Graphes I

CHAPITRE 9(WEISS)

NOTES DE COURS, É. BAUDRY

NOTES DE COURS, S. HAMEL

Graphes

- Structure de données non linéaire.
- Représentation de relations entre des paires d’objets
- Applications
  - Cartes

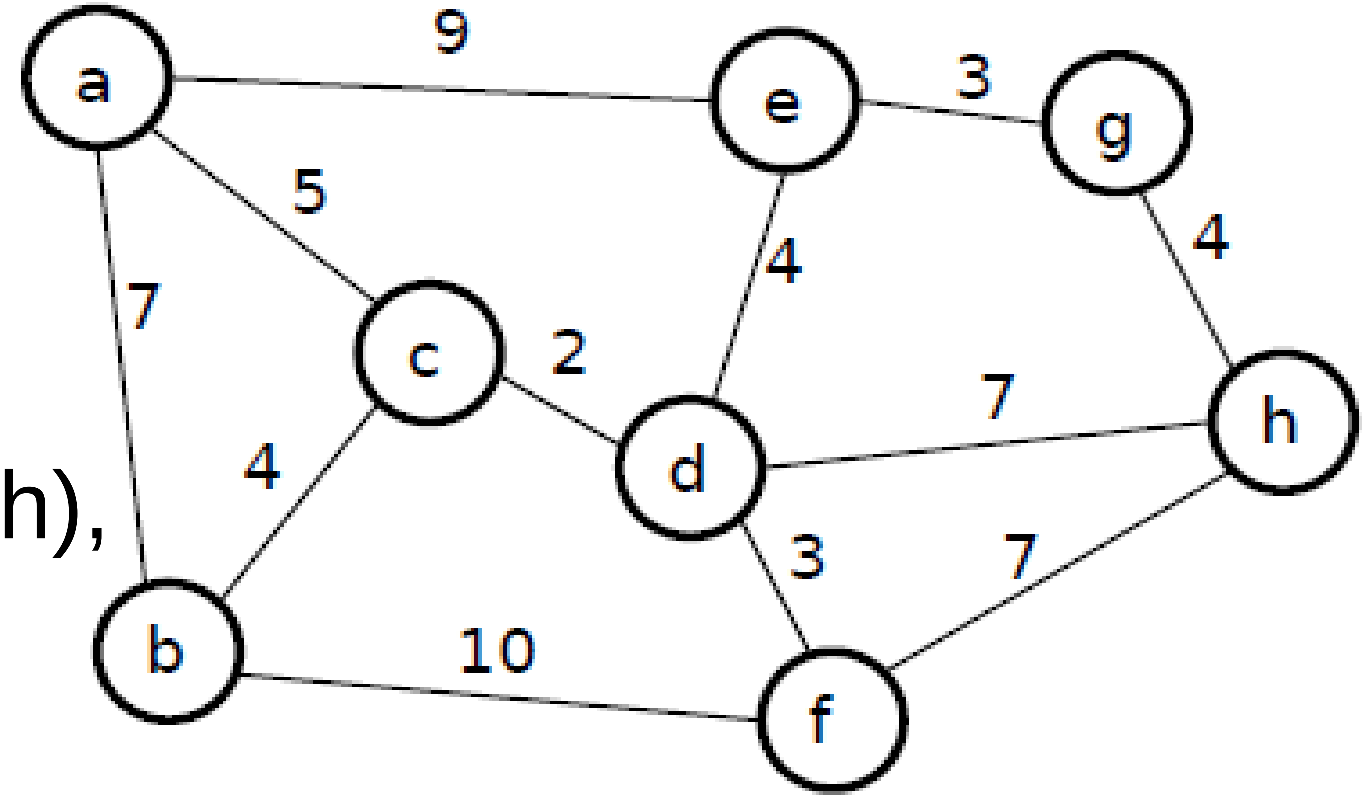
Définition formelle d’un graphe

- Un **graphe** est représenté formellement par G = (V;E) où V est un ensemble de **sommets** **(noeuds)** et E un ensemble d’**arêtes** **(arcs)**
- Un **sommet** est une représentation abstraite d’un objet
- Une **arête** représente une **relation** binaire entre deux objets

Définition formelle d'un graphe

G = (V;E)
V = {a, b, c, d, e, f, g, h}

E = { (a,b), (a,c), (a,e),
(c,b), (c,d), (e,d), (b,f), (f,h),
(e,g), (d,h), (d,f), (g,h) }

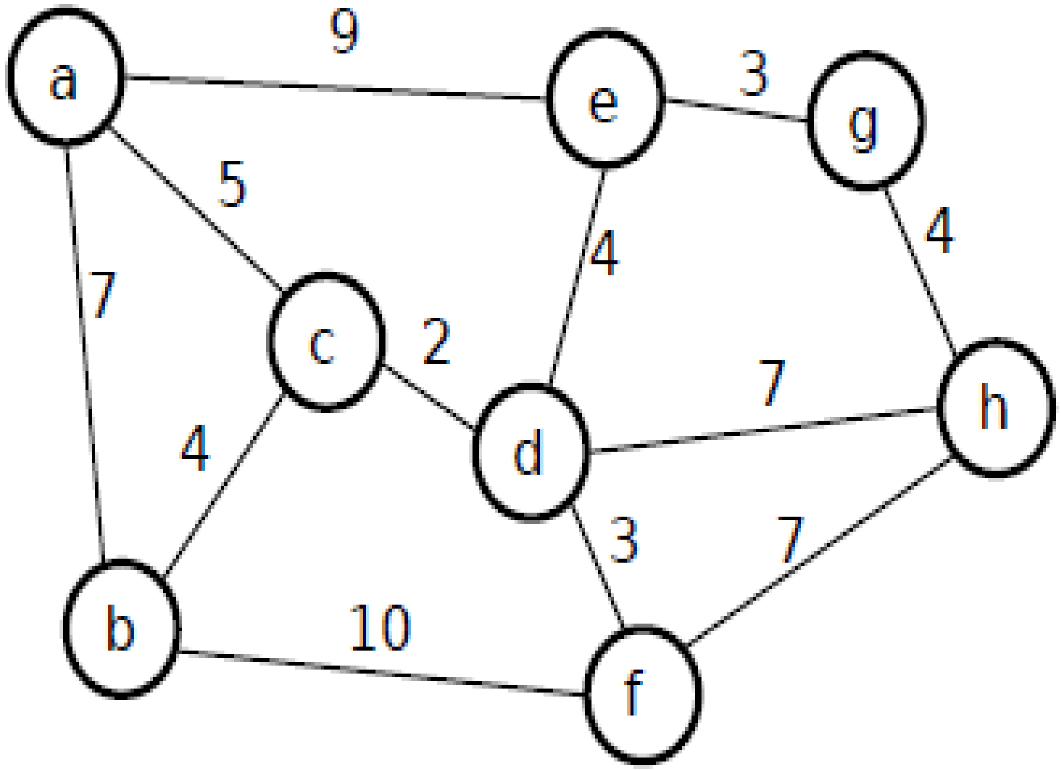


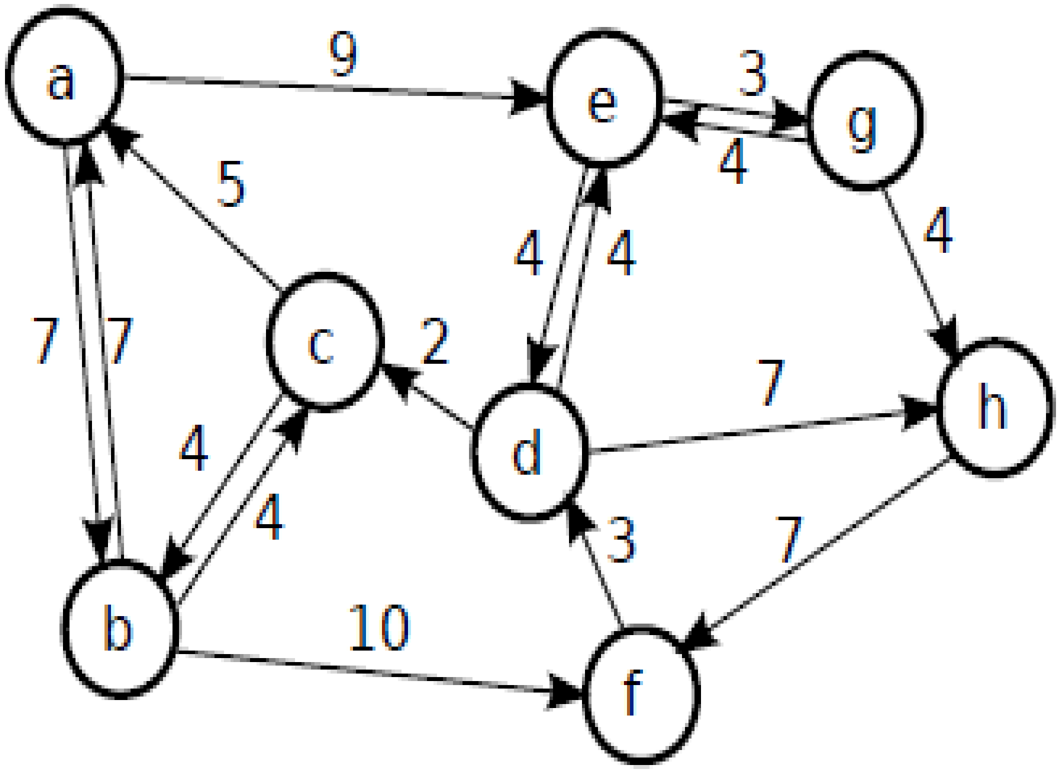
Sous graphe

Le graphe $G' = (V',E')$ est un **sous-graphe** de $G = (V,E)$ ssi $V' \subset V$, $E' \subset E$ et $\forall\, e = (x,y) \in E'$ $x \in V'$, $y \in V'$

Graphes

Orienté versus non orienté



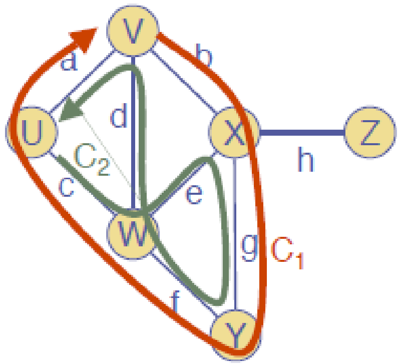


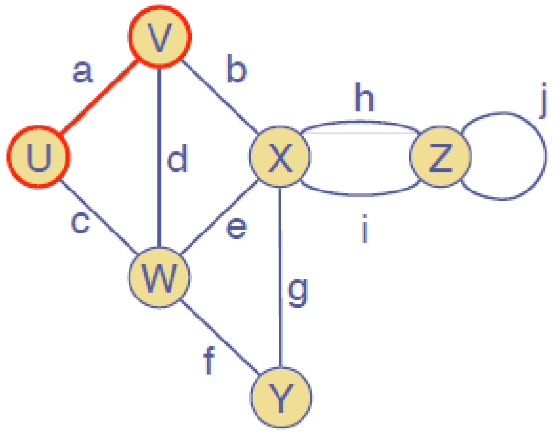
Chemin

- Un **chemin** est une séquence de sommets / arêtes dans un graphe
- La **longueur** d’un chemin peut être le nombre de sommets ou le nombre d’arêtes

Cycle

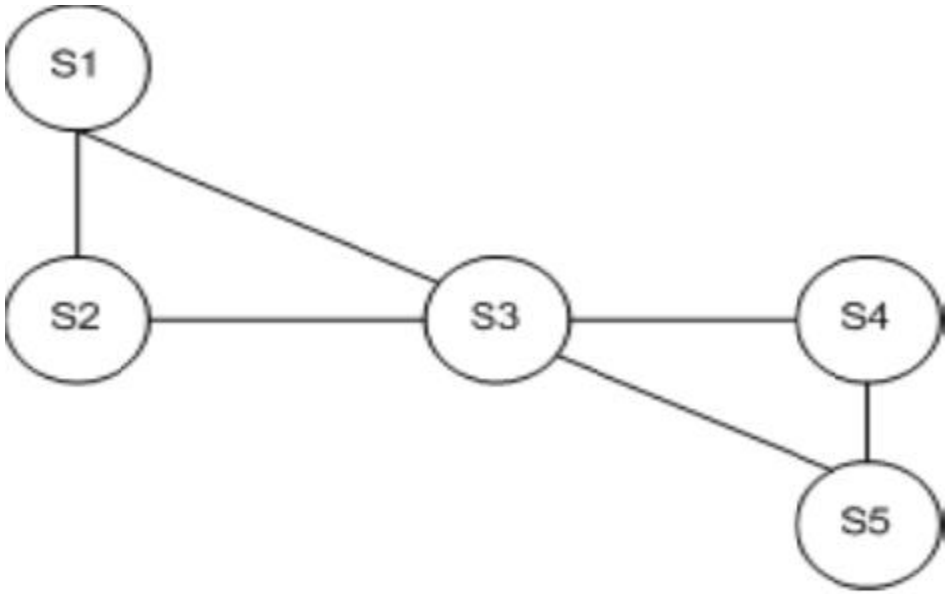
- Un **cycle** est chemin dans un graphe dont le sommet de départ est le même que le sommet d’arrivé
- Une **boucle** est une arête dont les sommets de départ et d’arrivée sont les mêmes. Une boucle est un cycle de longueur 1
- Un graphe est dit **acyclique** si et seulement s’il ne contient aucun cycle





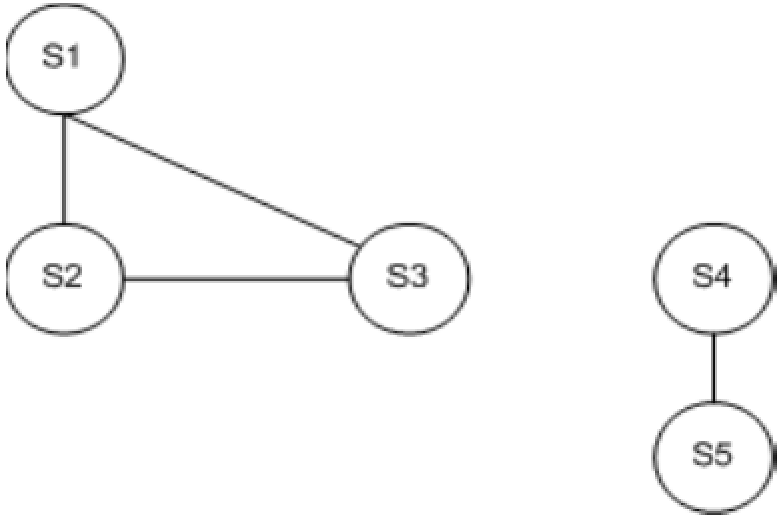
Graphes connexes

- Un graphe non orienté est **connexe** (ou **connecté**) s'il existe au moins un chemin entre toutes les paires de sommets
- Un graphe non orienté où on peut aller de tout sommet vers tous les autres sommets



Graphes connexes

- Une **composante connexe** est un sous-graphe connecté et maximal
- Un graphe G = (V,E) non connecté est composé de plusieurs composantes connexes
- Graphe non connexe avec deux composantes connexes

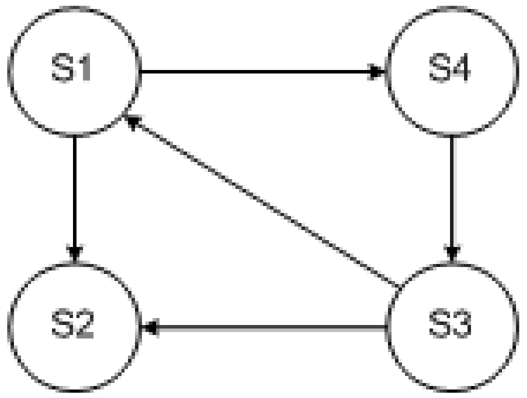


Graphes fortement connexes

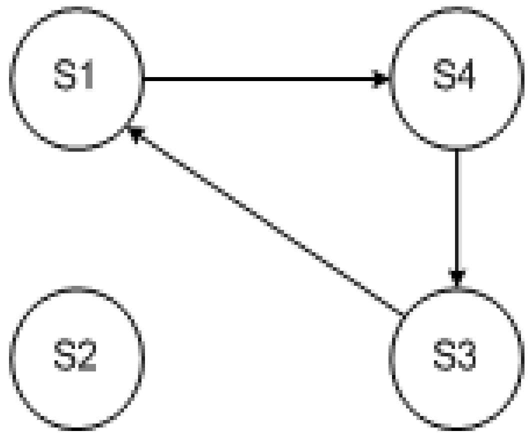
- Un graphe orienté est **fortement connexe** (ou **fortement connecté**) si et seulement si pour toute paire de sommets (a, b) il existe un chemin de a à b, et de b à a
- Un graphe orienté où on peut aller de tout sommet vers tous les autres sommets en passant éventuellement par un ou plusieurs sommets intermédiaires

