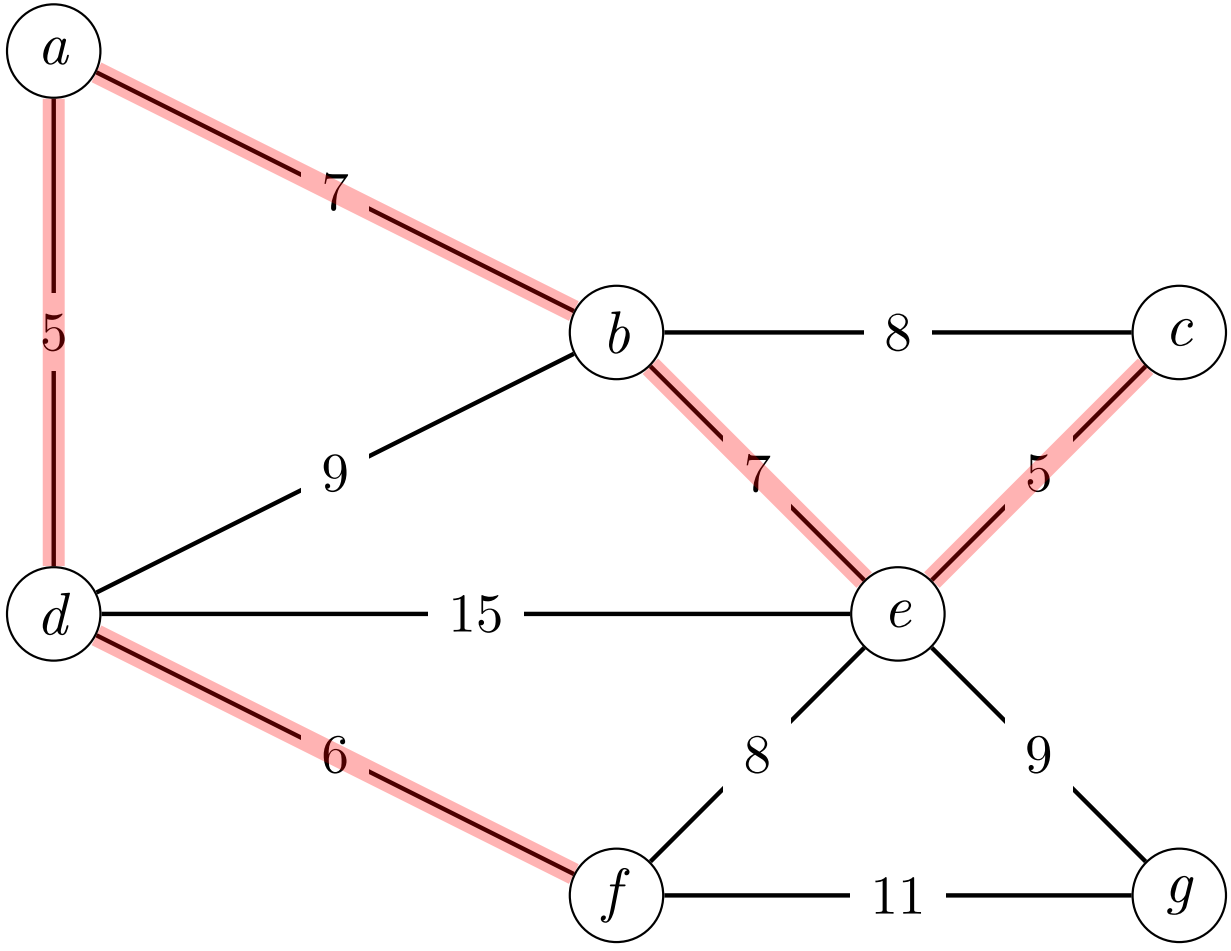
Kruskal



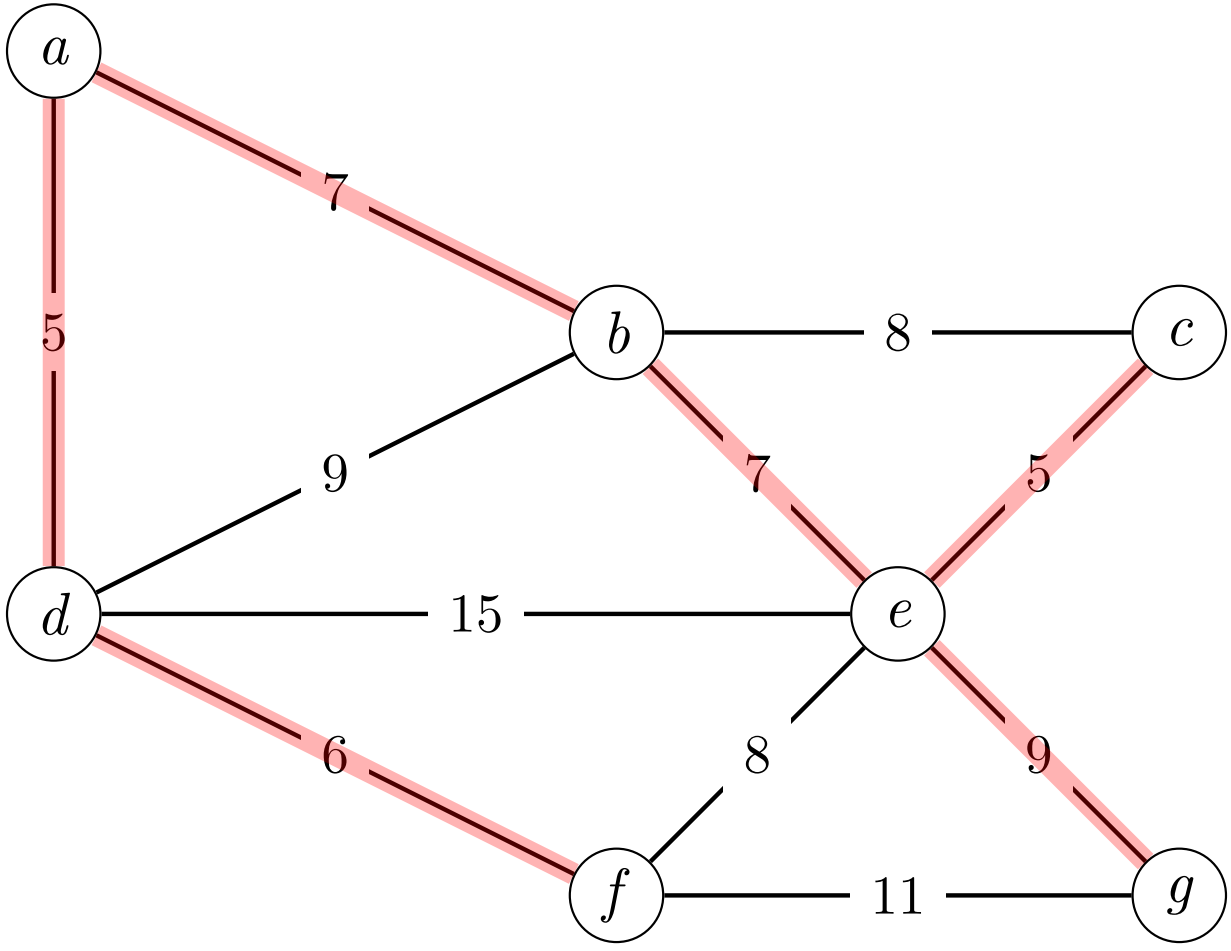
Kruskal



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 d'abord que T est un arbre couvrant :

1. T est sans cycle :

Kruskal

Montrons d'abord que T est un arbre couvrant :

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

Kruskal

Montrons d'abord que T est un arbre couvrant :

1. T est sans cycle : C'est un invariant (l'algorithme ne créé pas de cycle)

2. T est connexe (et couvrant) :
   Soit u et v deux sommets de G.

Kruskal

Montrons d'abord que T est un arbre couvrant :

1. T est sans cycle : C'est un invariant (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

Montrons d'abord que T est un arbre couvrant :

1. T est sans cycle : C'est un invariant (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 $S \setminus U$.

Kruskal

Montrons d'abord que T est un arbre couvrant :

1. T est sans cycle : C'est un invariant (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 $S \setminus U$.
   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 $e^* \in T^* \setminus T$.

