INF3105 – Arbres de recouvrement minimal (ARM)

Jaël Champagne Gareau

Université du Québec à Montréal (UQAM)

Été 2024

http://cria2.uqam.ca/INF3105/

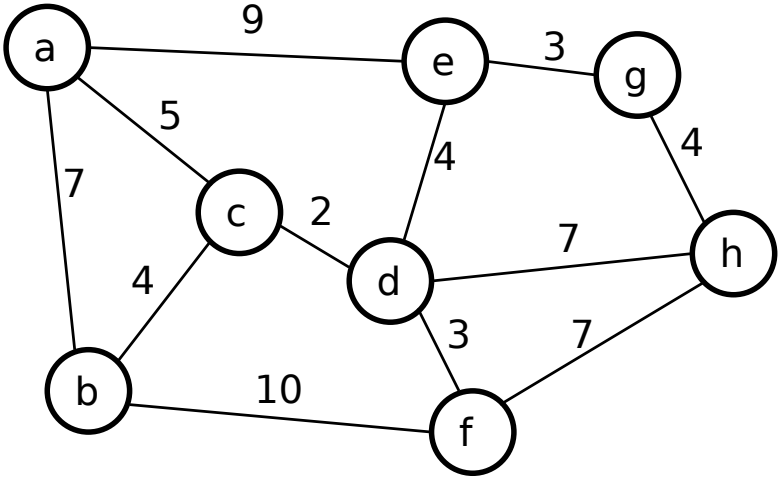
UQÀM

Définitions

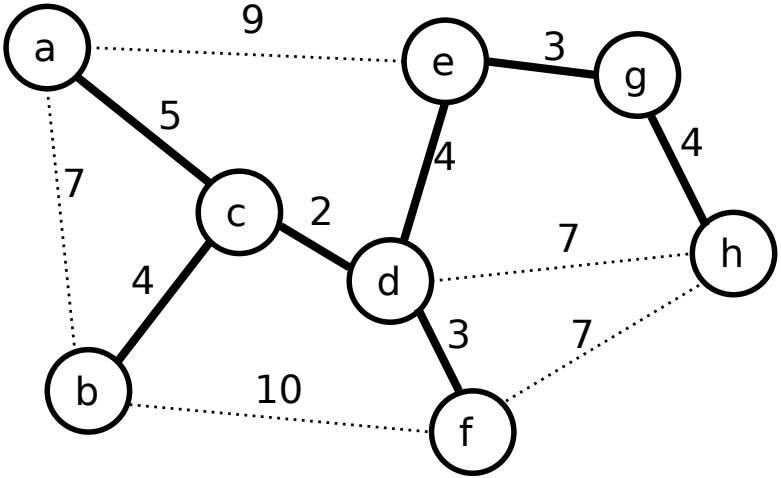
Arbre de recouvrement

- Entrée : graphe non orienté pondéré et connexe.
- Sortie : Sous-graphe connexe acyclique (arbre).
  - Nombre d'arêtes = Nombre de sommets - 1.
  - Existence de chemin reliant toutes les paires de sommets.
  - Contient tous les sommets du graphe initial.

Exemple de graphe



Exemple d'arbre de recouvrement



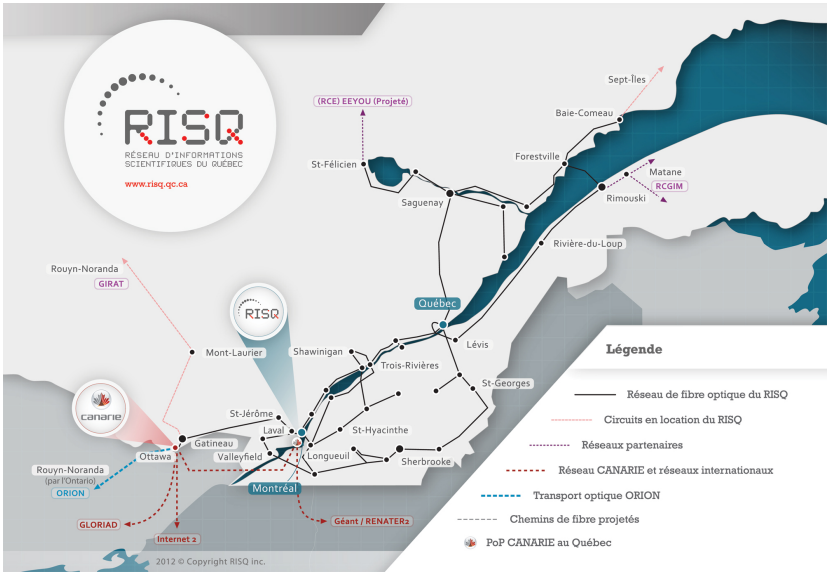
Définitions

Arbre de recouvrement à coût minimal (ARM)