Graphes fortement connexes

Graphe non fortement connexe

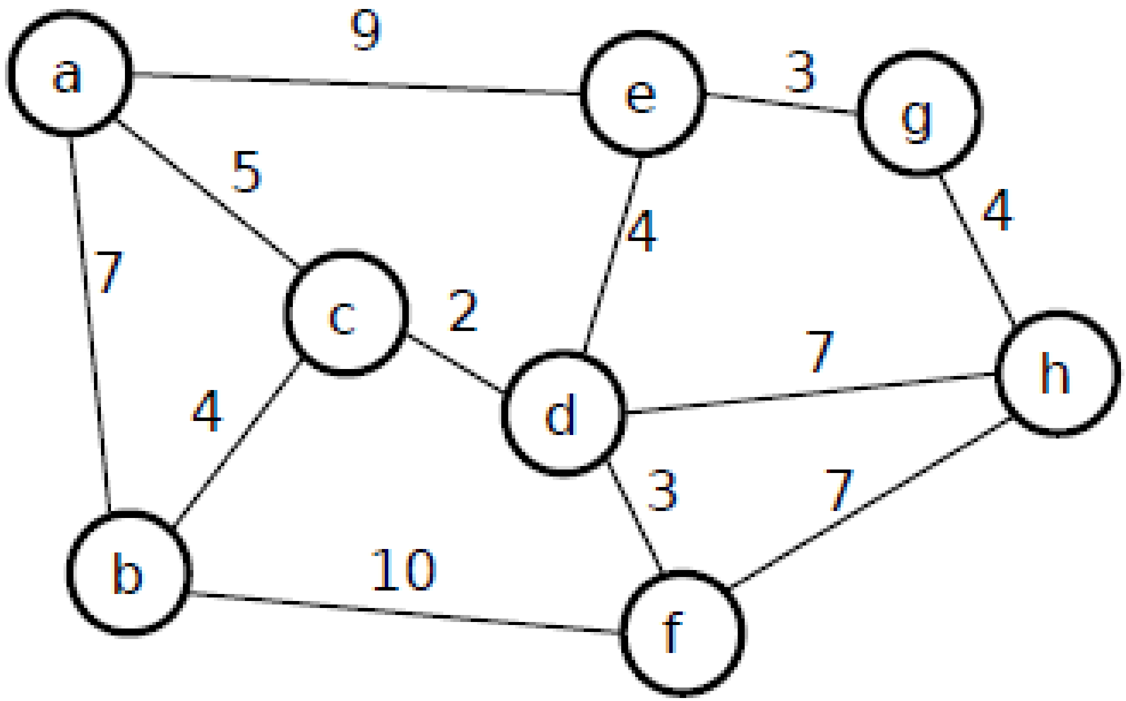


Composantes fortement connexes



Étiquette

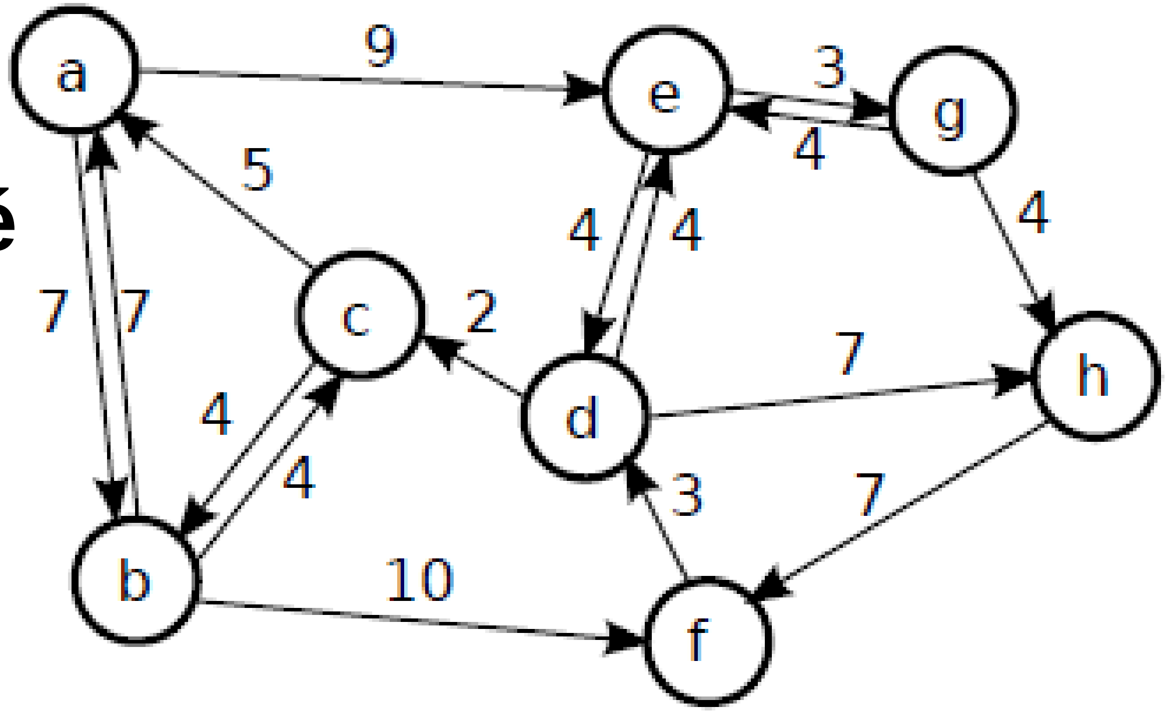
- Une **étiquette** indique une propriété d'une arête
- Poids d'une arête



Étiquette

- Dans un graphe non orienté, le **degré** d'un sommet v ∈ V, noté *deg(v)*, est le nombre d'arêtes qui y sont reliées

- Dans un graphe orienté, on fait la distinction entre le **degré sortant** et le **degré entrant**, qui sont respectivement notés *deg_out (v)* et *deg_in(v)*



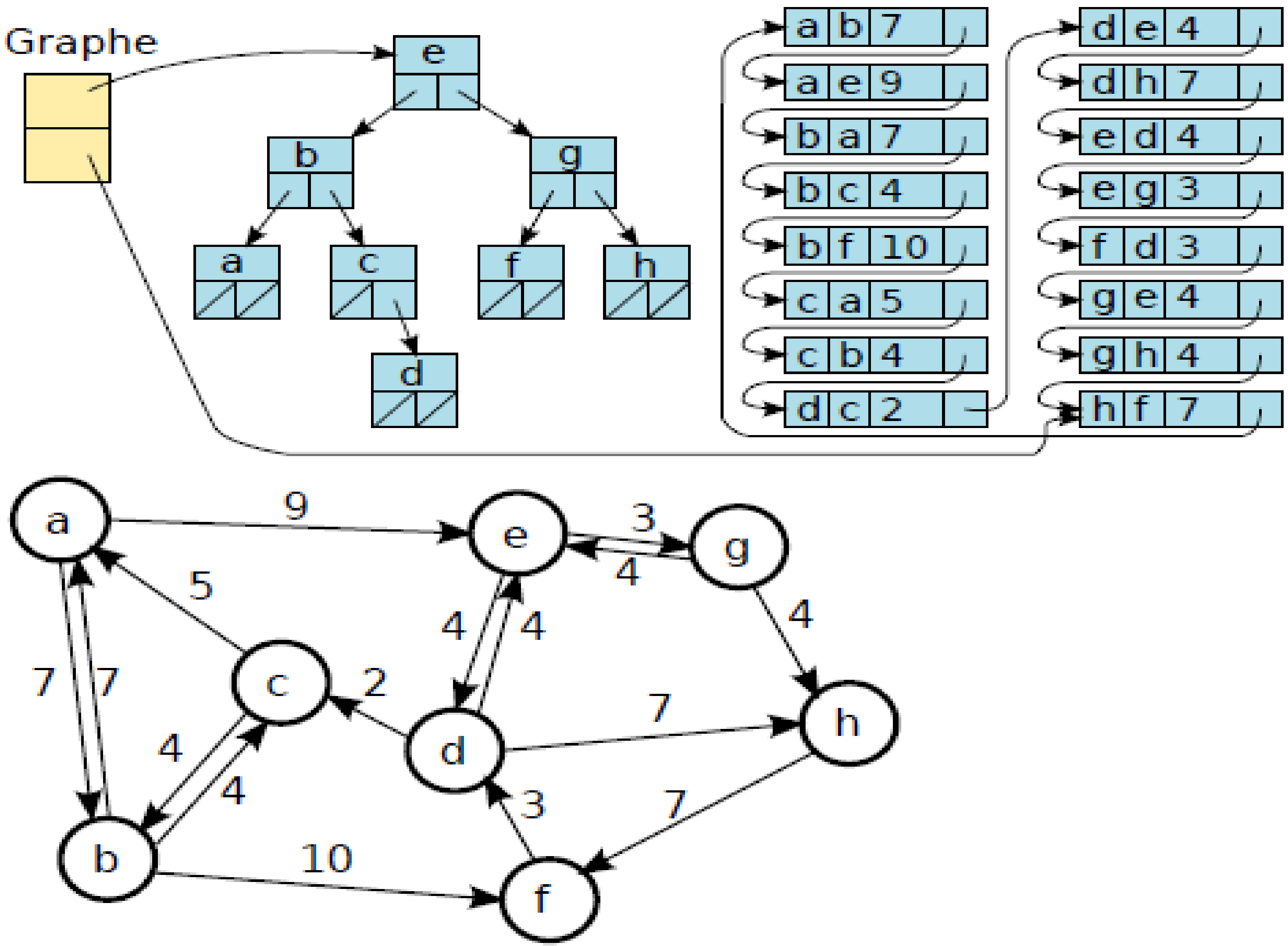
Arbre, Forêt

- Un **arbre** est un cas particulier de graphe non orienté
- Un arbre est un graphe ayant une seule composante connexe tel que le nombre de sommets est égal au nombre d’arêtes plus un ($|V| = |E|+1$), et où tous les sommets sont accessibles à partir d’un sommet qualifié de **racine**
- Un arbre est un graphe qui est forcément acyclique
- Un graphe qui est composé d’un ensemble d’arbres est appelé une **forêt**

Représentations

Ensemble de sommets et collection d’arêtes

```
class Graphe {
class Arete{
  S depart, arrivee;
  A etiquette;
};
Ensemble<Sommet> sommets;
Collection<Arete> aretes;
//...
};
```



Représentations

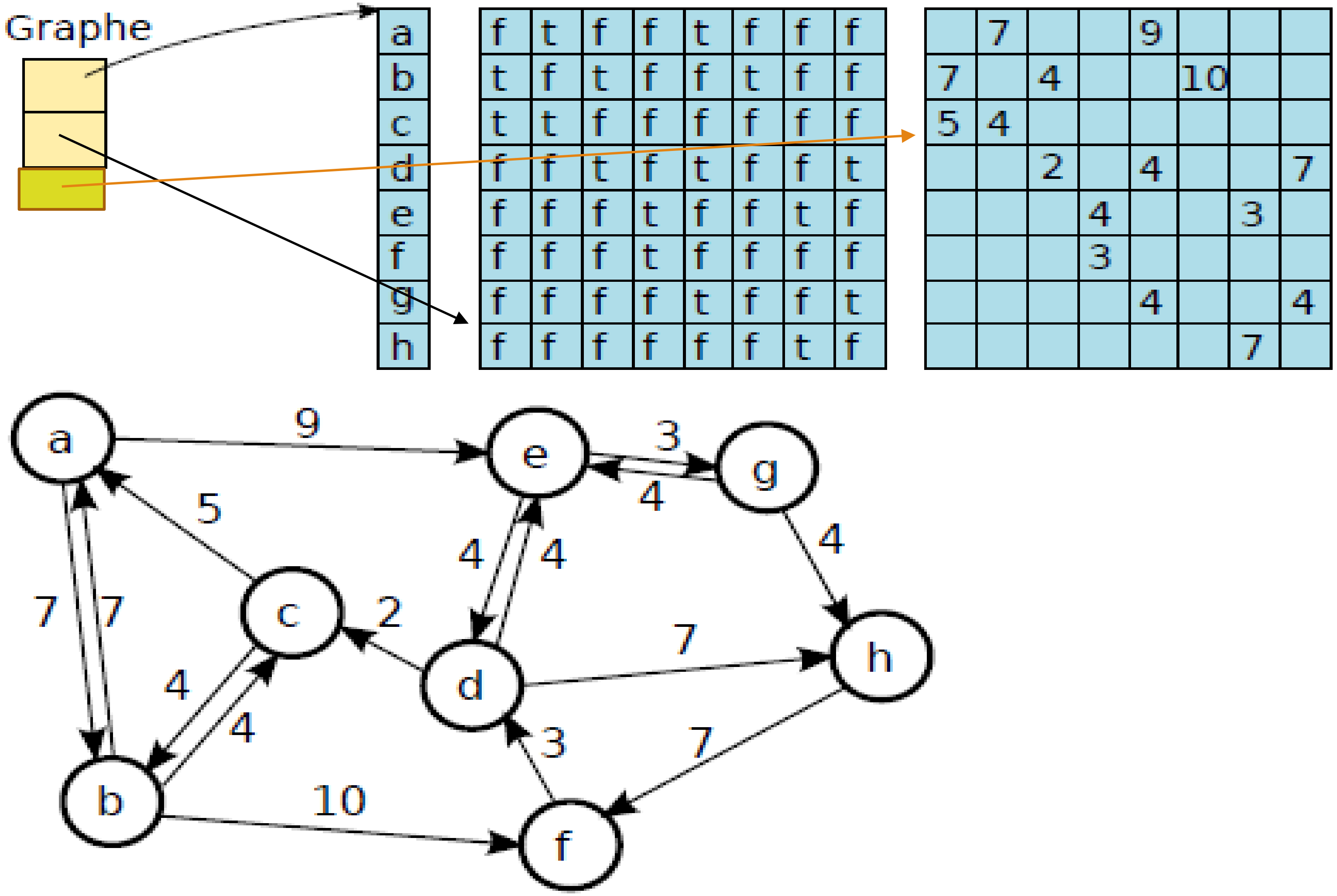
- Deux façons classiques de représenter un graphe $G = (V, E)$
  - Ensemble de listes d'adjacences
    - Graphes peu denses -> $|E| \ll |V|^2$
  - Matrice d'adjacences
    - Graphe est dense -> $|E| \approx |V|^2$

Représentations

Ensemble de sommets
et collection d’arêtes

Matrice d’adjacence (1)

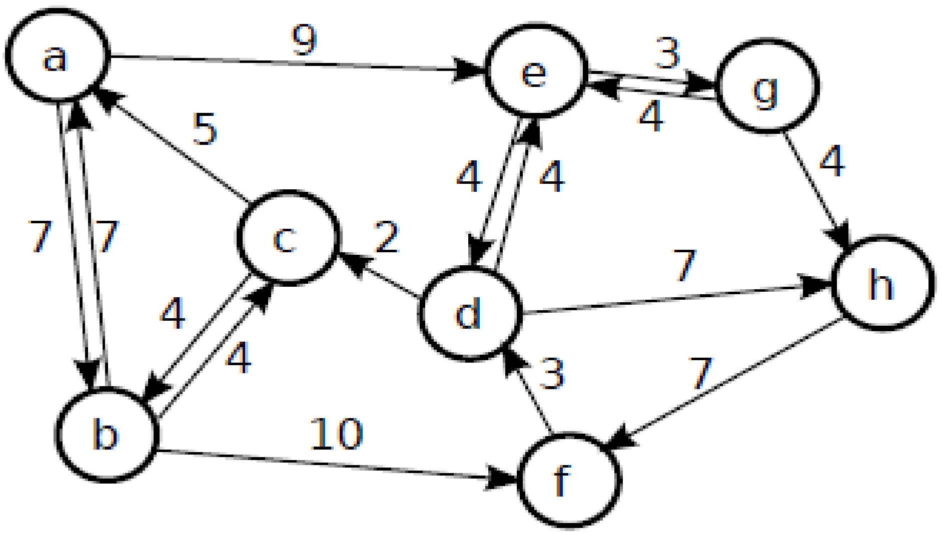
```
class Graphe {
Tableau<S> sommets;
Tableau2D<bool> relations;
Tableau2D<A> etiquettes;
};
```