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 $e^* \in T^* \setminus T$.
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 $e^* \in T^* \setminus T$.
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 $e^* \in T^* \setminus T$.
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.
- $p(e) \leq p(e^*)$ (sinon Kruskal aurait ajouté e^* à T).

Kruskal

Considérons $T_2 = T + e^* - e$.

Kruskal

Considérons $T_2 = T + e^* - e$.

- T_2 est un arbre couvrant

Kruskal

Considérons $T_2 = T + e^* - e$.

- T_2 est un arbre couvrant car connexe avec $n - 1$ arêtes.

Kruskal

Considérons $T_2 = T + e^* - e$.

- T_2 est un arbre couvrant car connexe avec $n - 1$ arêtes.
- $p(T) \leq p(T_2)$.

Kruskal

Considérons $T_2 = T + e^* - e$.

- T_2 est un arbre couvrant car connexe avec $n-1$ arêtes.
- $p(T) \leq p(T_2)$.

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

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

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

Kruskal

Exercice

1. Peut-on adapter l'algorithme de Kruskal pour trouver un arbre couvrant de poids maximum ?
2. De façon similaire, peut-on adapter un algorithme de plus courts chemins (par exemple Dijkstra) pour trouver des chemins de poids maximum ?
3. Soit $e \in A$. Peut-on adapter l'algorithme de Kruskal pour trouver un arbre couvrant de poids minimum contenant e ?

Implémentation naïve

On suppose G implémenté par une liste d'adjacence
`g : (int*float) list array` telle que `g.(i)` est la liste des couples (j, p) tels que $\{i, j\}$ est une arête de poids p.

Fonction qui renvoie la liste des arêtes du graphe g, où une arête $\{u, v\}$ de poids p est représentée par le couple (u, v, p) :

```
let aretes g : list =
  let rec aux i =
      if i = Array.length g then []
      else
          let rec aux_i = function
              | [] -> []
              | (j, p)::q -> (i, j, p)::aux_i q in
          aux_i g.(i) @ aux (i + 1) in
  aux 0
```

Implémentation naïve

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

1. Trier les arêtes par poids croissant :

Implémentation naïve

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

1. Trier les arêtes par poids croissant : $\mathrm{O}(p \log(p)) = \mathrm{O}(p \log(n))$ (avec tri fusion)
2. Pour chaque arête $\{u, v\}$, déterminer si l'ajout de cette arête créé un cycle revient à savoir si u et v sont dans la même composante connexe de l'arbre T en construction. Deux possibilités :
   - Parcours de graphe (DFS/BFS)

Implémentation naïve

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

1. Trier les arêtes par poids croissant : $\mathrm{O}(p\log(p)) = \mathrm{O}(p\log(n))$ (avec tri fusion)
2. Pour chaque arête $\{u, v\}$, déterminer si l'ajout de cette arête créé un cycle revient à savoir si u et v sont dans la même composante connexe de l'arbre T en construction. Deux possibilités :
   - Parcours de graphe (DFS/BFS) en $\mathrm{O}(n+p)$
     $\longrightarrow$ Complexité $\mathrm{O}(p^2)$ pour Kruskal
   - Union-Find en $\approx \mathrm{O}(1)$
     $\longrightarrow$ Complexité $\boxed{\mathrm{O}(p\log(n))}$ pour Kruskal

Implémentation naïve : 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`.