- Arbre de recouvrement.
- La somme des coûts des arêtes sélectionnées est minimale.
- La solution n'est pas nécessairement unique.

Télécommunications

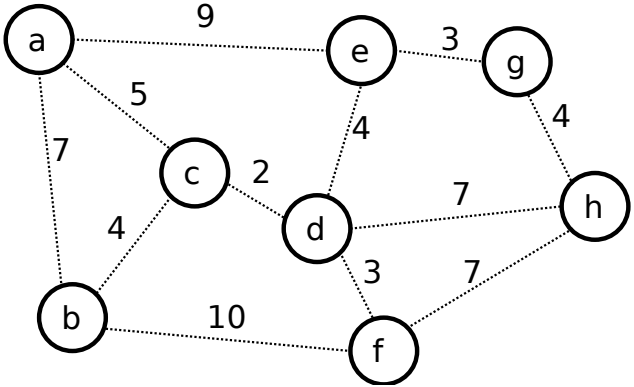
- Sommets : ensemble de sites à desservir.
- Arêtes : liens potentiels entre les sites pouvant être reliés.
- Coût des arêtes : coût d'installation d'un lien de transmission.
- Aucune redondance (si ceci devenait un objectif, ce ne serait plus un ARM).


RISQ
RÉSEAU D'INFORMATIONS SCIENTIFIQUES DU QUÉBEC
www.risq.qc.ca
(RCE) EEYOU (Projeté)
Sept-Îles
Baie-Comeau
Forestville
St-Félicien
Saguenay
Matane
RCGIM
Rimouski
Rivière-du-Loup
Rouyn-Noranda
GIRAT
Québec
Lévis
Mont-Laurier
Shawinigan
Trois-Rivières
St-Georges
St-Jérôme
Laval
St-Hyacinthe
Gatineau
Ottawa
Valleyfield
Longueuil
Sherbrooke
Rouyn-Noranda
(par l'Ontario)
ORION
Montréal
GLORIAD
Internet 2
Géant / RENATER2
2012 © Copyright RISQ inc.
Légende
Réseau de fibre optique du RISQ
Circuits en location du RISQ
Réseaux partenaires
Réseau CANARIE et réseaux internationaux
Transport optique ORION
Chemins de fibre projetés
PoP CANARIE au Québec

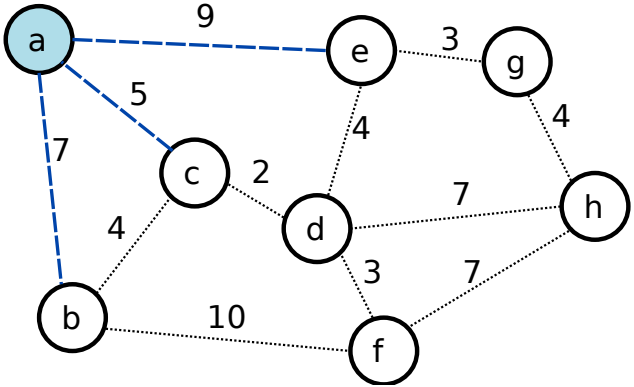
Ébauche de l'algorithme de Prim-Jarnik

1. PRIM_JARNIK($G = (S, A)$)
2. $S' \leftarrow \{s\}$ où v est un sommet choisi arbitrairement dans S
3. $A' \leftarrow \{\}$
4. tant que $S' \neq S$
5. $e \leftarrow (s_1, s_2) \in A$ tel que $s_1 \in S'$, $s_2 \notin S'$ et $cout(e)$ est minimal
6. ajouter s_2 à S'
7. ajouter e à A'
8. retourner $G' = (S', A')$