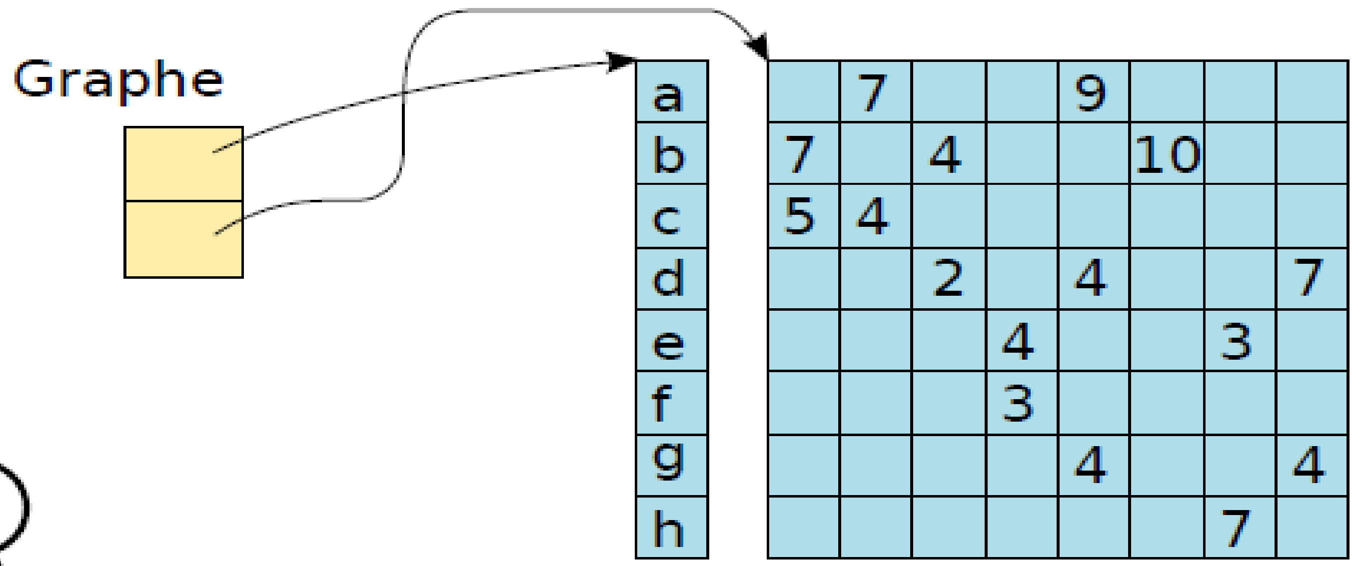


Représentations

Ensemble de sommets et collection d’arêtes

Matrice d’adjacence (2)
Mémoire – $\Theta(|V|^2)$





a		7			9			
b	7		4			10		
c	5	4						
d			2		4			7
e				4			3	
f				3				
g					4			4
h							7	

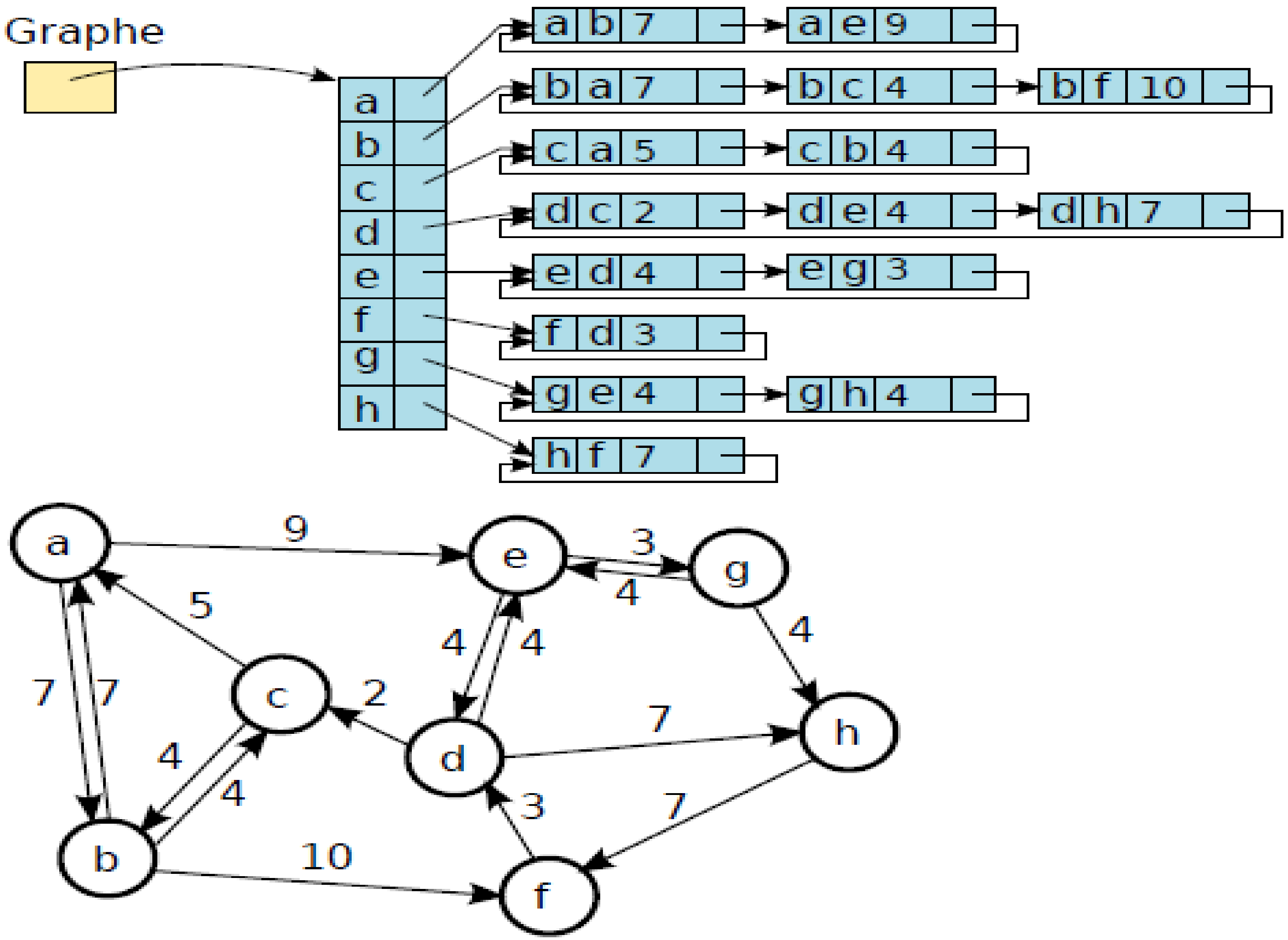
```
class Graphe {
Tableau<S> sommets;
Tableau2D<A> etiquettes;
static A  AUCUNE_RELATION;
}
```

Représentations

Listes d'adjacence

Ensemble de sommets et collection d'arêtes

```
class Graphe {
class Arete{
S depart, arrivee;
A valeur;
};
class Sommet {
S valeur;
Liste<Arete>
aretesSortantes;
}
Tableau<Sommet>
sommets;
```



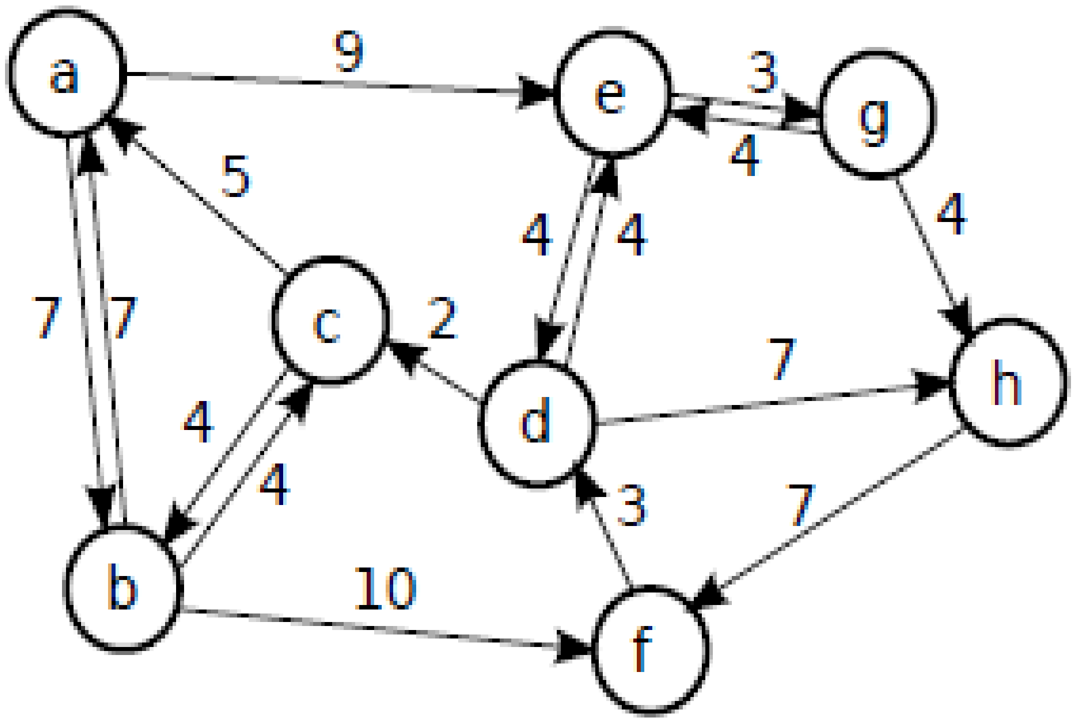
Représentations

Ensemble de sommets et collection d’arêtes

Dictionnaire avec liste d’adjacences

Exercice : représentation mémoire

```
class Graphe {
class Arete{
S depart, arrivee;
A valeur;
};
class Sommet {
Liste<Arete> aretesSortantes;
};
TreeMap<S, Sommet> sommets;
//...
}
```



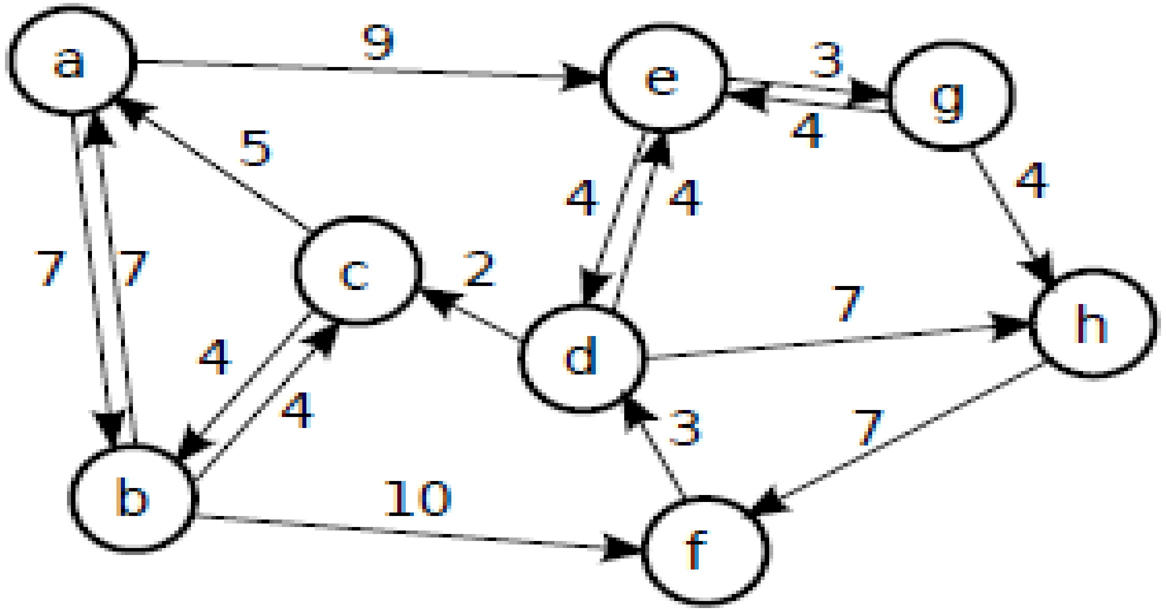
Représentations

Ensemble de sommets et collection d’arêtes

Exercice : représentation mémoire

Dictionnaire de dictionnaires d’adjacences

```
class Graphe {
class Sommet {
TreeMap<S, A> aretesSortantes;
};
TreeMap<S, Sommet> sommets;
//...
```

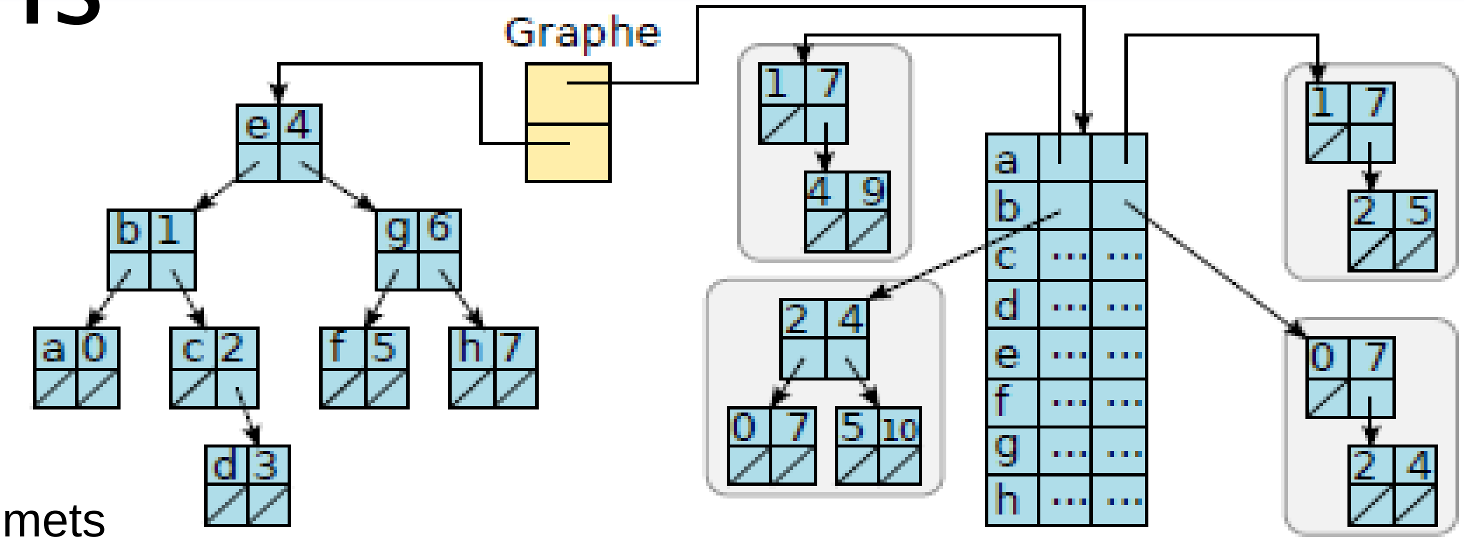


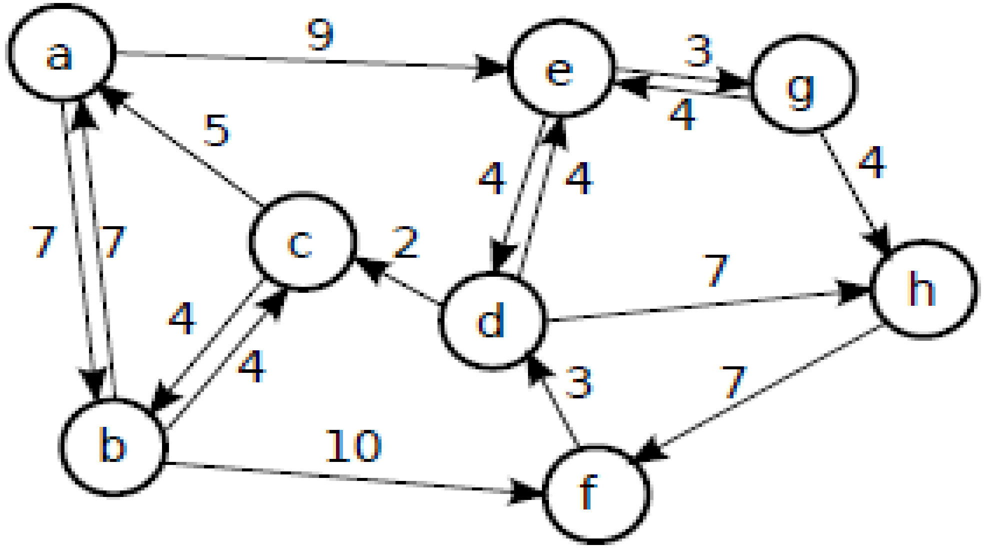
Représentations

Ensembles d’adjacence avec des indices

Ensemble de sommets et collection d’arêtes

```
class Graphe {
class Sommet{
S s; // optionnel
// e(v,w) sortante. s = v
// clé – indice de w dans une table des sommets
// valeur – poids de e sortante
// Exemple: (1,7) = 1 => b; e = (a,b) poids 7
TreeMap aretesSortantes;
TreeMap aretesEntrantes; //optionel
};
//clé – sommet, valeur – indice de sommet
TreeMap indices;
Tableau<Sommet> sommets;
//...
}
```





