Algorithmique des graphes

3 — Graphes pondérés

Anthony Labarre

10 février 2021

Université Gustave Eiffel

Cours précédent : caractérisation des graphes bipartis

Théorème 1

Un graphe est biparti si et seulement s'il ne contient pas de cycle de longueur impaire.

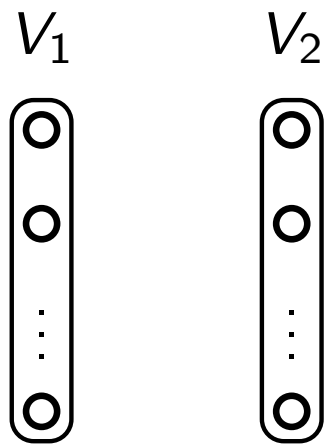
Cours précédent : caractérisation des graphes bipartis

Théorème 1

Un graphe est biparti si et seulement s'il ne contient pas de cycle de longueur impaire.

Démonstration.

$\Rightarrow$: tout cycle partant de v y revient par des aller-retours :



. . . et donc tout cycle de G est pair.

□

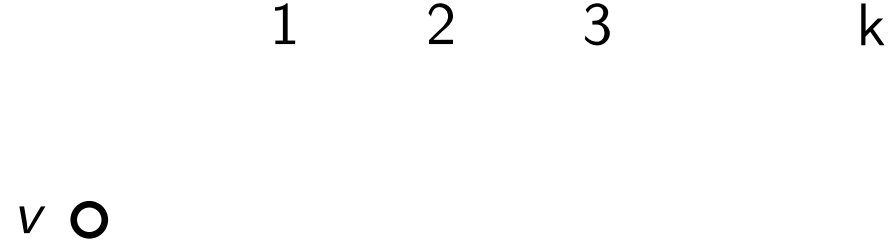
Cours précédent : caractérisation des graphes bipartis

Théorème 1

Un graphe est biparti si et seulement s'il ne contient pas de cycle de longueur impaire.

Démonstration.

$\Leftarrow$: $\equiv$ "si G n'est pas biparti, alors il contient un cycle impair".



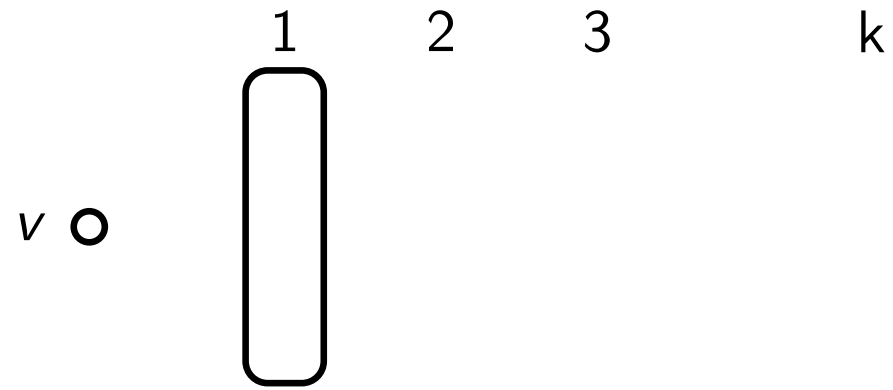
Cours précédent : caractérisation des graphes bipartis

Théorème 1

Un graphe est biparti si et seulement s'il ne contient pas de cycle de longueur impaire.

Démonstration.

$\Leftarrow$: $\equiv$ "si G n'est pas biparti, alors il contient un cycle impair".