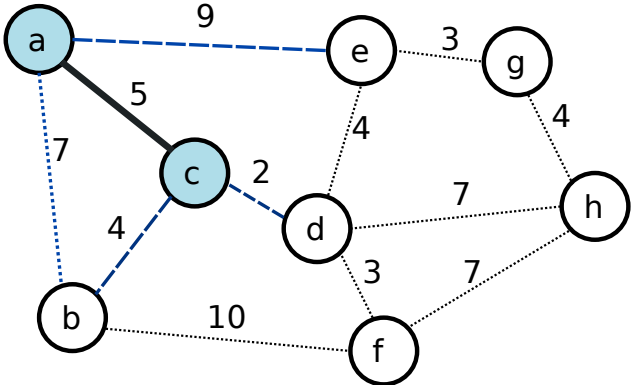
Étape 0



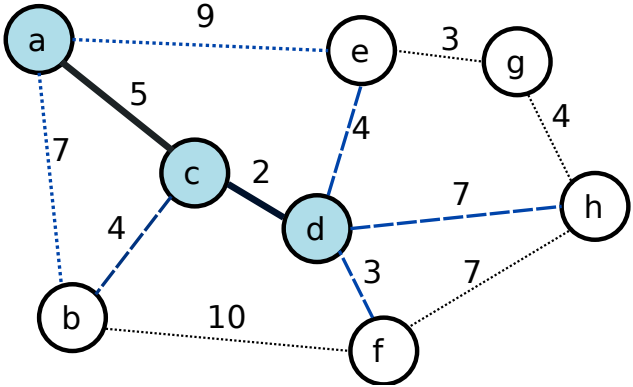
Étape 1



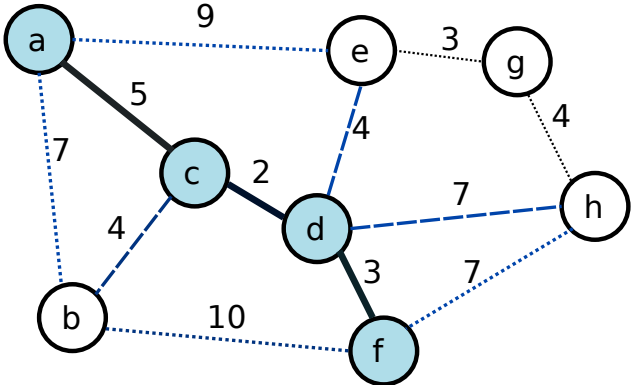
Étape 2



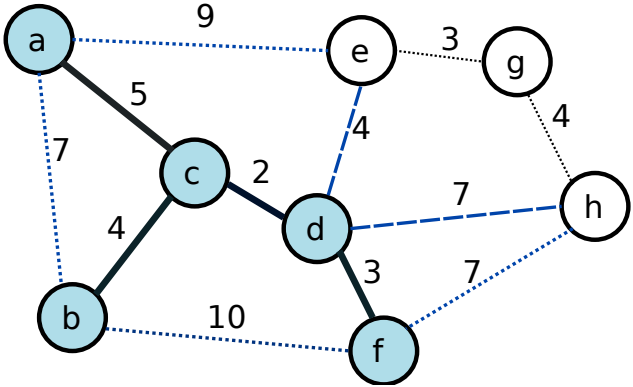
Étape 3



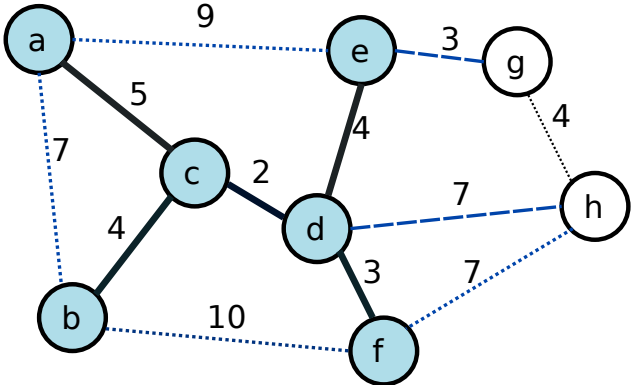
Étape 4



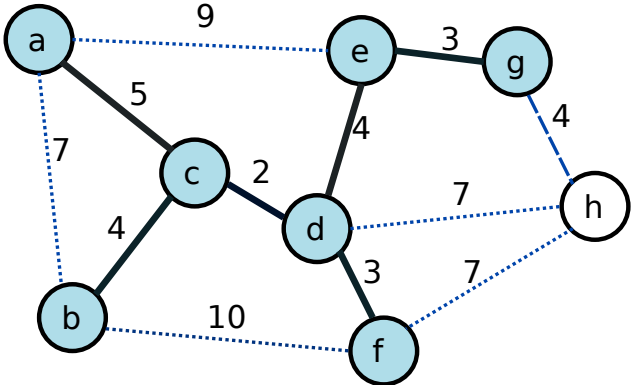
Étape 5



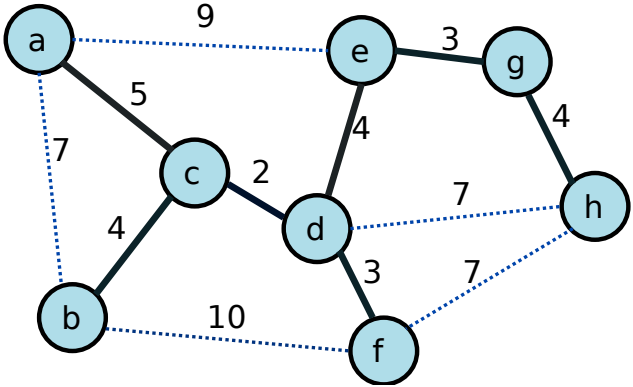
Étape 6



Étape 7



Étape 8



Comment implémenter efficacement l'algorithme de Prim-Jarnik ?

- Nécessité de choisir le prochain sommet qui peut être relié par l'arête la moins coûteuse permettant d'étendre la composante connexe en construction.
- Donc, il faut ordonner les sommets ...
- Quelle structure de données ?

Comment implémenter efficacement l'algorithme de Prim-Jarnik ?

- Nécessité de choisir le prochain sommet qui peut être relié par l'arête la moins coûteuse permettant d'étendre la composante connexe en construction.
- Donc, il faut ordonner les sommets ...
- Quelle structure de données ?