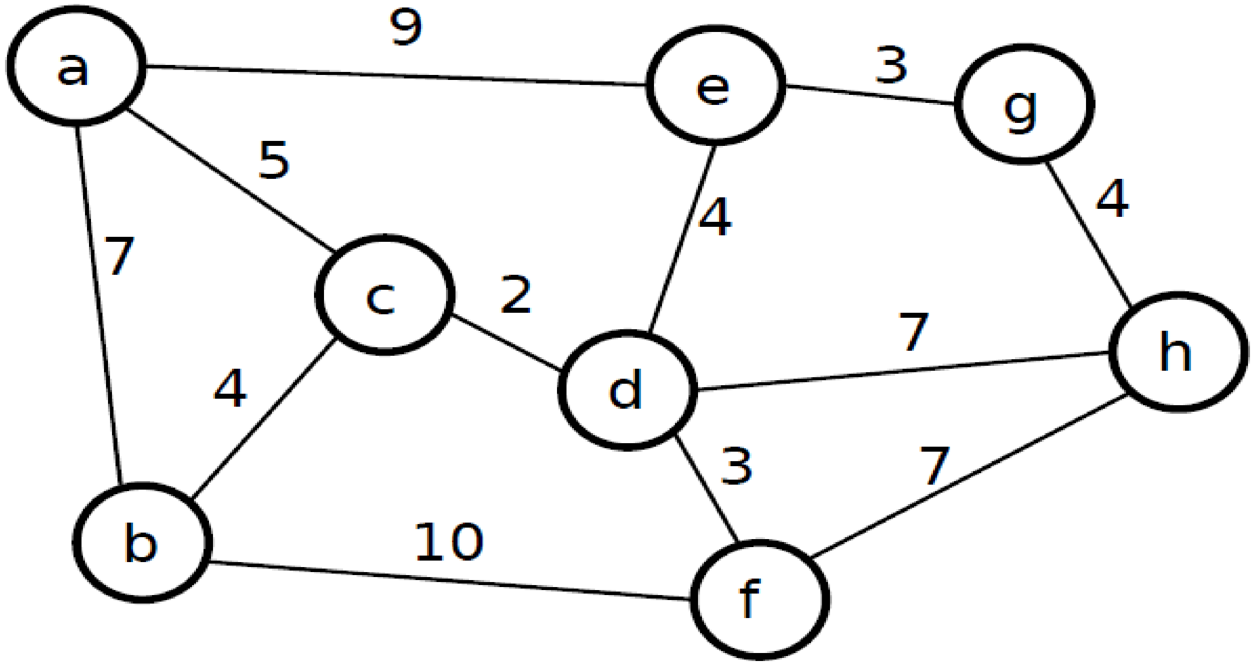
Parcours

- Parcourir un graphe revient à emprunter de façon systématique les arcs du graphe, pour en visiter les sommets
- Un algorithme de parcours permet de mettre en évidence plusieurs caractéristiques de la structure du graphe
- Types de parcours
  - Recherche en profondeur
  - Recherche en largeur

Parcours

Recherche en profondeur

1. RECHERCHEPROFONDEUR(G = (V,E), v ∈ V)
2. v.visité ← vrai
3. pour toute arête e ∈ v.aretesSortantes()
4. w ← e.arrivee
5. si !w.visité
6. RechercheProfondeur(G, w)



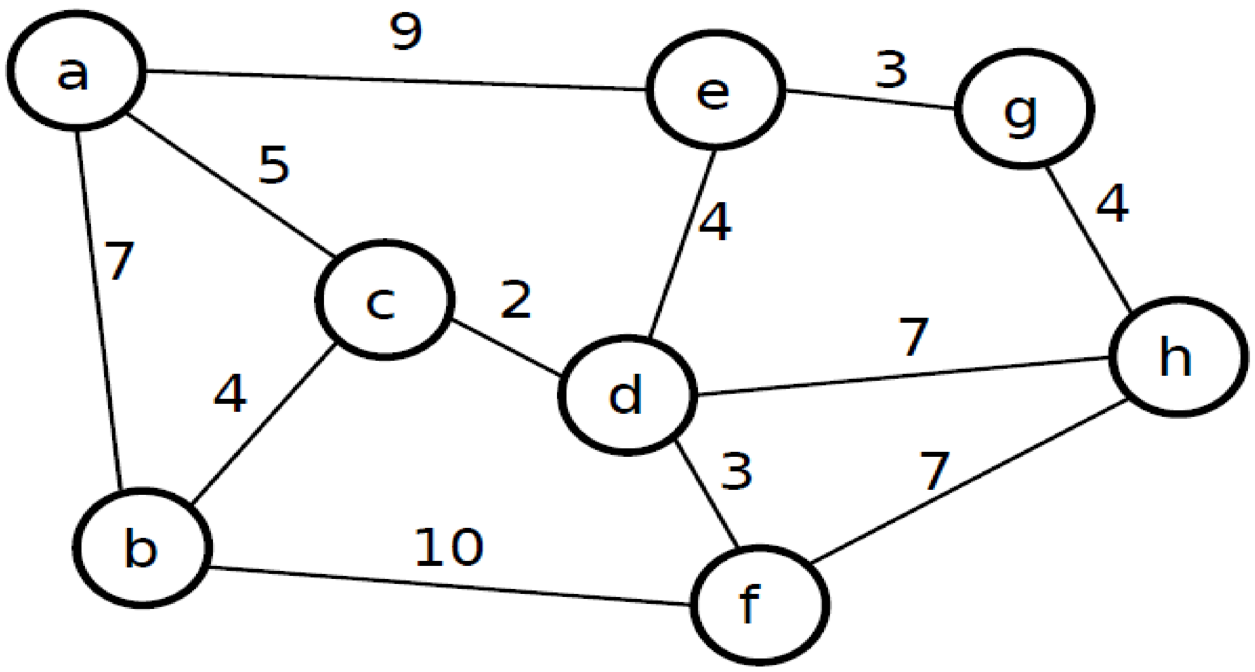
Parcours en profondeur

- Prends un temps en $O(n + m)$ pour un graphe avec n sommets et m arêtes
- peut être modifié pour résoudre d'autres problèmes sur les graphes
  - Trouver et retourner un chemin entre deux sommets donnés
  - Trouver un cycle dans le graphe

Parcours

Recherche en largeur

1. RECHERCHELARGEUR(G = (V,E), v ∈ V)
2. file CRÉERFILE
3. v.visité ← vrai
4. file.ENFILER(v)
5. tant que !file.vide()
6.   s ← file.defiler()
7.   pour tout arête a = (s,s′) ∈ E
8.   si !s′.visité
9.   s′.visité ← vrai
10.   file.ENFILER(s′)



Parcours en largeur

- prends temps $O(n + m)$ sur un graphe avec n sommets et m arêtes

- peut être modifié pour résoudre d'autres problèmes sur les graphes
  - Découvrir et retourner d'un chemin entre deux sommets donnés avec le minimum d'arêtes
  - Trouve un cycle simple dans le graphe s'il y en a un

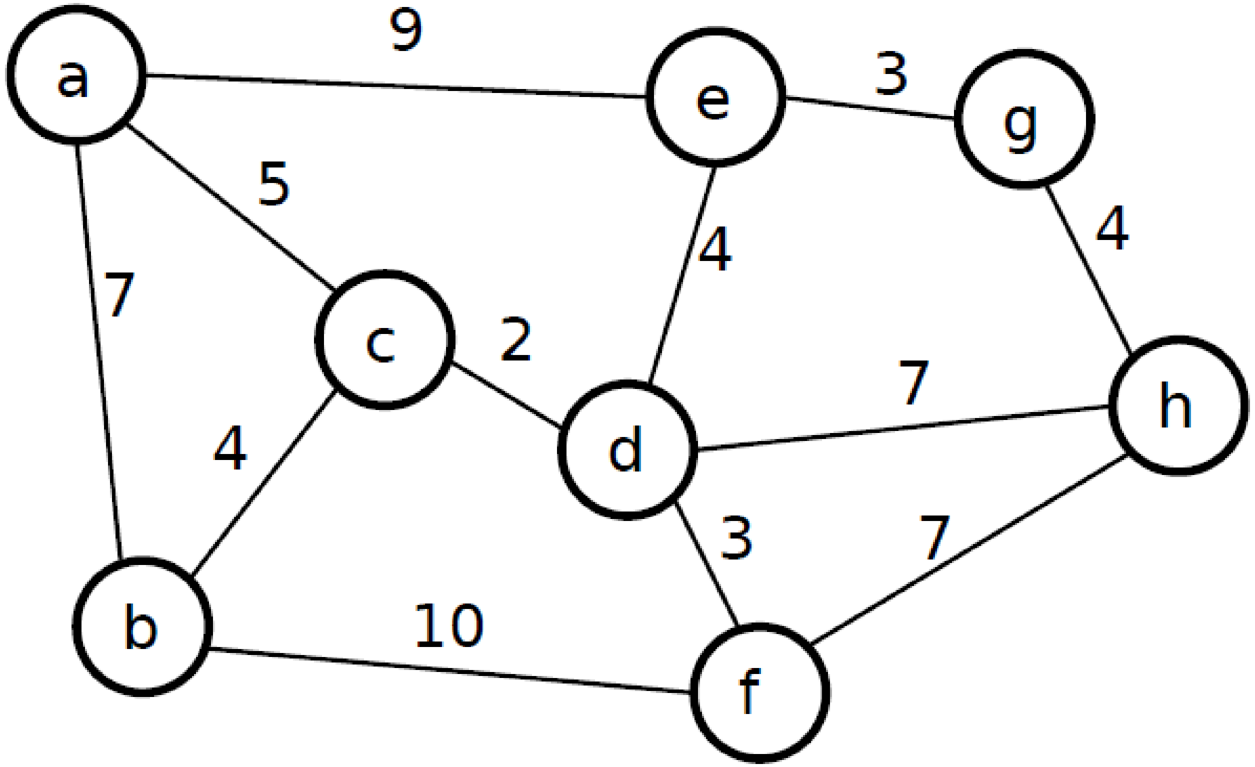
Graphes / Arbres de recouvrement à coût minimal (ARM)

Définitions

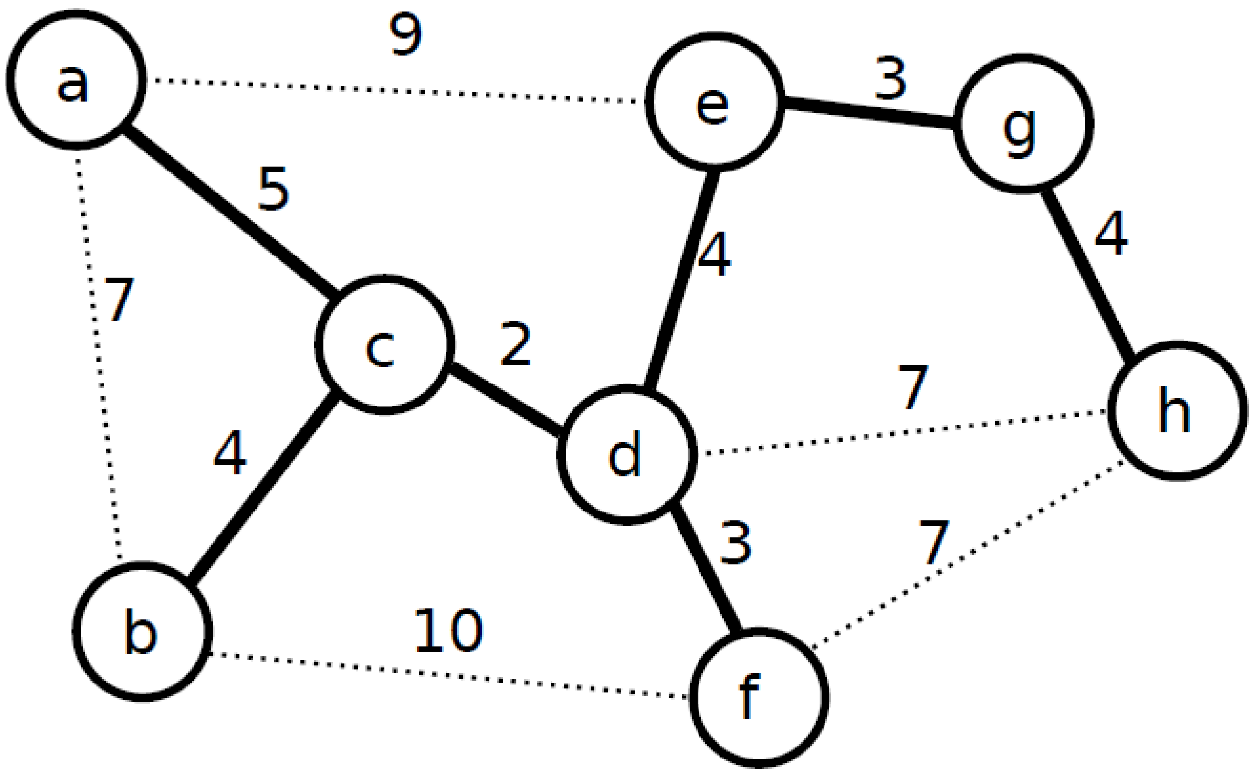
Arbre de recouvrement

- Sous-graphe connexe
- Acyclique
- Nombre de sommets = Nombre d’arêtes + 1
- Existence de chemin reliant toutes les paires de sommets
- = Arbre

Graphes / ARM



Graphes / ARM



Graphes / ARM

Définition

- Arbre de recouvrement à coût minimal
  - Arbre de recouvrement
  - La somme des coûts des arêtes sélectionnées est minimale
- La solution n’est pas nécessairement unique

ARM, applications

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)

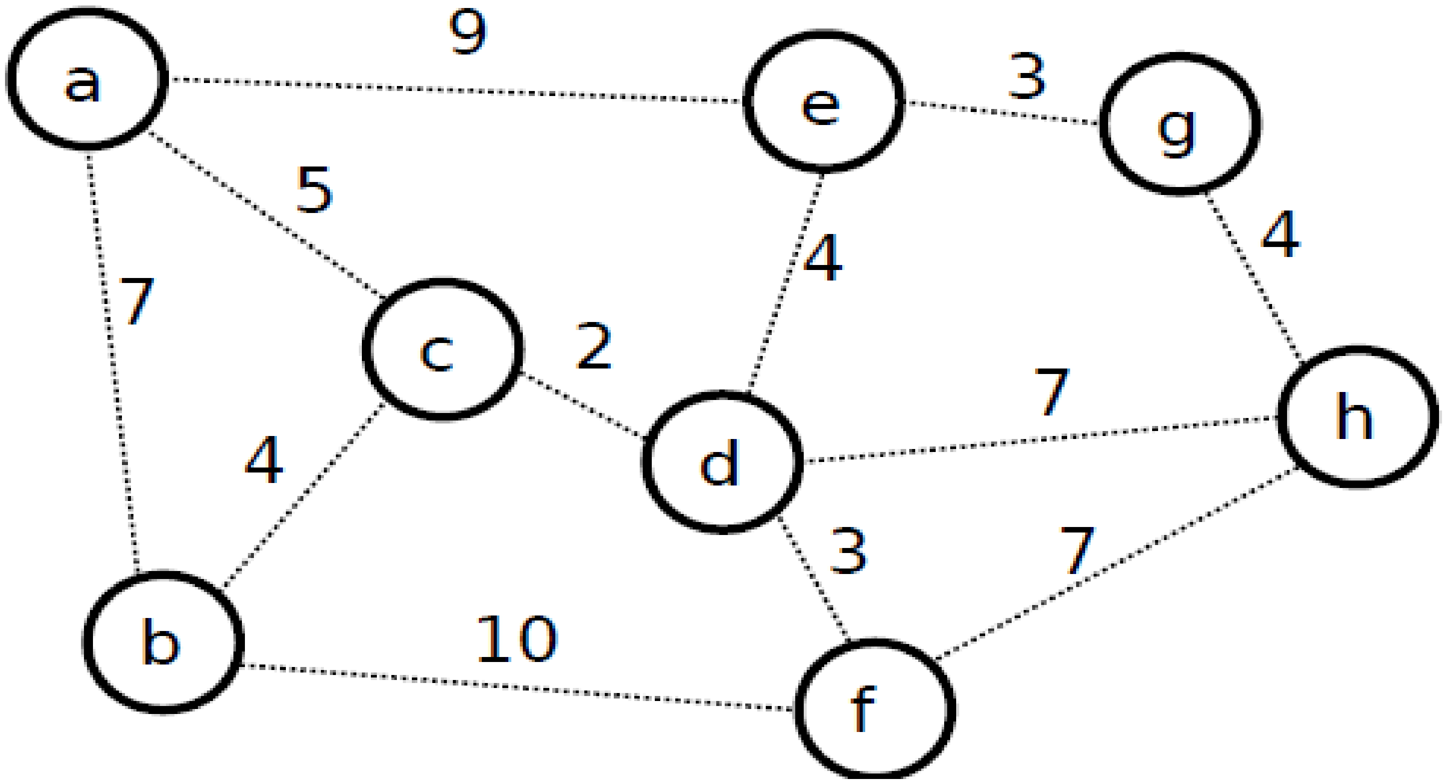
ARM, applications

ARM, ébauche de l’algorithme de Prim-Jarnik

1. PRIM_JARNIK(G = (V,E))
2. V′ ← {v} où v est un sommet choisi arbitrairement dans V
3. E′ ← {}
3. tant que V′ ≠ V
4. e = (s1,s2) ← e = (s1,s2) ∈ E tel que s1 ∈ V′, s2 ∉ V′ et cout(e) est minimal
5. ajouter s2 à V′
6. ajouter e à E′
7. retourner G′ = (V′,E′)