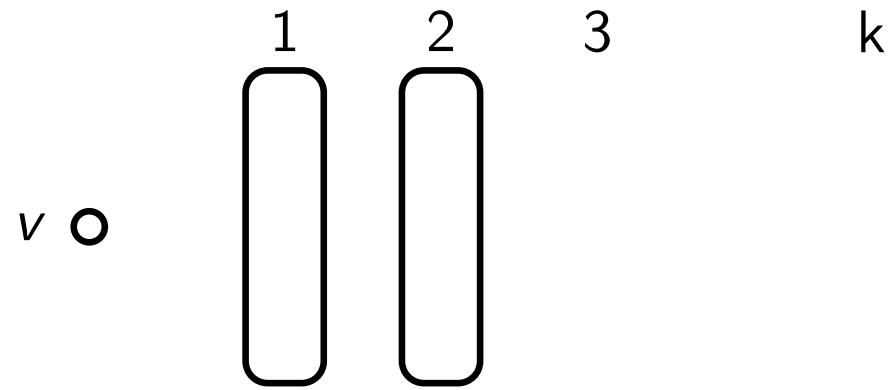
Cours précédent : caractérisation des graphes bipartis

Théorème 1

Un graphe est biparti si et seulement s'il ne contient pas de cycle de longueur impaire.

Démonstration.

$\Leftarrow$: $\equiv$ "si G n'est pas biparti, alors il contient un cycle impair".



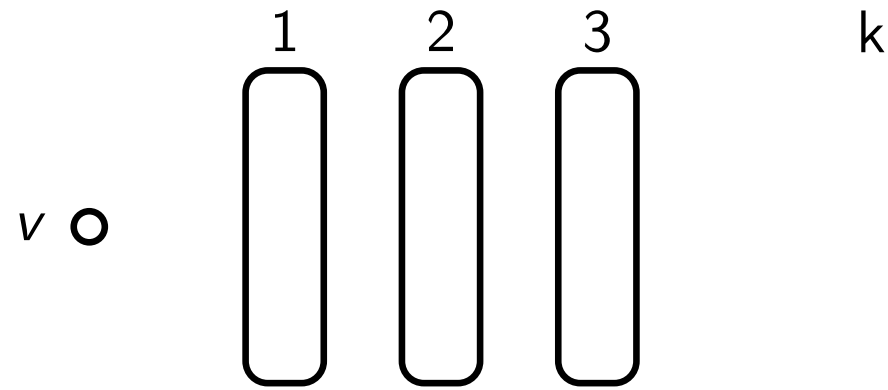
Cours précédent : caractérisation des graphes bipartis

Théorème 1

Un graphe est biparti si et seulement s'il ne contient pas de cycle de longueur impaire.

Démonstration.

$\Leftarrow$: $\equiv$ "si G n'est pas biparti, alors il contient un cycle impair".



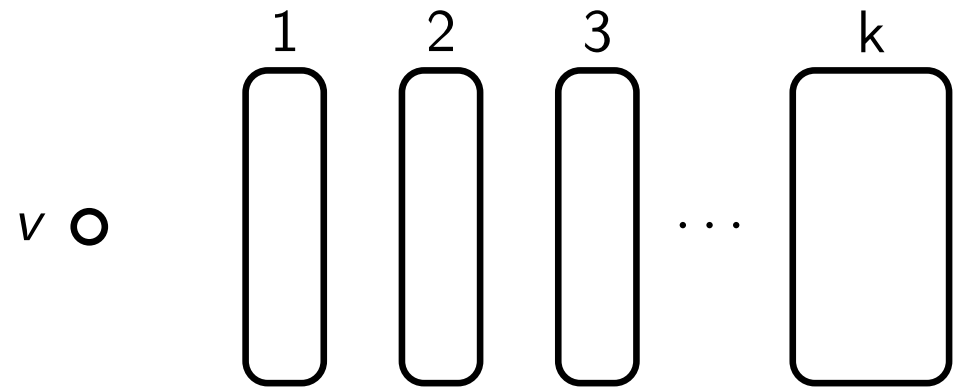
Cours précédent : caractérisation des graphes bipartis

Théorème 1

Un graphe est biparti si et seulement s'il ne contient pas de cycle de longueur impaire.

Démonstration.

$\Leftarrow$: $\equiv$ "si G n'est pas biparti, alors il contient un cycle impair".