- File prioritaire, monceau (*heap*).

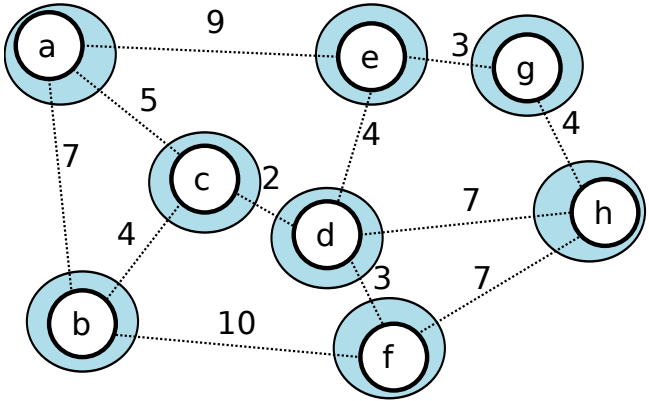
Algorithme de Prim-Jarnik

1 : **fonction** PRIM_JARNIK($G = (S, A)$)
2 : $s \leftarrow$ sommet de S choisi arbitrairement
3 : $Q \leftarrow$ FilePrioritaire<Sommet> qui extrait selon *priorite*
4 : **pour tout** $v \in S$ **faire**
5 : $arete[v] \leftarrow ?$
6 : $priorite[v] \leftarrow \infty$
7 : **pour tout** arête $(s, v) \in A$ incidente à s **faire**
8 : $arete[v] \leftarrow (s, v)$
9 : $priorite[v] \leftarrow c(s, v)$
10 : Q.INSÉRER(v)
11 : $S' \leftarrow \{s\}$, $A' \leftarrow \{\}$
12 : **tant que** Q n'est pas vide **faire**
13 : $u \leftarrow Q$.ENLEVERMINIMUM
14 : S'.AJOUTER(u), A'.AJOUTER($arete[u]$)
15 : **pour tout** arête $(u, v) \in A$ **faire**
16 : **si** $v \notin S'$ et $c(u, v) < priorite[v]$ **alors**
17 : $arete[v] \leftarrow (u, v)$
18 : $priorite[v] \leftarrow c(u, v)$
19 : Q.INSÉRER(v)
20 : retourner $G' = (S', A')$