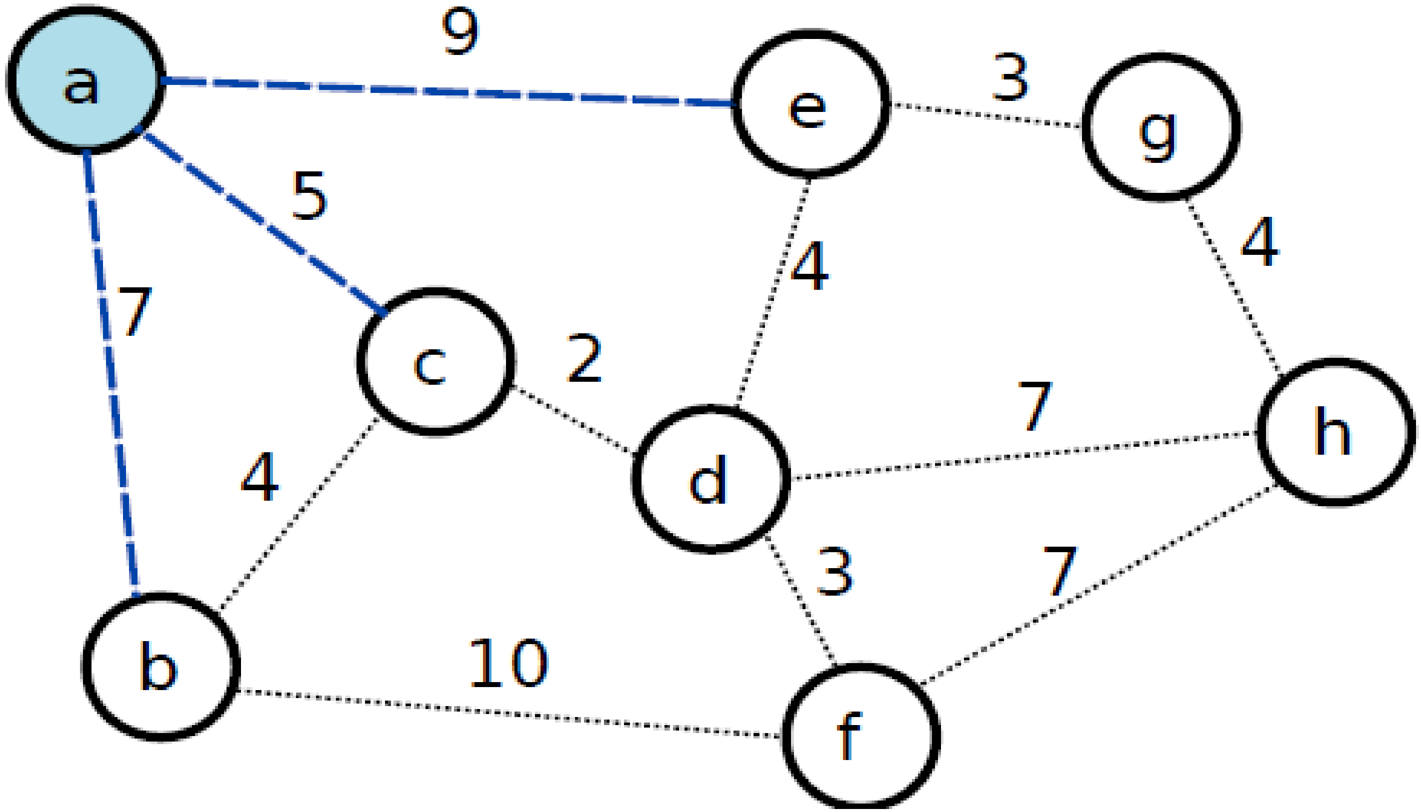
ARM, algorithme de Prim-Jarnik

Étape 0



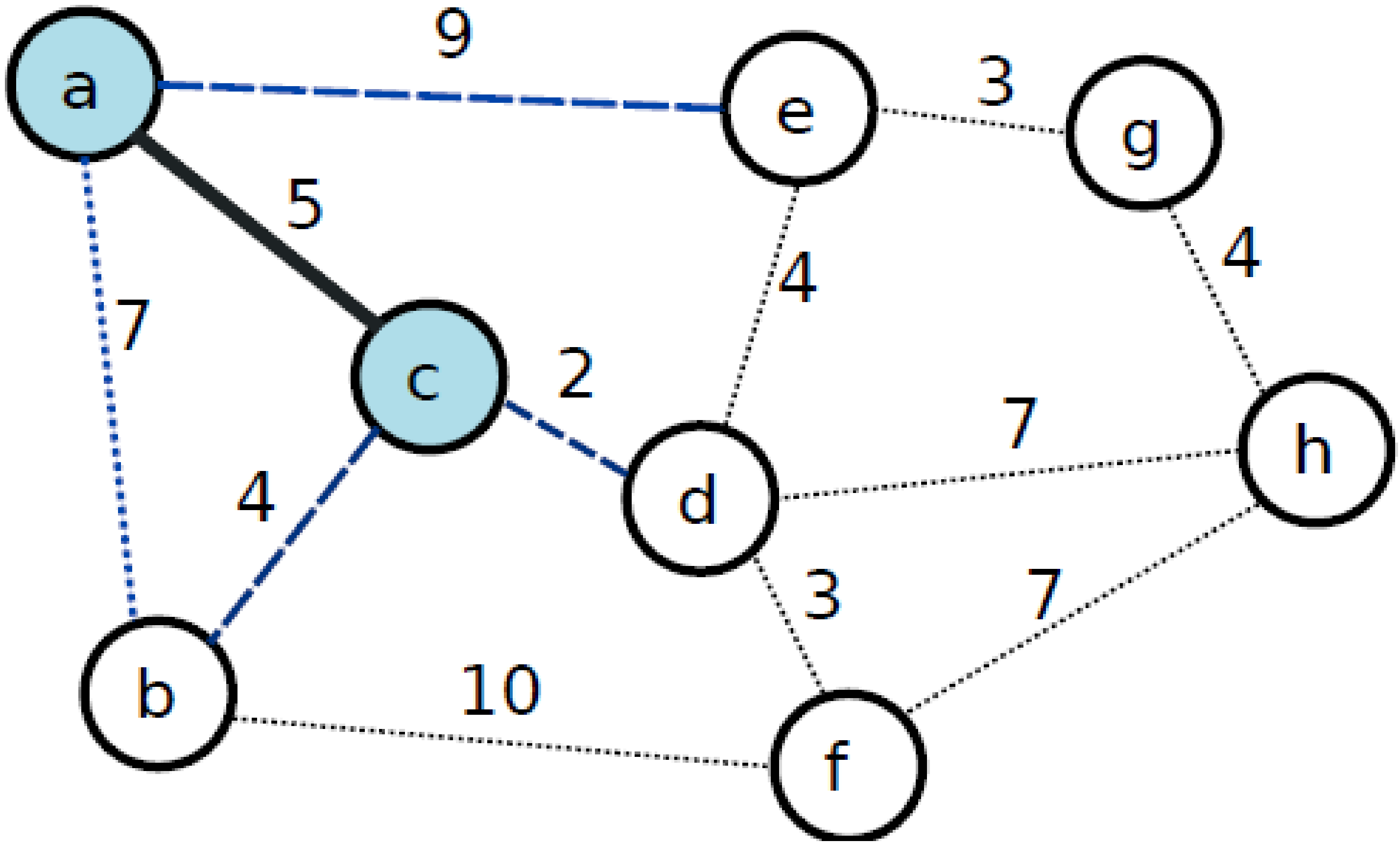
ARM, algorithme de Prim-Jarnik

Étape 1



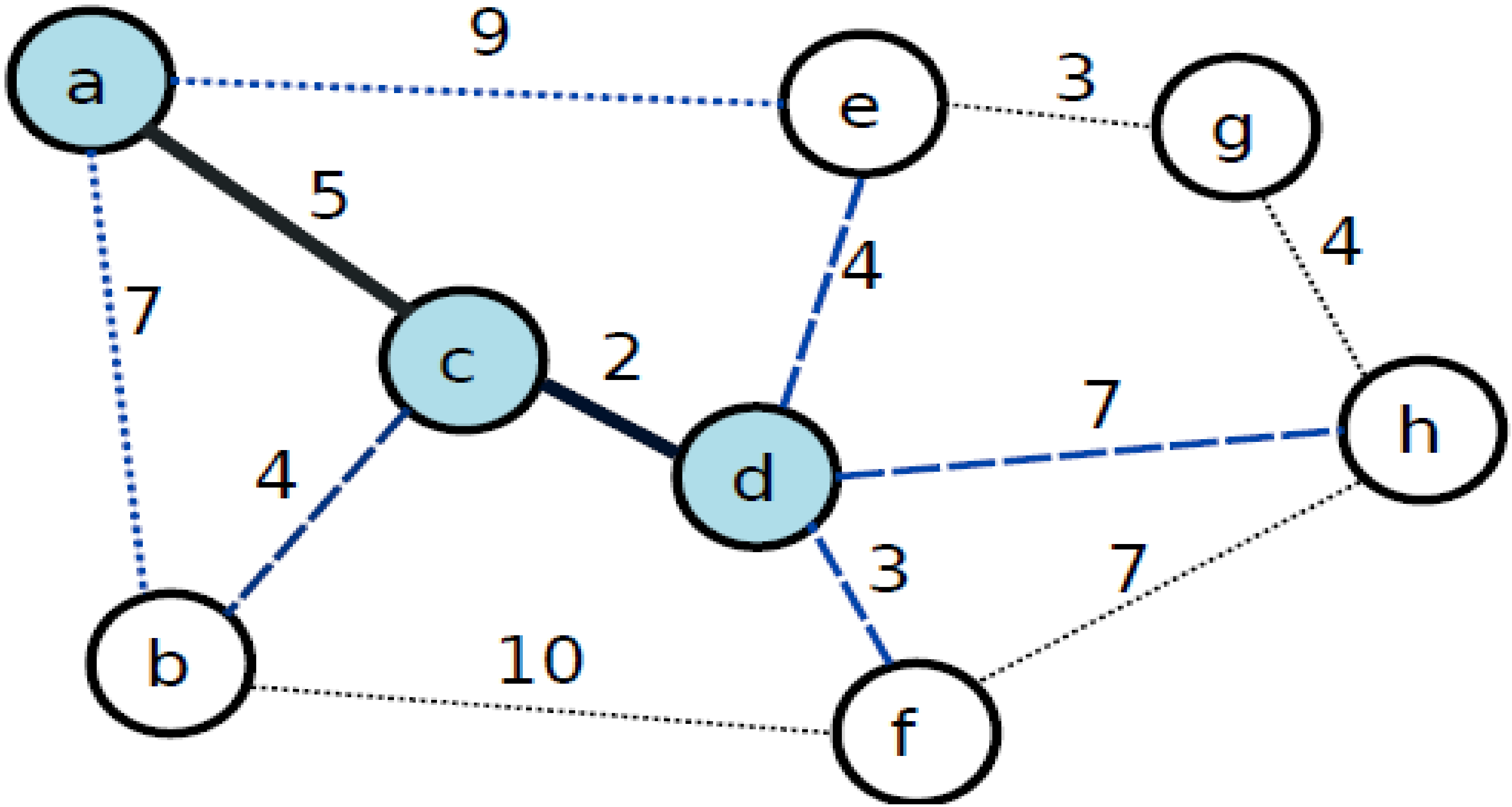
ARM, algorithme de Prim-Jarnik

Étape 2



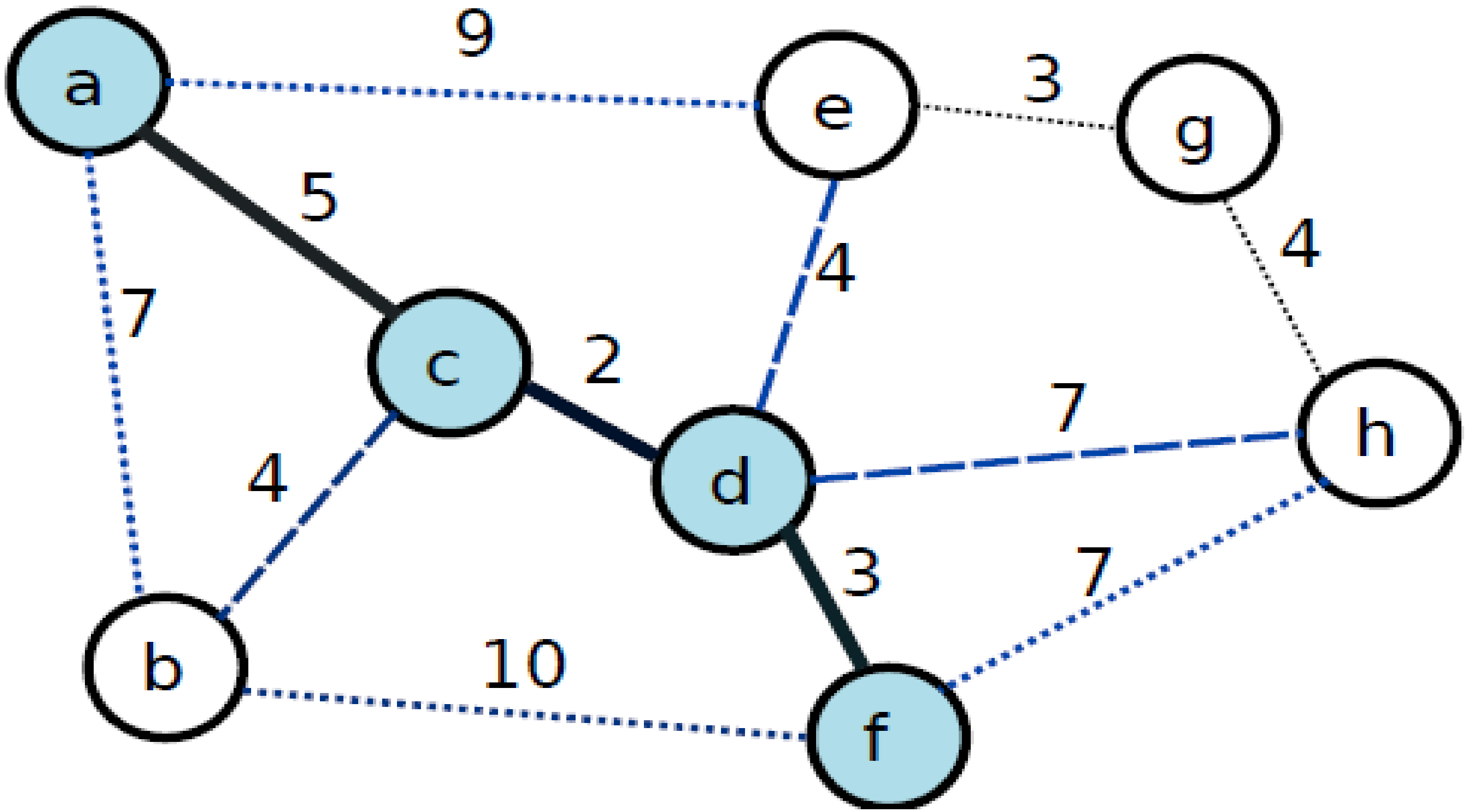
ARM, algorithme de Prim-Jarnik

Étape 3



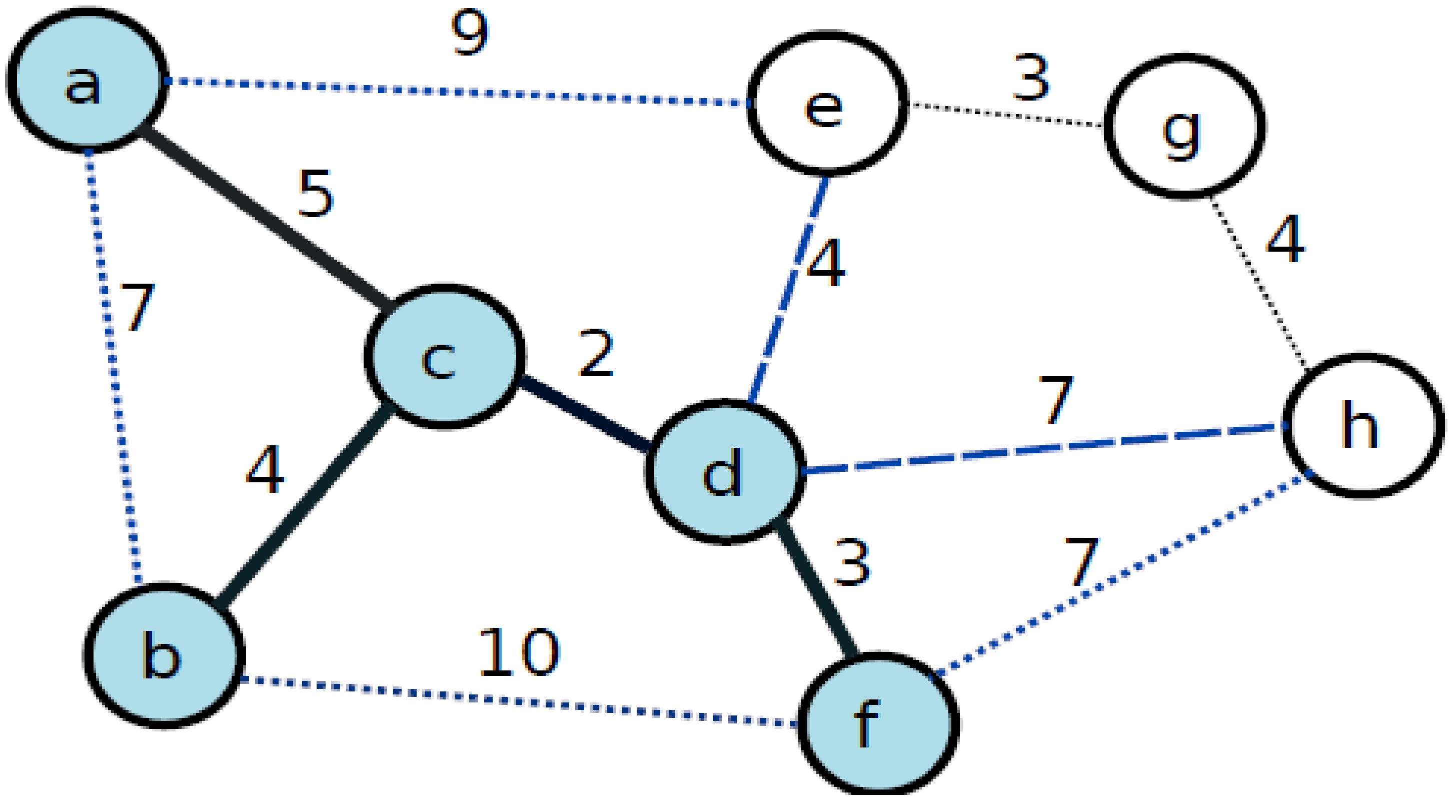
ARM, algorithme de Prim-Jarnik

Étape 4



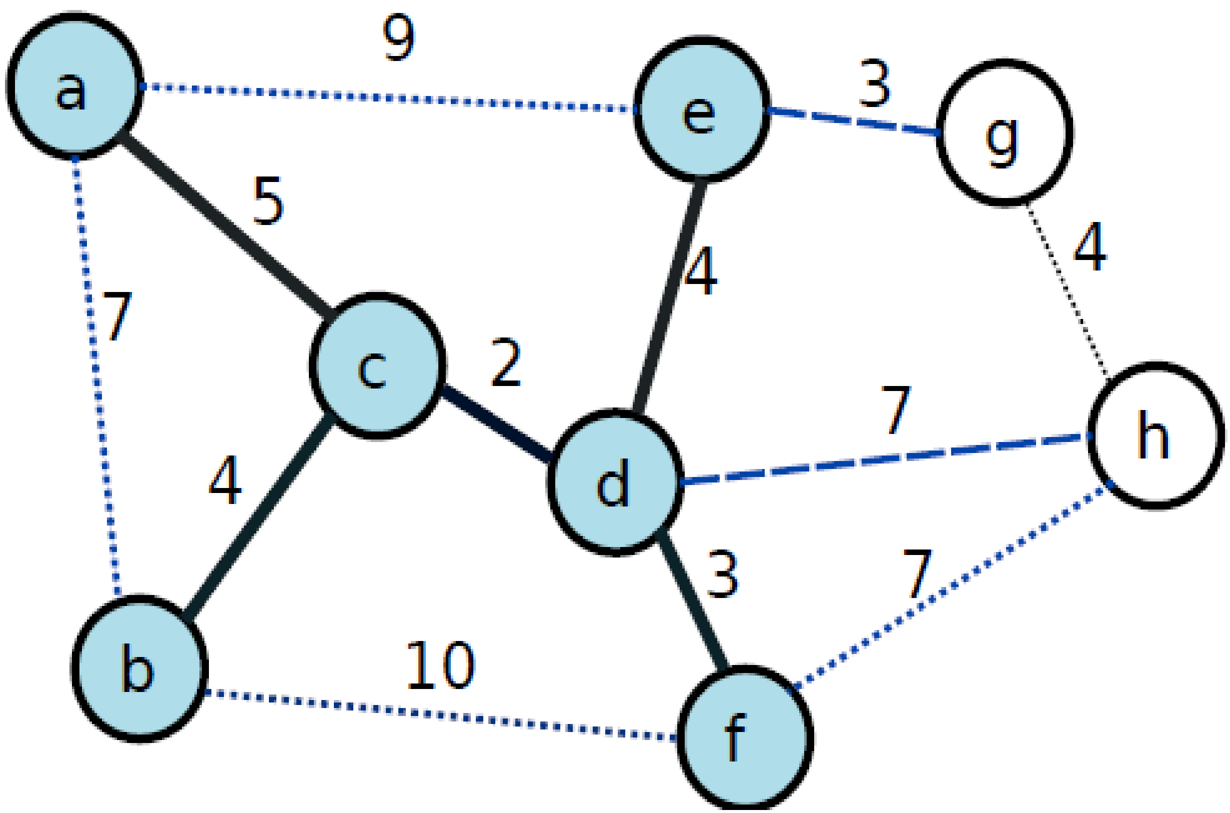
ARM, algorithme de Prim-Jarnik

Étape 5



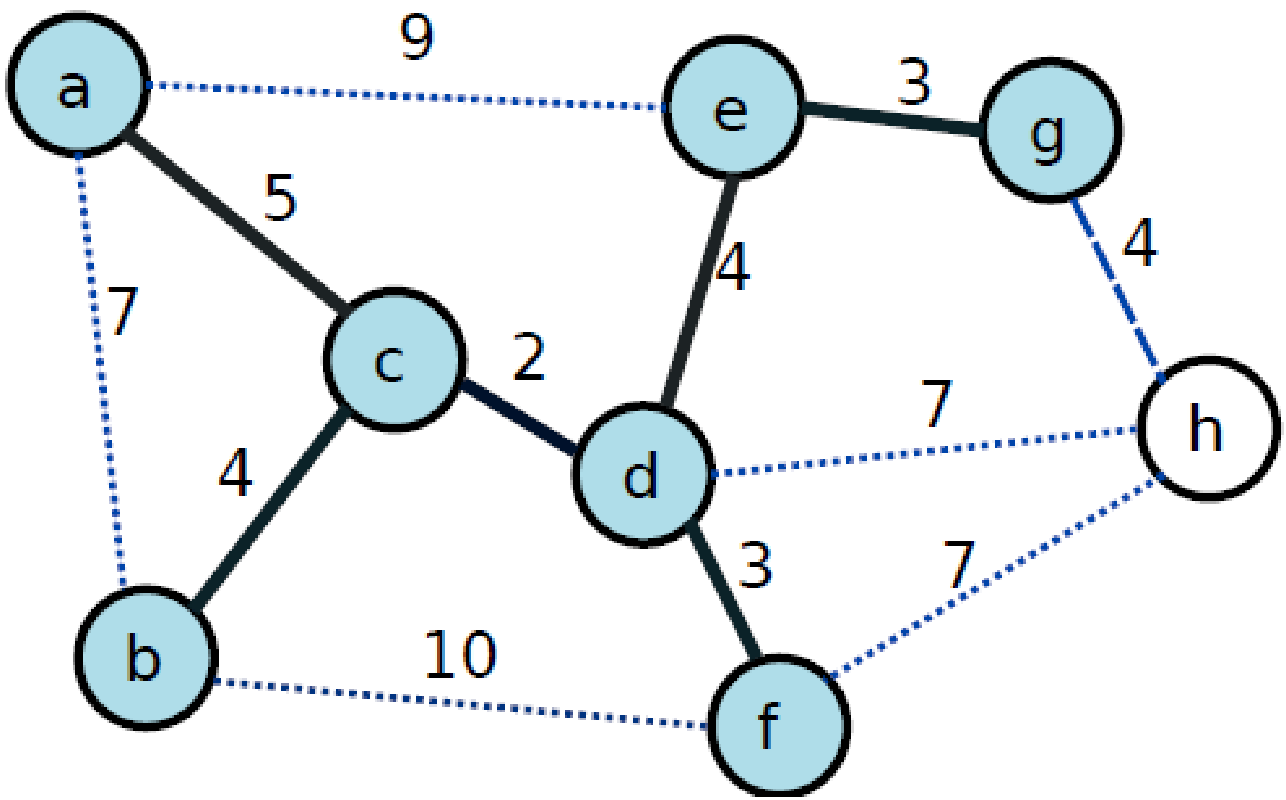
ARM, algorithme de Prim-Jarnik

Étape 6



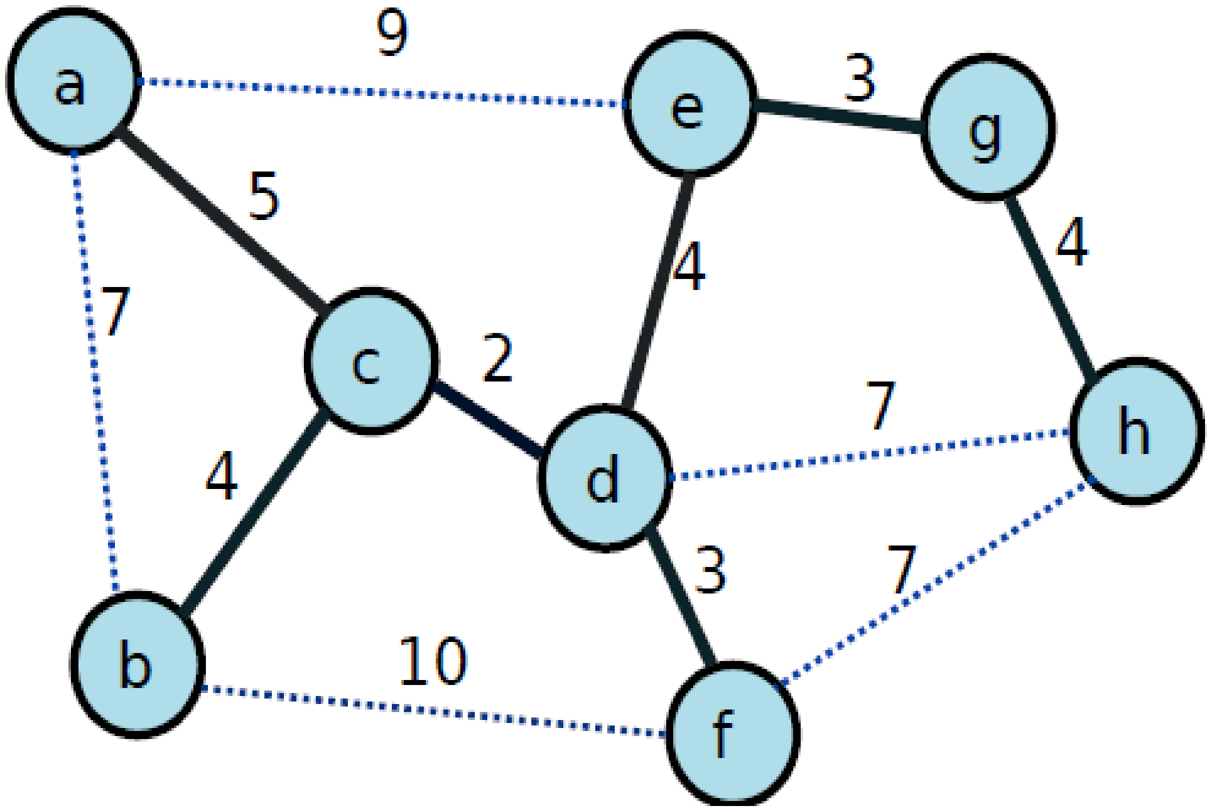
ARM, algorithme de Prim-Jarnik

Étape 7



ARM, algorithme de Prim-Jarnik

Étape 8



ARM, algorithme de Prim-Jarnik

Comment implémenter efficacement l’algorithme
de Prim-Jarnik ?

- Algo nécessite 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)