□

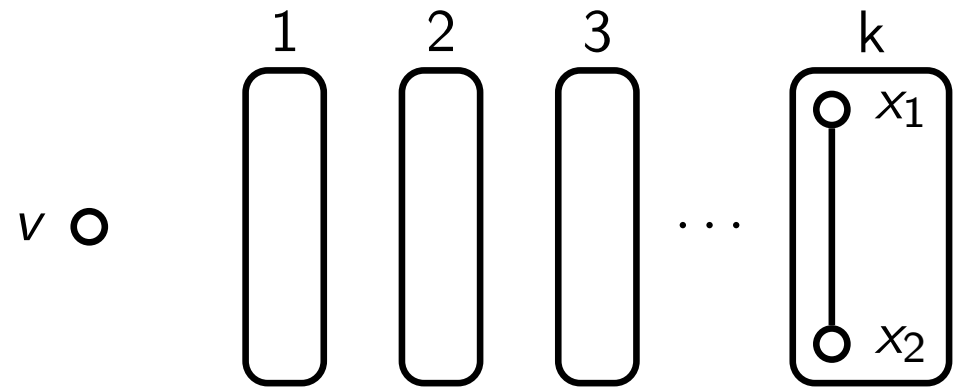
Cours précédent : caractérisation des graphes bipartis

Théorème 1

Un graphe est biparti si et seulement s'il ne contient pas de cycle de longueur impaire.

Démonstration.

$\Leftarrow$: $\equiv$ "si G n'est pas biparti, alors il contient un cycle impair".



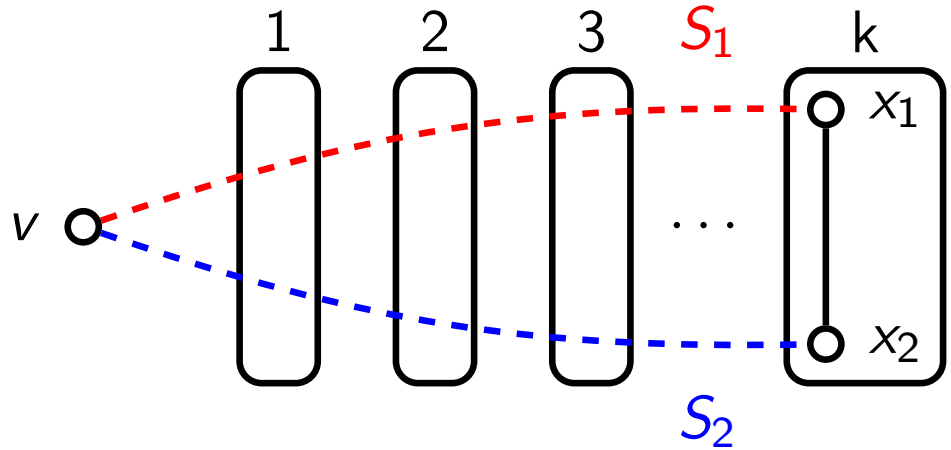
Cours précédent : caractérisation des graphes bipartis

Théorème 1

Un graphe est biparti si et seulement s'il ne contient pas de cycle de longueur impaire.

Démonstration.

$\Leftarrow$: $\equiv$ "si G n'est pas biparti, alors il contient un cycle impair".



□

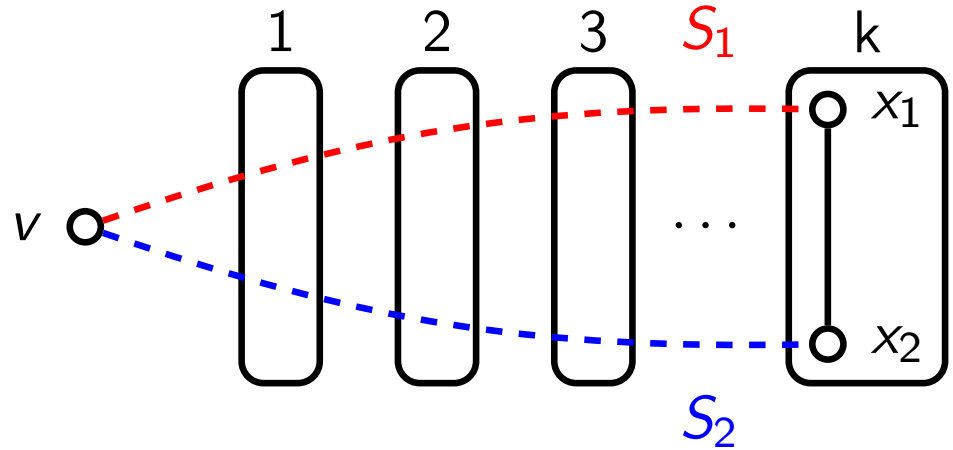
Cours précédent : caractérisation des graphes bipartis

Théorème 1

Un graphe est biparti si et seulement s'il ne contient pas de cycle de longueur impaire.