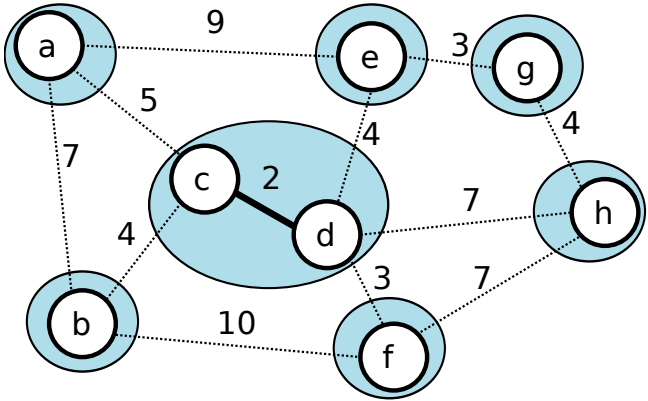
Algorithme de Kruskal

1 : **fonction** KRUSKAL($G = (S, A)$)
2 : Trier les arêtes de A par ordre croissant de poids
3 : **pour tout** $s \in S$ **faire**
4 : $C(s) \leftarrow \{s\}$
5 : $A' \leftarrow \{\}$
6 : **pour tout** $(u, v) \in A$, en ordre croissant de poids **faire**
7 : **si** $|A'| = |S| - 1$ **alors retourner** $G' = (S, A')$
8 : **si** $C(u) \neq C(v)$ **alors**
9 : $A' \leftarrow A' \cup \{(u, v)\}$
10 : Fusionner $C(u)$ et $C(v)$