Implémentation naïve : 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`.

```
let chemin g u v =
    let n = Array.length g in
    let vus = Array.make n false in
    let rec aux w =
        if not vus.(w) then (
            vus.(w) <- true;
            List.iter (fun (x, p) -> aux x) g.(w)
        ) in
    aux u;
    vus.(v)
```

Complexité : $O(n + p)$ (DFS avec liste d'adjacence).

Implémentation naïve : Détection de cycle avec DFS

On suppose l'existence d'une fonction `tri : ('a*'a*float) list -> ('a*'a*float) list` qui trie une liste d'arêtes par ordre croissant de poids.

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

let kruskal g =
    let n = Array.length g in
    let t = Array.make n [] in
    let rec aux l = match l with
        | [] -> t
        | (u, v, p)::q ->
            if not (chemin t u v) then ajout_arete t u v p;
            aux q in
    g |> aretes |> tri |> aux
```

Union-Find

La structure Union-Find (unir et trouver) permet de représenter une partition d'un ensemble $E = [\![0, n-1]\!]$ comme réunion disjointe de sous-ensembles (classes).
À chaque élément de E est associé un représentant, qui est l'élément de sa classe.

Union-Find

La structure Union-Find (unir et trouver) permet de représenter une partition d'un ensemble $E = [\![0, n-1]\!]$ comme réunion disjointe de sous-ensembles (classes).
À chaque élément de E est associé un représentant, qui est l'élément de sa classe.

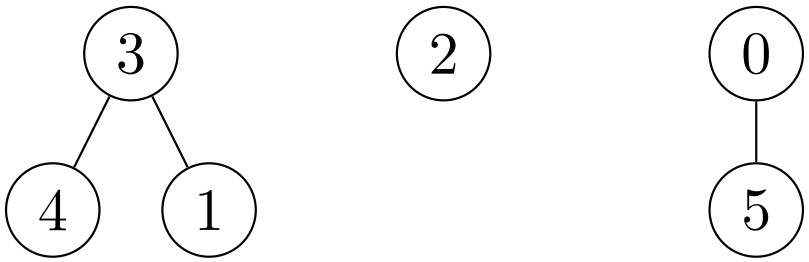
Opérations sur une structure d'Union-Find :

- Création : créer une structure Union-Find avec n éléments, chaque élément étant seul dans sa classe.
- Find : trouver le représentant de la classe d'un élément.
- Union : fusionner les classes de deux éléments.

Chaque classe est représentée par un arbre, enraciné en le représentant.

Union-Find

Par exemple, la forêt suivante est une représentation possible de la partition $\{\{1,3,4\},\{2\},\{0,5\}\}$:



On la représente par un tableau `uf` tel que `uf.(i)` est le père de i dans l'arbre (`uf.(i) = i` si `i` est le représentant de sa classe).
Sur l'exemple ci-dessus, `uf = [|0; 3; 2; 3; 3; 0|]`.

Union-Find

```
let create n = (* O(n) *)
    Array.init n (fun i -> i)
(* Array.init n f renvoie [|f 0; f 1; ...|] *)

let rec find uf i = (* O(h) *)
    if uf.(i) = i then i
    else find uf uf.(i)
```

Avec `h` la hauteur de l'arbre contenant i, qu'on peut majorer par n.

Union-Find

On peut fusionner les classes de x et y en cherchant leurs représentants r_x et r_y et en mettant r_x comme père de r_y :

```
let union uf x y = (* O(h) *)
    let rx = find uf x in
    let ry = find uf y in
    uf.(ry) <- rx
```

Union-Find

Application d'Union-Find à Kruskal :

- Chaque classe correspond à une composante connexe dans T.
- Si u et v sont dans la même classe (`find t u = find t 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 t u v`.

Union-Find

```
let kruskal g =
    let n = Array.length g in
    let t = Array.make n [] in
    let uf = create n in
    let rec aux = function
        | [] -> t
        | (u, v, p)::q ->
            if find uf u <> find uf v then (
                union uf u v;
                ajout_arete t u v p
            );
            aux q in
    g |> aretes |> tri |> aux
```

Complexité : $O(p \log(n) + np) = O(np)$ (comme avec DFS...).

Union-Find

On peut améliorer la structure d'Union-Find en utilisant l'heuristique de l'union par rang et la compression de chemin :

- Union par rang : dans `union`, on met l'arbre de hauteur la plus petite comme fils de celui de hauteur plus grande.
- Compression de chemin : dans `find`, on attache directement chaque nœud rencontré à la racine.

Union-Find : Union par rang

On ajoute un tableau `h` tel que `h.(i)` est la hauteur de l'arbre enraciné en `i`.