Démonstration.

$\Leftarrow$: $\equiv$ "si G n'est pas biparti, alors il contient un cycle impair".



$S_1 + \{x_1, x_2\} + S_2$ est un cycle de longueur impaire.

□

Graphes pondérés

Définition 2

Un **graphe pondéré** est un graphe $G = (V, E, w)$, où $w : E \to \mathbb{R} : \{u, v\} \mapsto w(\{u, v\})$ est une fonction affectant à chaque arête un poids réel.

Graphes pondérés

Définition 2

Un **graphe pondéré** est un graphe $G = (V, E, w)$, où $w : E \to \mathbb{R} : \{u, v\} \mapsto w(\{u, v\})$ est une fonction affectant à chaque arête un poids réel.

Définition 3

Le **poids** d'un (sous-)graphe $G = (V, E, w)$ est la quantité $w(G) = \sum_{e \in E} w(e)$.

Graphes pondérés

Définition 2

Un **graphe pondéré** est un graphe $G = (V, E, w)$, où $w : E \to \mathbb{R} : \{u, v\} \mapsto w(\{u, v\})$ est une fonction affectant à chaque arête un poids réel.

Définition 3

Le **poids** d'un (sous-)graphe $G = (V, E, w)$ est la quantité $w(G) = \sum_{e \in E} w(e)$.

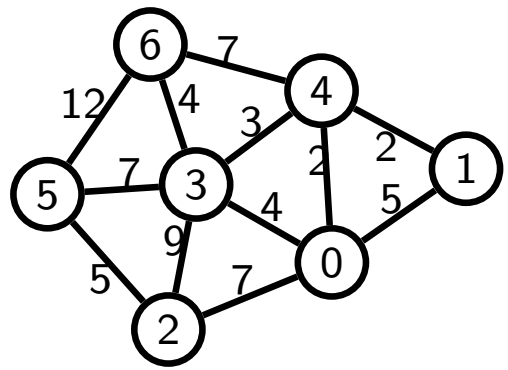
Tous les graphes pondérés vus aujourd'hui seront connexes pour simplifier la discussion, mais nos algorithmes sont facilement adaptables aux graphes non connexes.

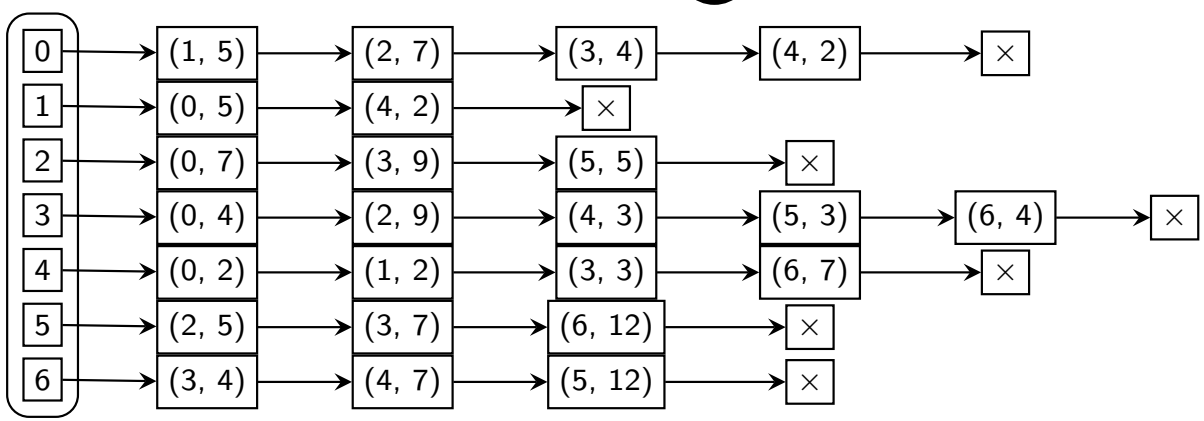
Implémentation des graphes pondérés

- On peut facilement modifier ce qu'on a déjà vu pour implémenter les graphes pondérés :
  - matrice d'adjacence : poids au lieu de booléens ;
  - listes d'adjacence : couples (voisin, poids) au lieu de voisins ;