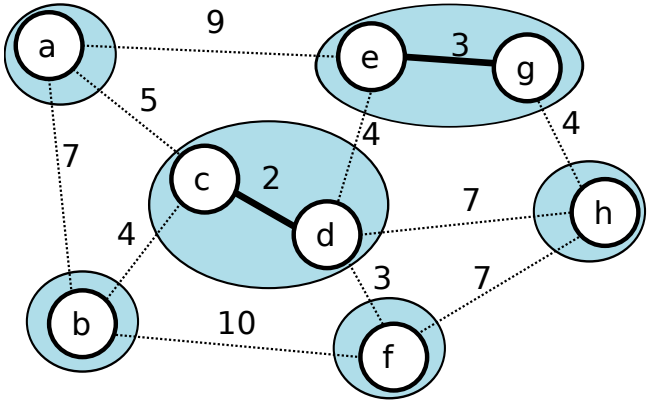
Étape 0



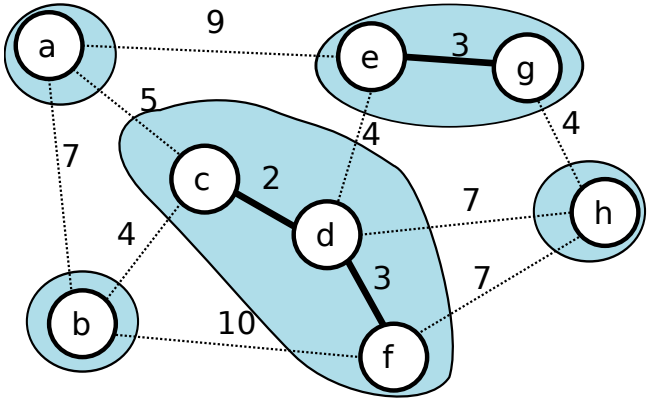
Étape 1



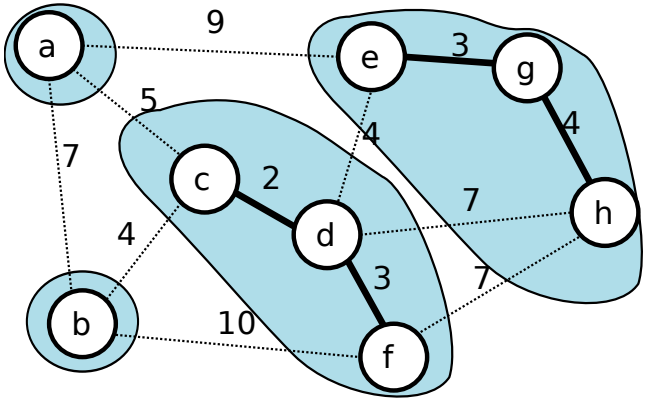
Étape 2



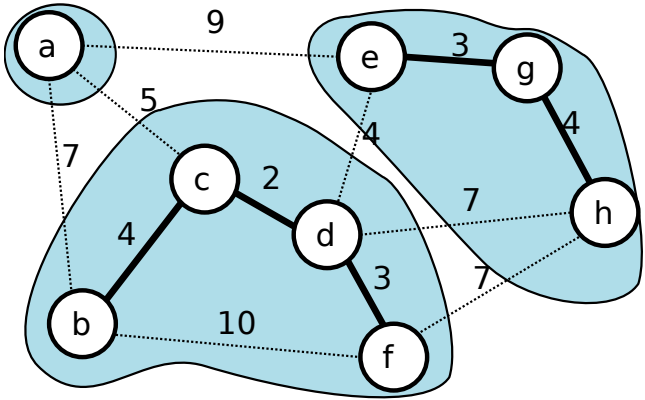
Étape 3



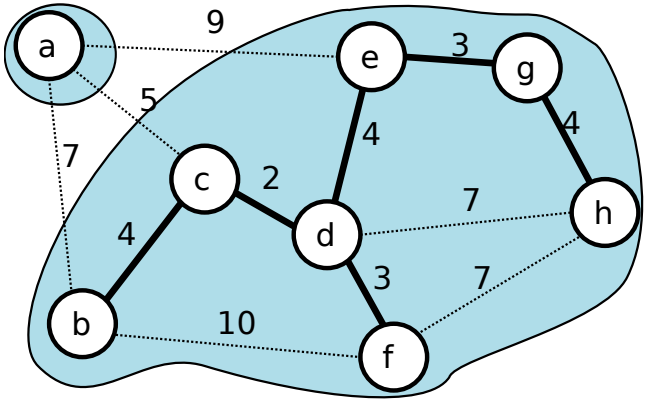
Étape 4



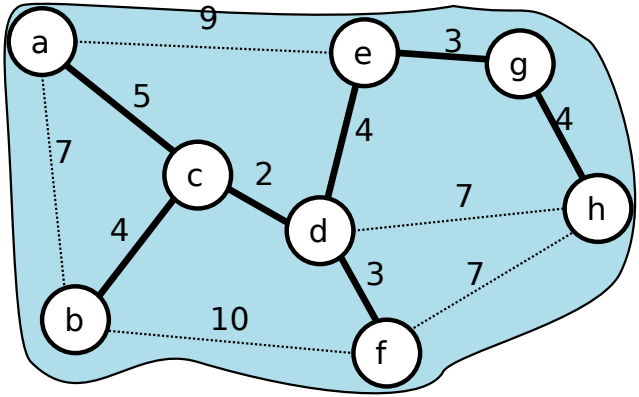
Étape 5



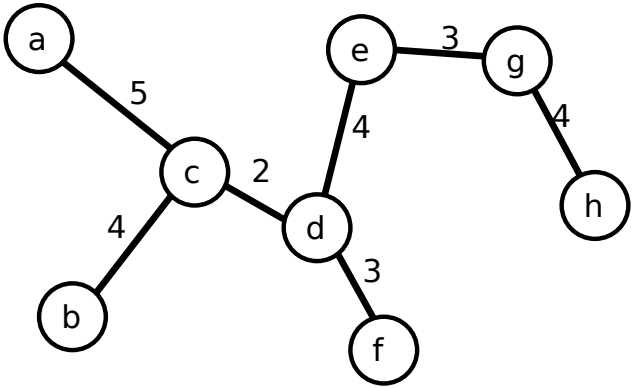
Étape 6



Étape 7



Étape 8



Comment implémenter efficacement l'algorithme de Kruskal ?

- Comment vérifier rapidement si deux sommets sont dans le même ensemble ?
- Comment fusionner efficacement deux ensembles ?