```
type uf = {t : int array; h : int array}

let create n =
    {t = Array.init n (fun i -> i); h = Array.make n 0}

let union uf x y =
    let rx = find uf x in
    let ry = find uf y in
    if rx <> ry then (
        if uf.h.(rx) < uf.h.(ry) then uf.t.(rx) <- ry
        else if uf.h.(rx) > uf.h.(ry) then uf.t.(ry) <- rx
        else (
            uf.t.(ry) <- rx;
            uf.h.(rx) <- uf.h.(rx) + 1
        )
    )
```

Union-Find : Union par rang

Théorème

Dans une structure d'Union-Find avec union par rang, la hauteur h d'un arbre à k nœuds vérifie $h \leq \log_2(k)$ (dit autrement : $k \geq 2^h$).

Preuve de l'invariant :

Union-Find : Union par rang

Théorème

Dans une structure d'Union-Find avec union par rang, la hauteur h d'un arbre à k nœuds vérifie $h \leq \log_2(k)$ (dit autrement : $k \geq 2^h$).

Preuve de l'invariant :

C'est vrai initialement car chaque arbre n'a qu'un nœud et sa hauteur est 0.

Union-Find : Union par rang

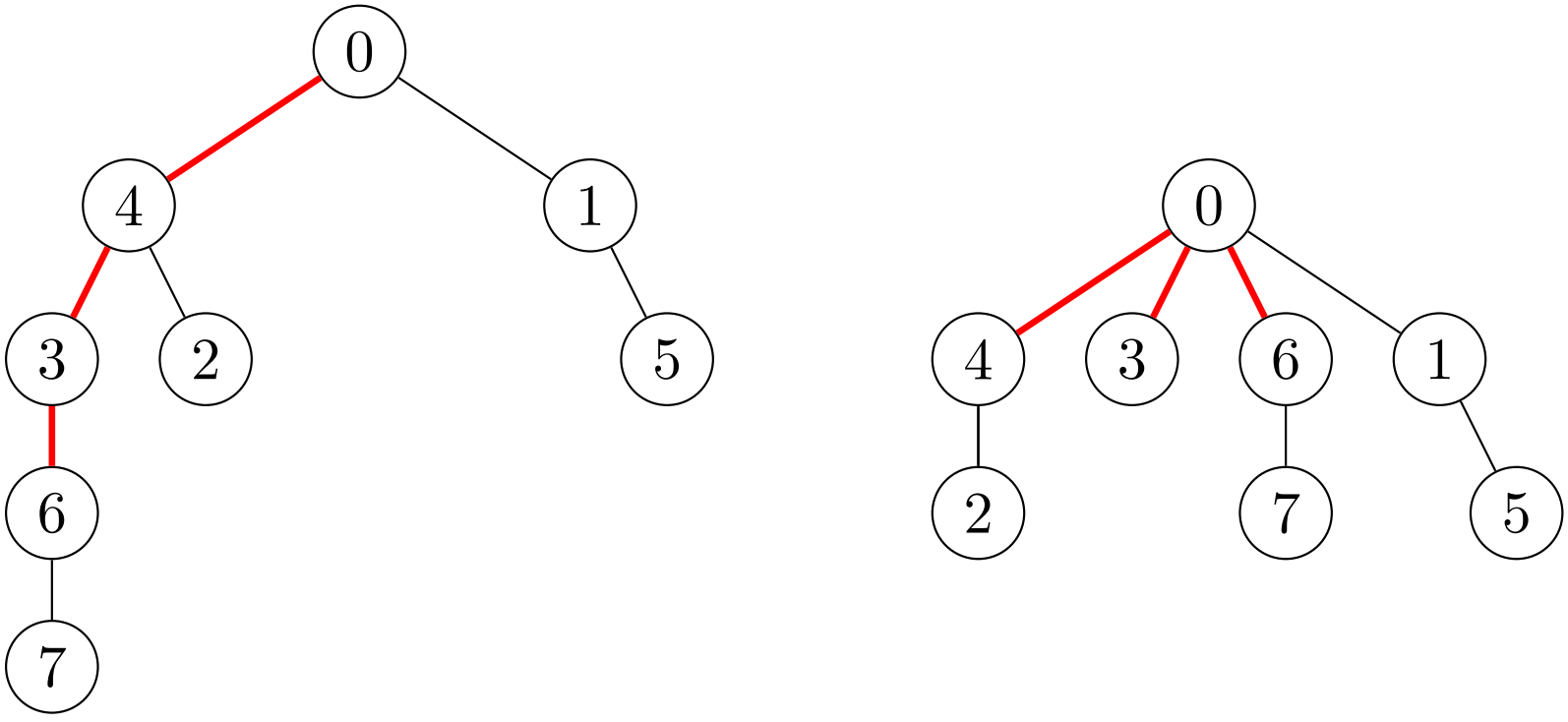
Théorème

Dans une structure d'Union-Find avec union par rang, la hauteur h d'un arbre à k nœuds vérifie $h \leq \log_2(k)$ (dit autrement : $k \geq 2^h$).

Preuve de l'invariant :
C'est vrai initialement car chaque arbre n'a qu'un nœud et sa hauteur est 0.
Supposons qu'on appelle `union` sur deux arbres de hauteur h_1 et h_2.
D'après l'invariant, ils contiennent 2^{h_1} et 2^{h_2} nœuds respectivement.

- Si $h_1 < h_2$, on attache l'arbre de hauteur h_1 à celui de hauteur h_2. Le nouvel arbre a une hauteur de h_2 et est de taille $2^{h_1} + 2^{h_2} \geq 2^{h_2}$.
- Si $h_1 = h_2$, le nouvel arbre a une hauteur de $h_1 + 1$ et est de taille $2^{h_1} + 2^{h_2} = 2^{h_1+1}$.

Union-Find : Compression de chemin



Appel de `find uf` 6 avec compression de chemin

Union-Find : Compression de chemin