ARM, algorithme de Prim-Jarnik

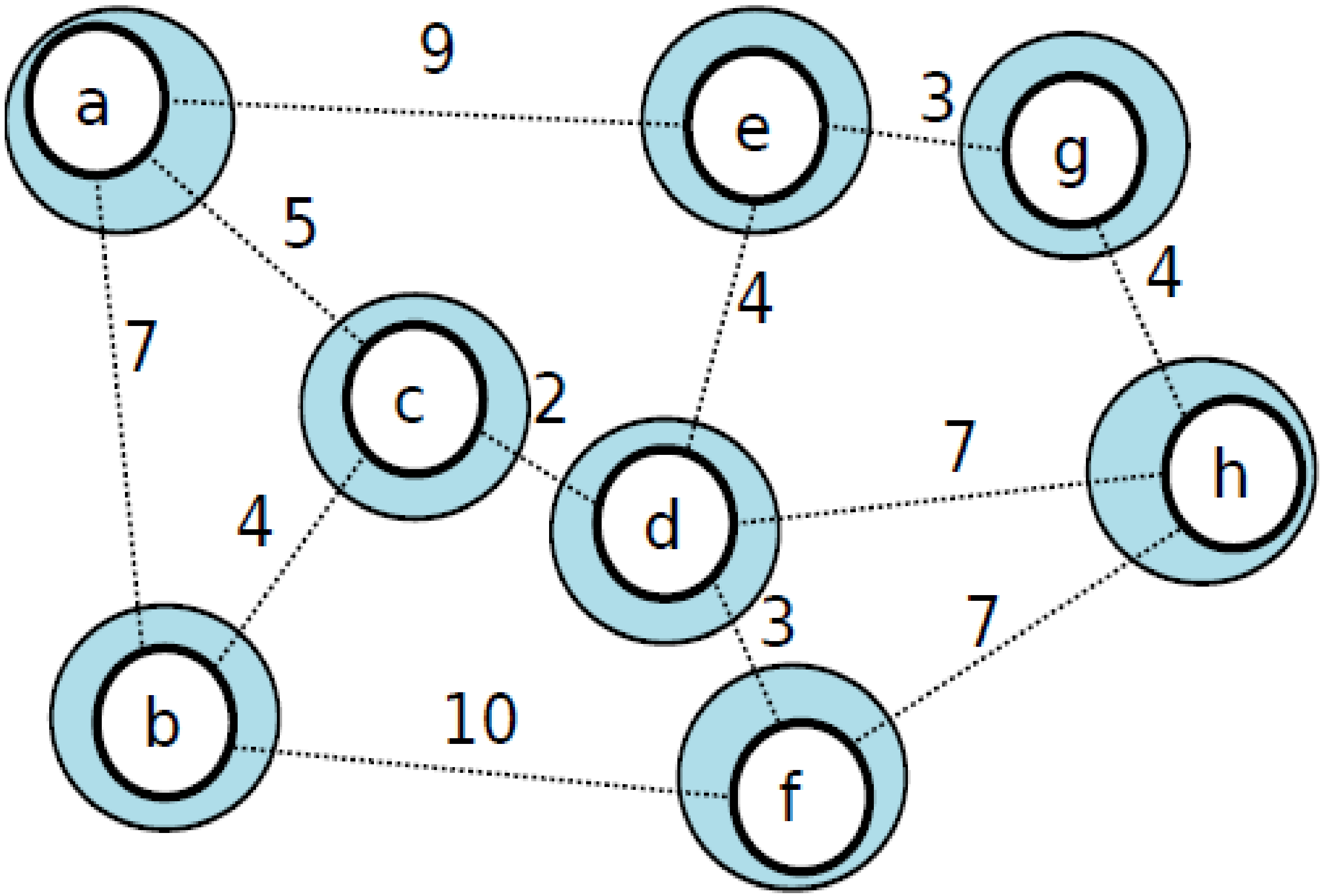
```
PRIM_JARNIK(G = (V, E))
   D ← créer un tableau de réels de dimension |V|
   pour tout i ∈ {0, . . . , |V| − 1}
      D[i] ← +∞
   v ← choisir arbitrairement un sommet v ∈ V
   D[v] ← 0
   Q ← créer une FilePrioritaire<Sommet v, Arête e> où la priorité de chaque élément est D[v].
   pour tout v ∈ V
      ajouter (v, nul) avec priorité D[v] dans V
   E′ ← {}
   tant que Q n'est pas vide
      (v, e) ← Q.enleverMinimum()
      ajouter l'arête e à E′
      pour tout arête sortante a = (v, w) à partir du sommet v, tel que le sommet w ∈ Q
         si cout(a) < D[w] alors
            D[w] ← cout(a)
            mettre à jour l'élément (w, a) pour le sommet w dans Q
            mettre à jour la clé du sommet w dans Q à D[w]
   retourner G′ = (V, E′)
```