Comment implémenter efficacement l'algorithme de Kruskal ?

- Comment vérifier rapidement si deux sommets sont dans le même ensemble ?
- Comment fusionner efficacement deux ensembles ?
- Nouvelle structure de données : ensembles disjoints (*Union-Find*)
- Trois opérations :
  - `Créer(n)` : crée n ensembles contenant chacun un élément ;
  - `Union(u, v)` : fusionne les ensembles contenant u et v ;
  - `Find(u)` : retourne le représentant de l'ensemble contenant u.

Union-Find – Implémentation simple

UnionFind.h

```
1  class UnionFind {
2    public:
3      UnionFind(int n) : parent(n) {
4          for(int i = 0; i < n; ++i)
5              parent[i] = i;
6      }
7
8      void union(int a, int b);
9      int find(int v);
10
11   private:
12     std::vector<int> parent;
13 };
```

UnionFind.cpp

```
1  void union(int a, int b) {
2      a = find(a);
3      b = find(b);
4      if (a == b) return;
5
6      parent[b] = a;
7  }
8
9  int find(int v) {
10     if (parent[v] == v) return v;
11     return find(parent[v]);
12 }
```

Union-Find – Implémentation optimisée

UnionFind.h

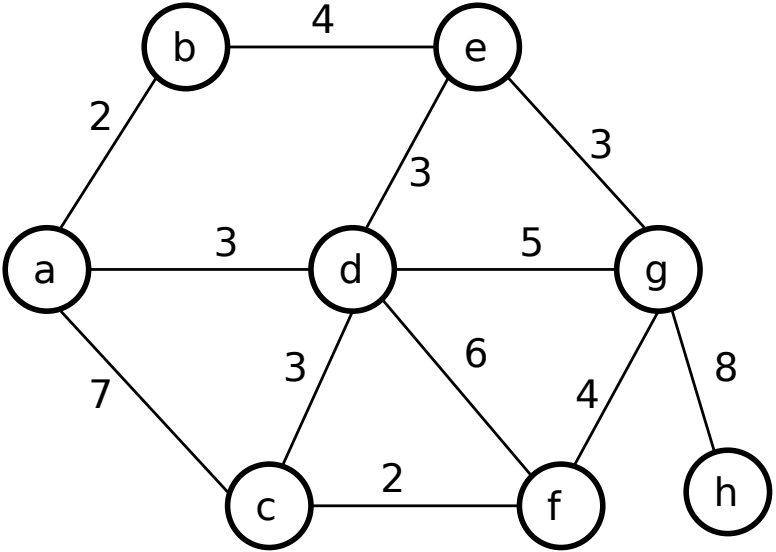
```
1  class UnionFind {
2    public:
3      UnionFind(int n)
4            : parent(n), taille(n, 1) {
5        for(int i = 0; i < n; ++i)
6            parent[i] = i;
7      }
8      void union(int a, int b);
9      int find(int v);
10
11   private:
12     std::vector<int> parent;
13     std::vector<int> taille;
14 };
```

UnionFind.cpp

```
1  void union(int a, int b) {
2      a = find(a);
3      b = find(b);
4      if (a == b) return;
5
6      if (taille[a] < taille[b]) swap(a, b);
7      parent[b] = a;
8      taille[a] += taille[b];
9  }
10
11 int find(int v) {
12     if (parent[v] == v) return v;
13     return parent[v] = find(parent[v]);
14 }
```

Implémentation de Kruskal avec Union-Find

1 : **fonction** KRUSKAL($G = (S, A)$)
2 : Trier les arêtes de A par ordre croissant de poids
3 : $C \leftarrow$ UNIONFIND($|S|$)
4 : $A' \leftarrow \{\}$
5 : **pour tout** $(u, v) \in A$, en ordre croissant de poids **faire**
6 : **si** $|A'| = |S| - 1$ **alors retourner** $G' = (S, A')$
7 : **si** C.FIND(u) $\neq$ C.FIND(v) **alors**
8 : $A' \leftarrow A' \cup \{(u, v)\}$
9 : C.UNION(u, v)


b
e
a
d
g
c
f
h
4
2
3
3
3
5
7
3
6
4
8
2