```
let rec find uf i =
    if uf.t.(i) = i then i
    else (
        let r = find uf uf.t.(i) in
        uf.t.(i) <- r;
        r
    )
```

Union-Find : Compression de chemin

Théorème (admis)

Avec union par rang et compression de chemin, la complexité amortie de `union` et `find` est en $O(\alpha(n))$, où α est une fonction à croissance tellement lente qu'on peut la considérer comme constante.

Ne pas confondre :

- Complexité en moyenne : on moyenne la complexité sur toutes les entrées possibles.
- Complexité amortie : complexité dans le pire cas d'une suite de n opérations, divisé par n.

Union-Find : Compression de chemin

```
let kruskal g =
    let n = Array.length g in
    let t = Array.make n [] in
    let uf = create n in
    let rec aux = function
        | [] -> t
        | (u, v, p)::q ->
            if find uf u <> find uf v then (
                union uf u v;
                ajout_arete t u v p
            );
            aux q in
    g |> aretes |> tri |> aux
```

Complexité : $O(p\log(n) + p\alpha(n)) = \boxed{O(p\log(n))}$.

Union-Find : Compression de chemin

Exercice

Dessiner une exécution possible des opérations suivantes :

```
let uf = create 6 in
union uf 0 2;
union uf 3 4;
union uf 1 4;
union uf 0 1;
find uf 1;
union uf 2 5
```