ARM, algorithme de Kruskal

```
1. KRUSKAL(G = (V,E))
2.      pour tout sommet v ∈ V
3.           C(v) ← créer un ensemble {v}
4.      Q ← créer FilePrioritaire<Arête>(E) où la clé est le poids
5.      tant que |E′| < |V|-1
6.           e = (v1,v2) Q ← enleverMinimum()
7.           si C(v1) ≠ C(v2)
8.                ajouter e à E′
9.                fusionner C(v1) et C(v2) afin que C(v1) = C(v2)
10.     retourner G′ = (V,E′)
```

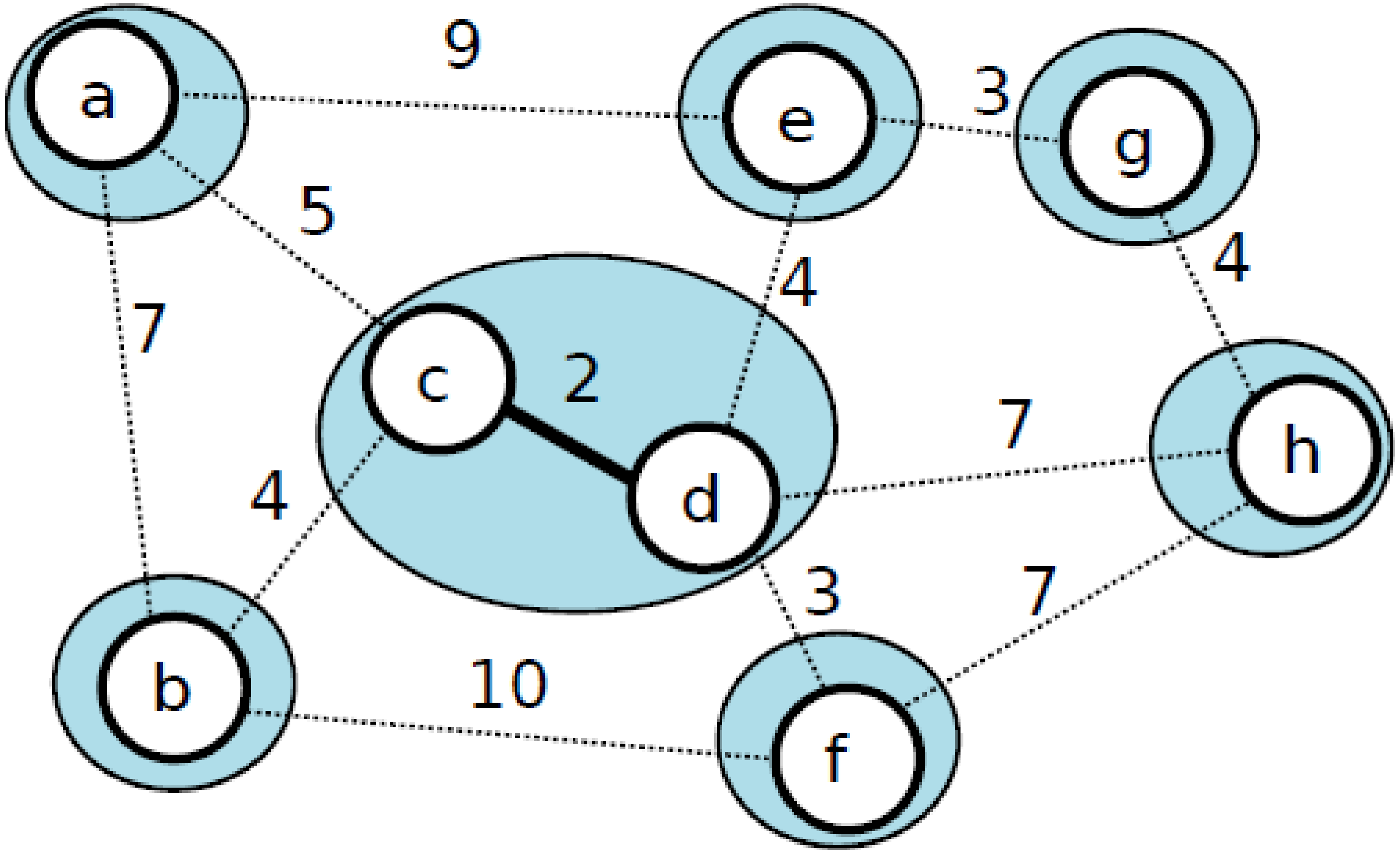
ARM, algorithme de Kruskal

Étape 0



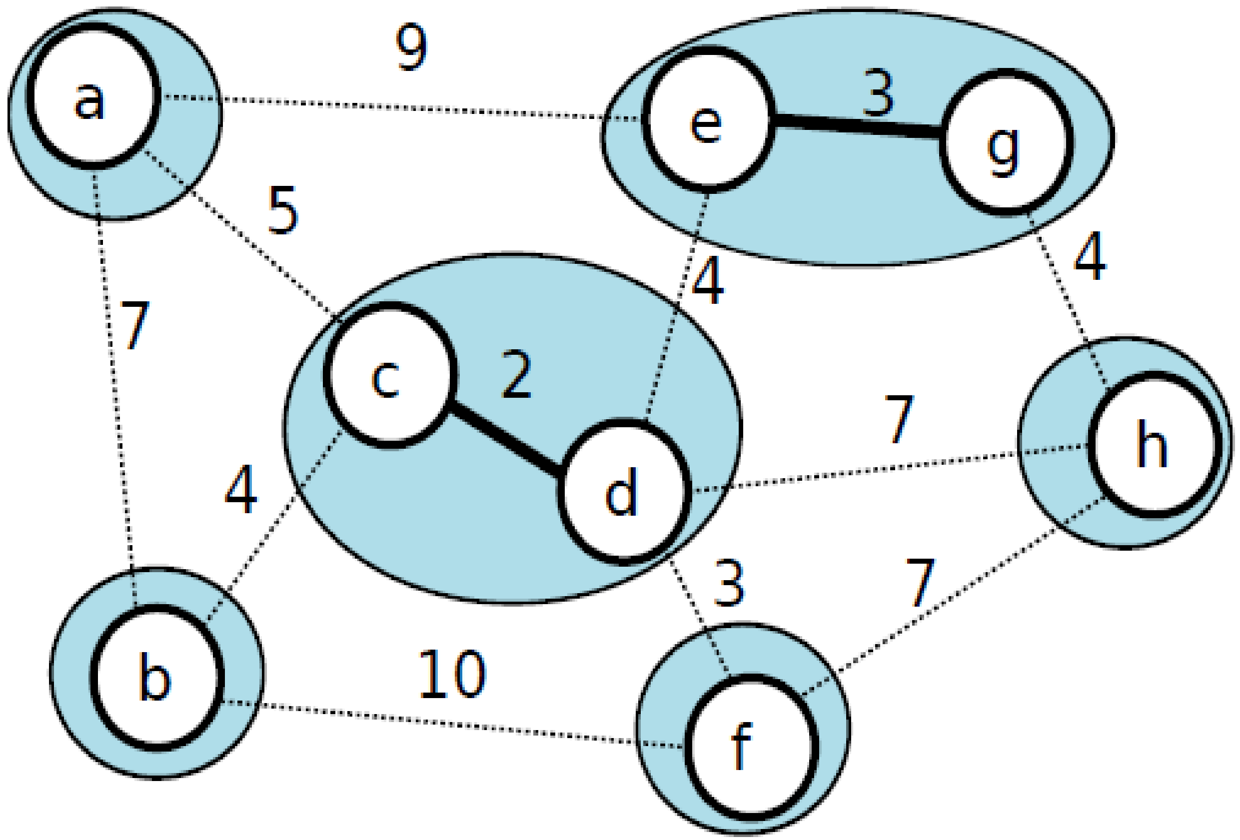
ARM, algorithme de Kruskal

Étape 1



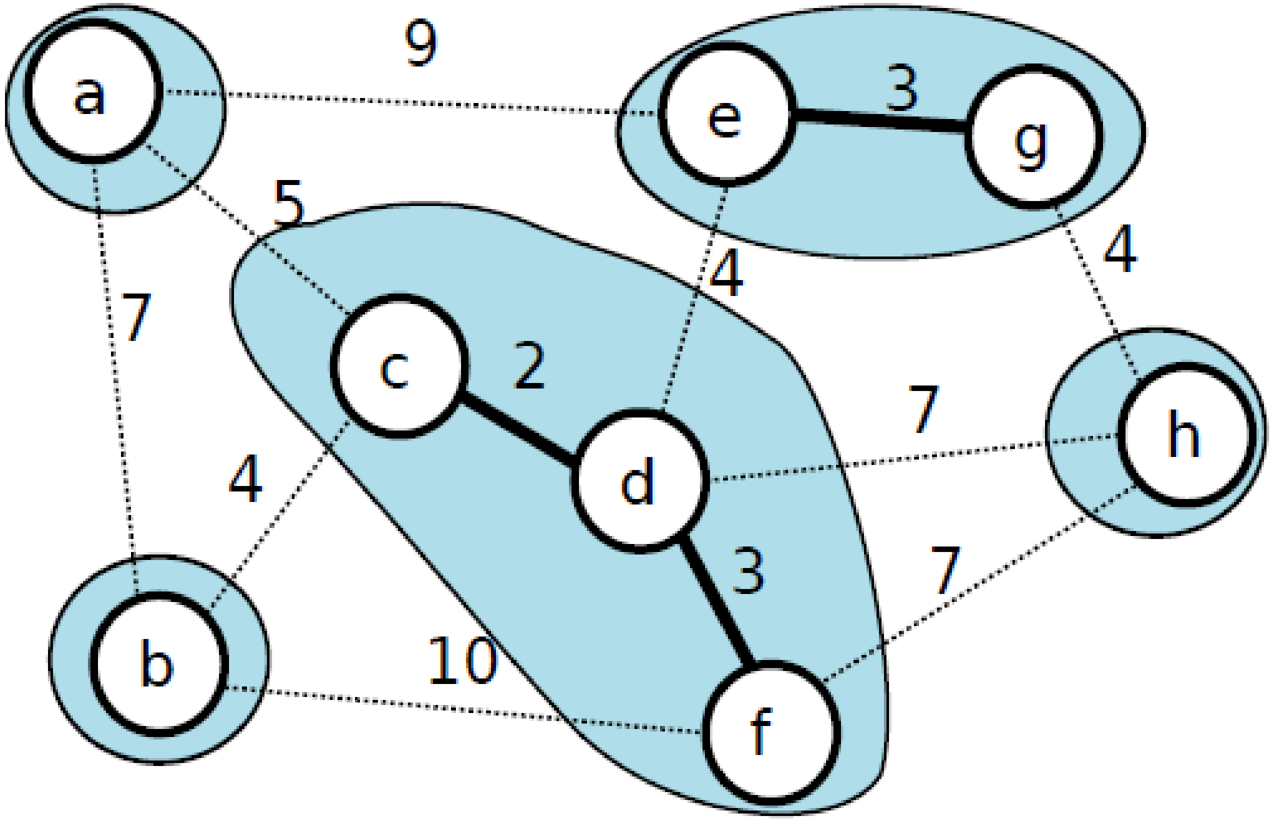
ARM, algorithme de Kruskal

Étape 2



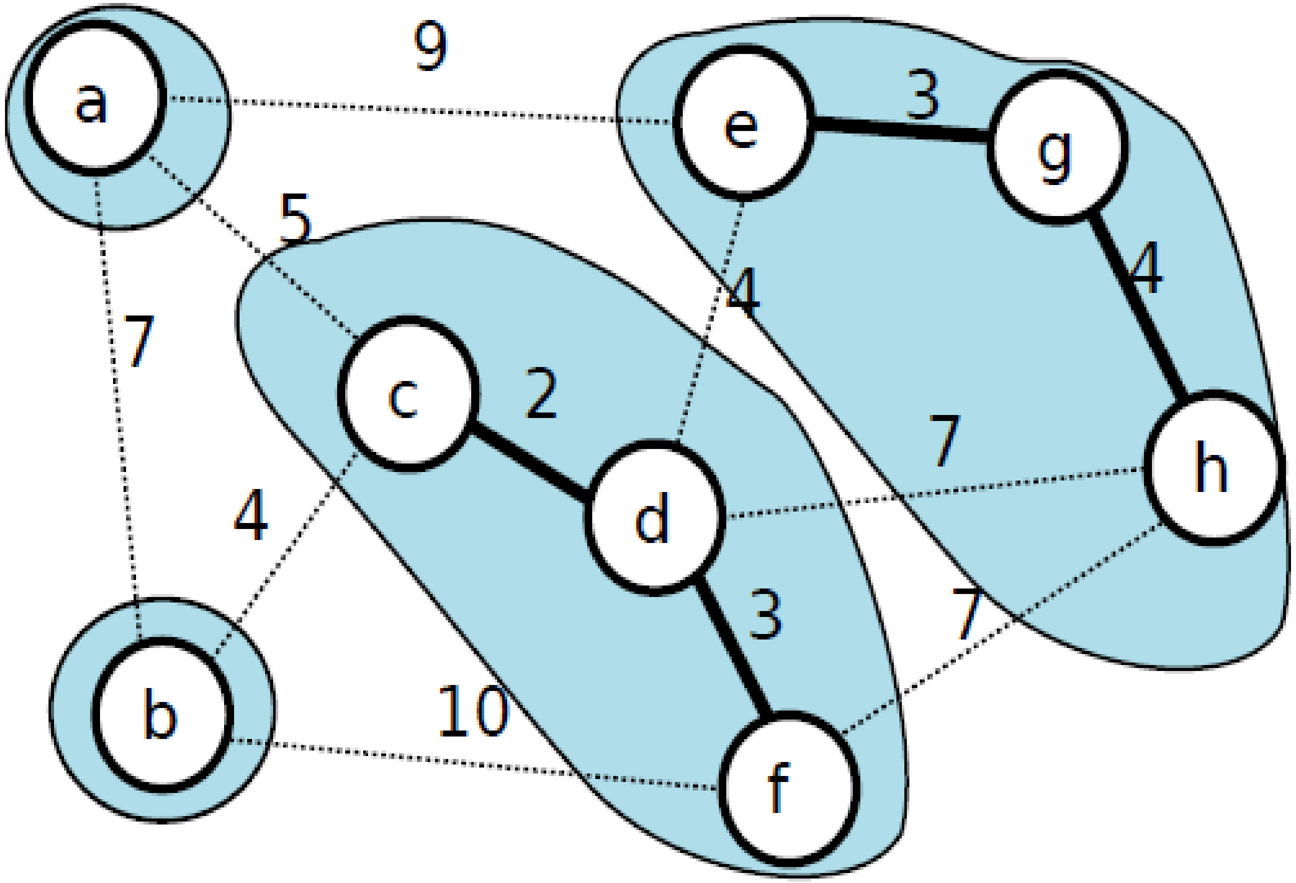
ARM, algorithme de Kruskal

Étape 3



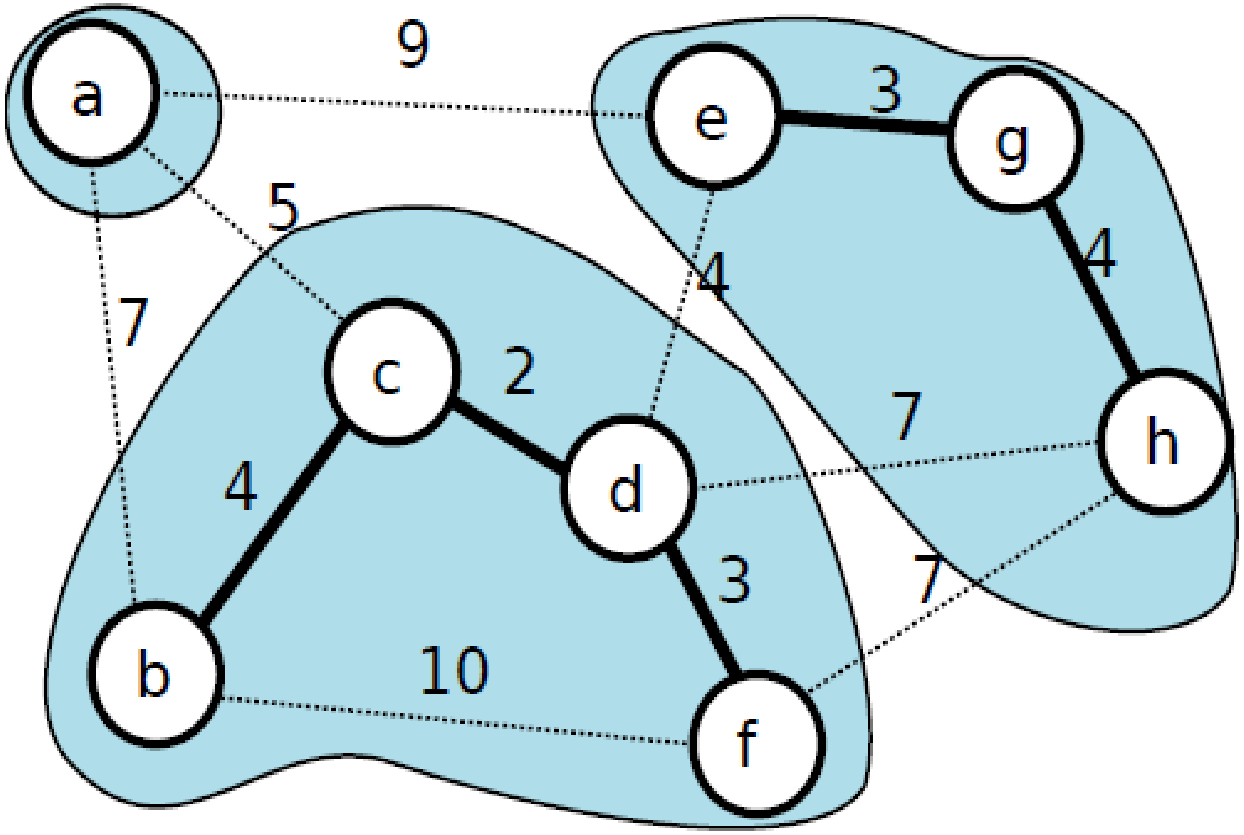
ARM, algorithme de Kruskal

Étape 4



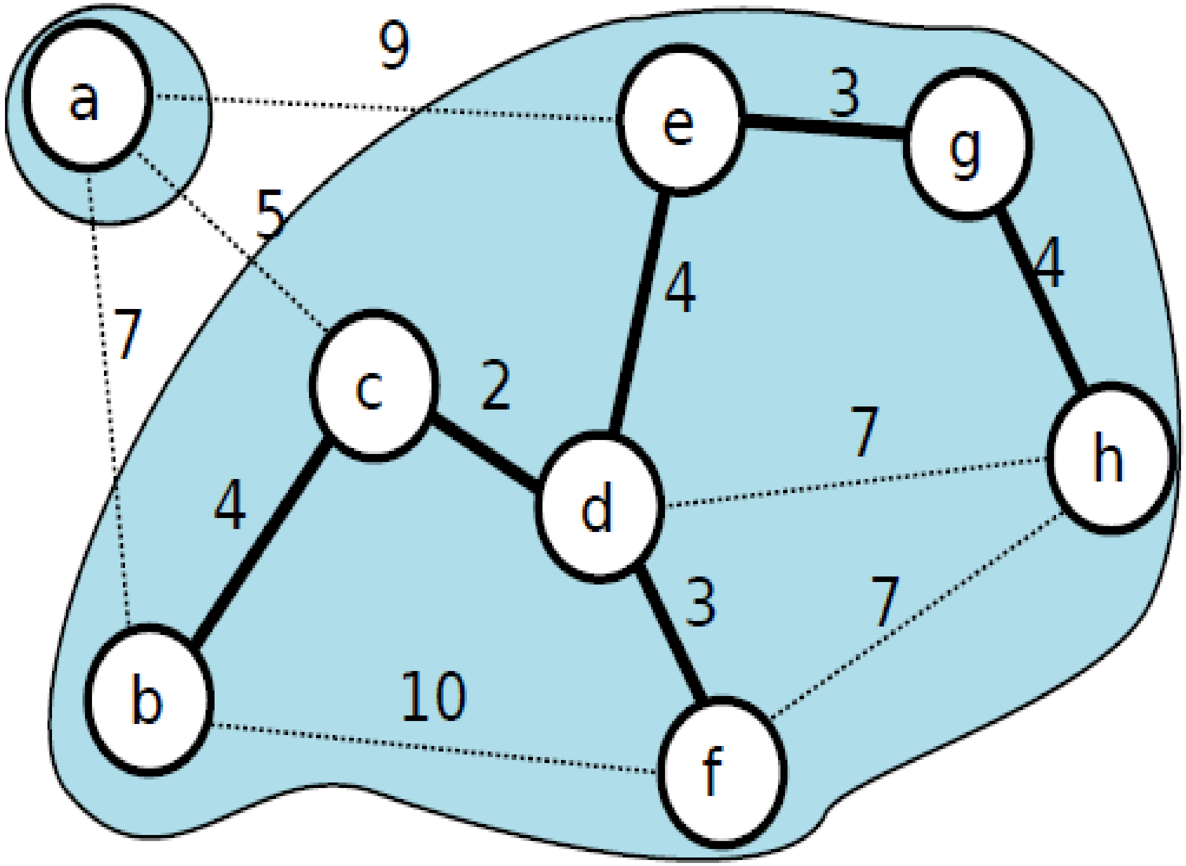
ARM, algorithme de Kruskal

Étape 5



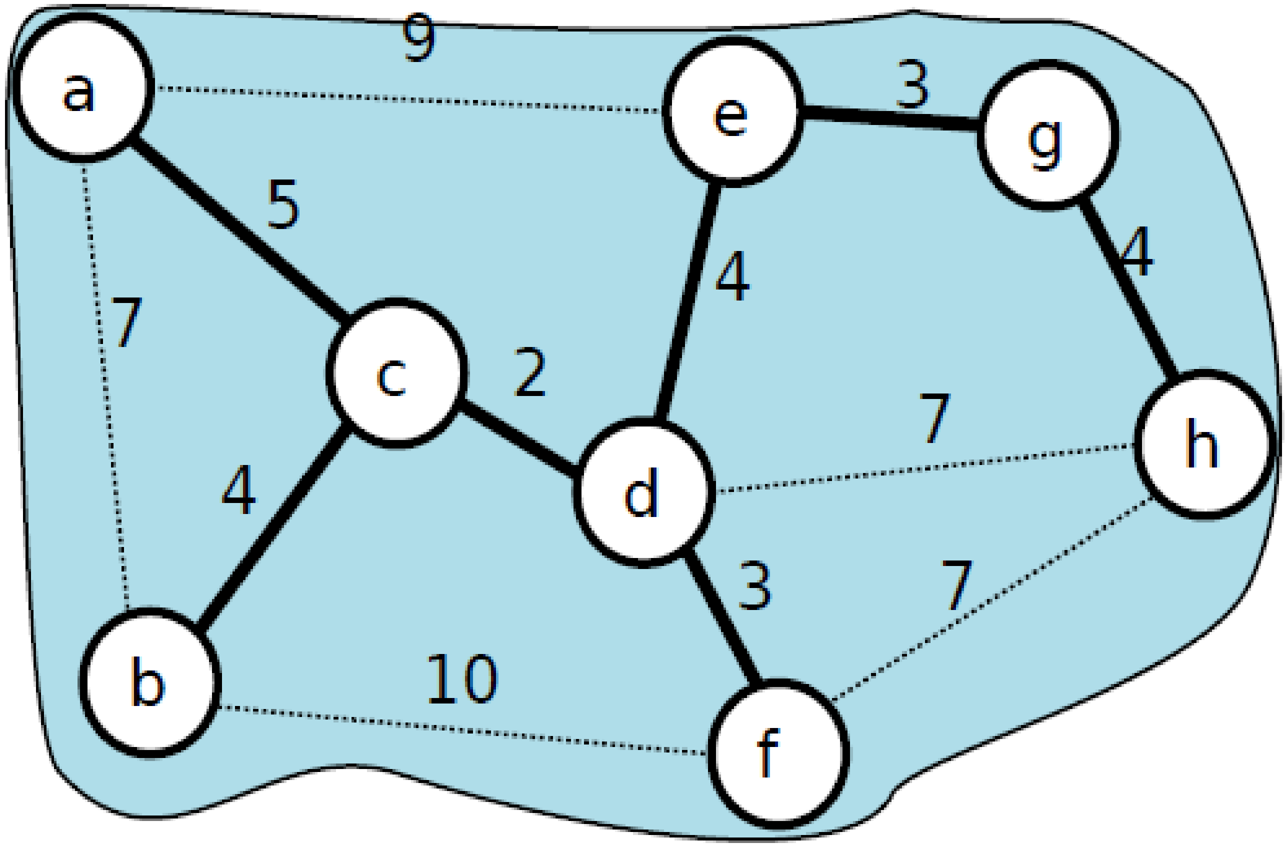
ARM, algorithme de Kruskal

Étape 6



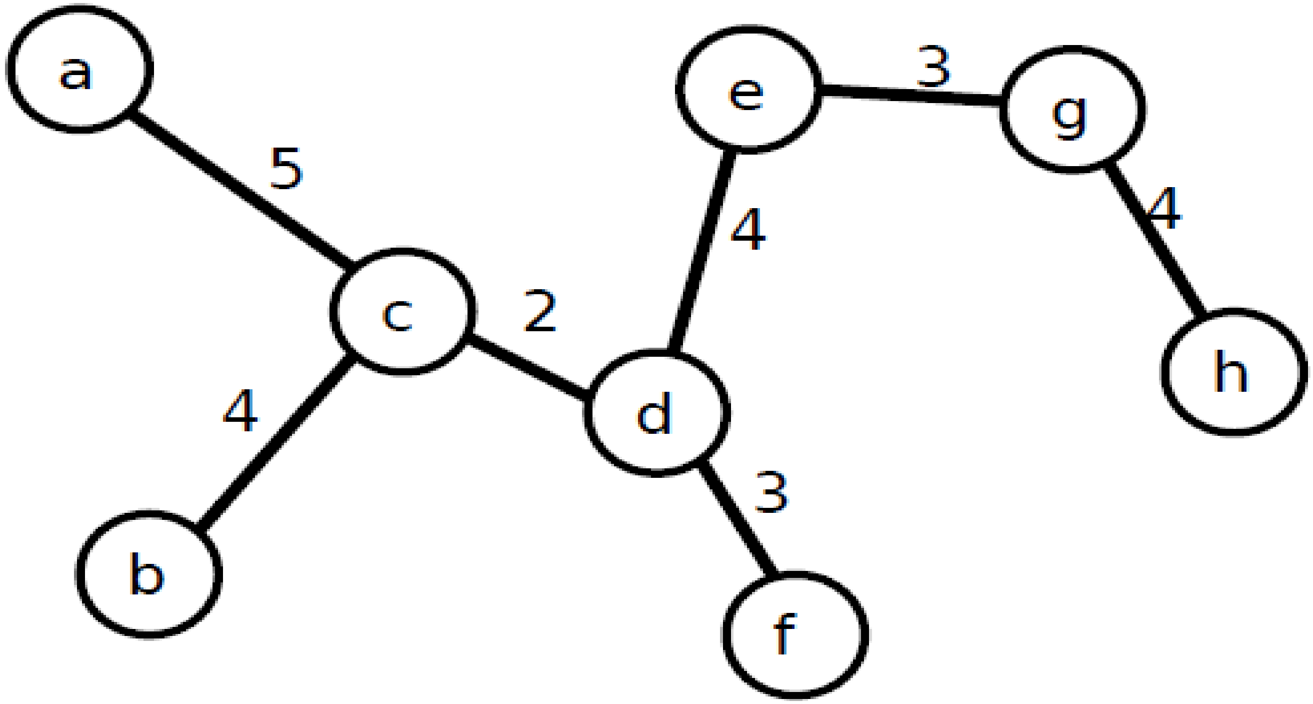
ARM, algorithme de Kruskal

Étape 7

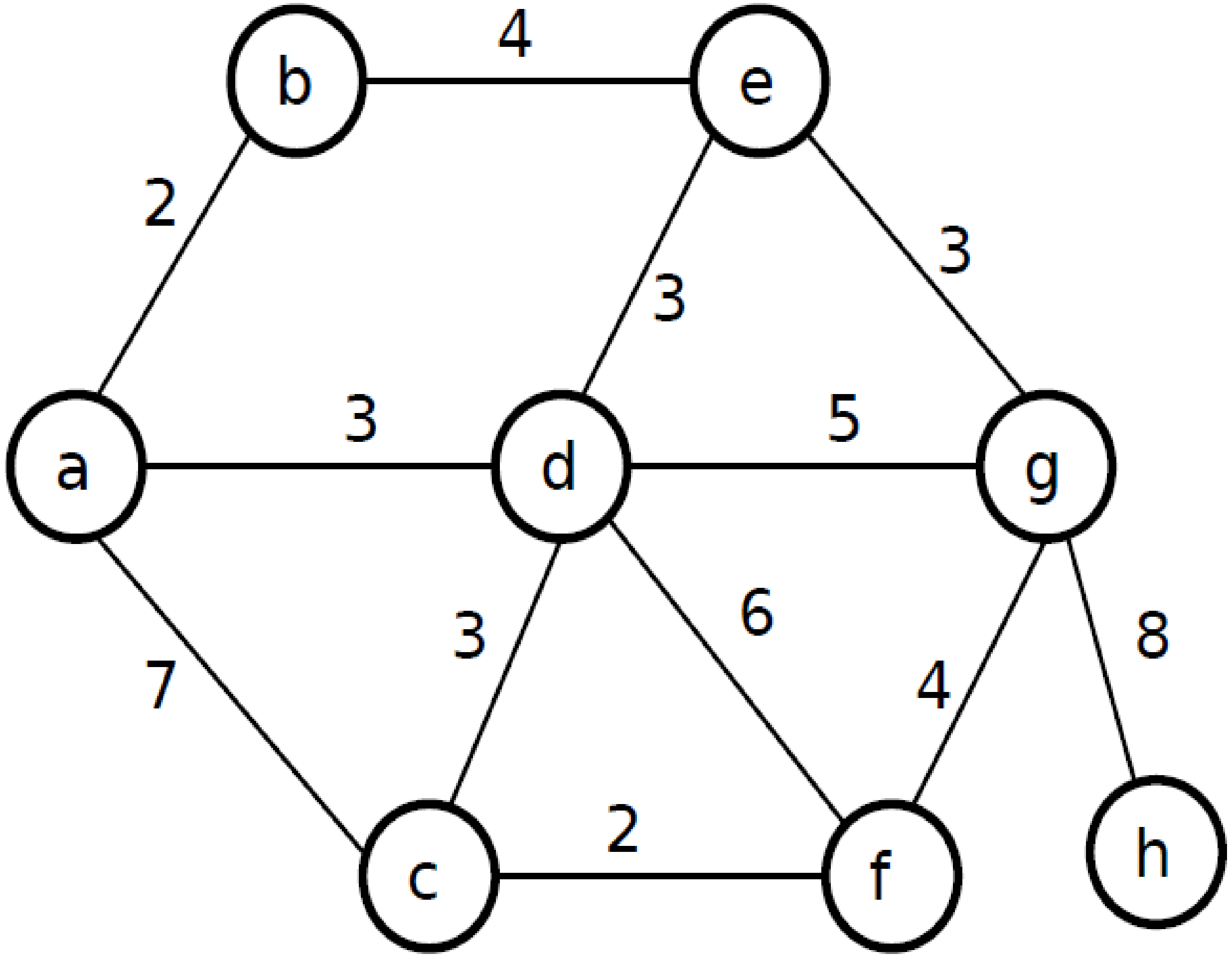


ARM, algorithme de Kruskal

Étape 8



ARM, Exercices



ARM, Complexités de Prim et Kruskal

- S’exécutent chacun en utilisant des tas binaires ordinaires (Monceau)
    - **$O(|E| \log |V|)$**
- En utilisant des tas de Fibonacci, l’algorithme de Prim peut être accéléré pour atteindre un temps d’exécution en
    - **$O(|E| + |V| \log |V|)$**
    - Amélioration quand $|V|$ est très inférieur à $|E|$

ARM, Complexités de Prim et Kruskal

- Les deux algorithmes sont des algorithmes gloutons
- À chaque étape de l’algorithme, une option parmi plusieurs possibles doit être choisie
- La stratégie gloutonne effectue le choix qui semble le meilleur à l’instant donné
- Une telle stratégie n’aboutit pas forcément à des solutions globalement optimales
- Pour le problème de l’ARM stratégies gloutonnes génèrent un arbre couvrant de poids minimum