Exemple 1

	0	1	2	3	4	5	6
0		5	7	4	2		
1	5				2		
2	7			9		5	
3	4		9		3	7	4
4	2	2		3			7
5			5	7			12
6				4	7	12	





Implémentation des graphes pondérés

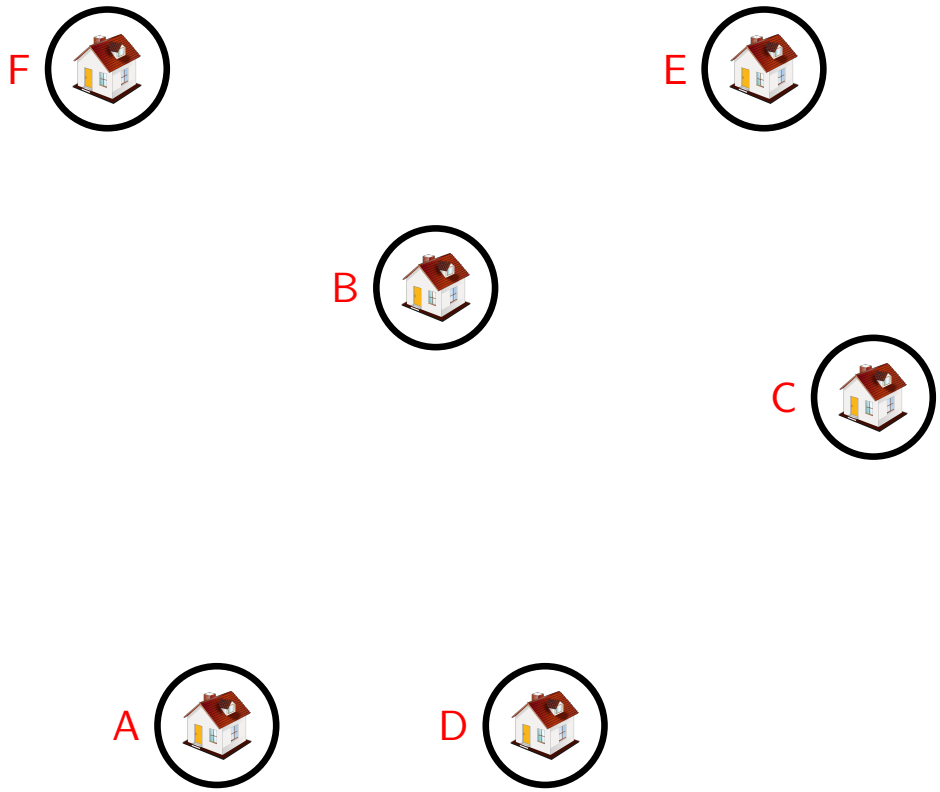
- On suppose l'existence d'une classe `GraphePondéré` très similaire à la classe `Graphe`, avec quelques modifications :
  - ajouter_arête(u, v, poids) ;
  - ajouter_arêtes(séquence) ;
  - arêtes() ;
  - boucles() ;
  - sous_graphe_induit(séquence) ;

Implémentation des graphes pondérés

- On suppose l'existence d'une classe `GraphePondéré` très similaire à la classe `Graphe`, avec quelques modifications :
  - ajouter_arête(u, v, poids) ;
  - ajouter_arêtes(séquence) ;
  - arêtes() ;
  - boucles() ;
  - sous_graphe_induit(séquence) ;
- . . . et quelques ajouts :
  - arêtes_incidentes(sommet) ;
  - poids_arête(u, v) ;

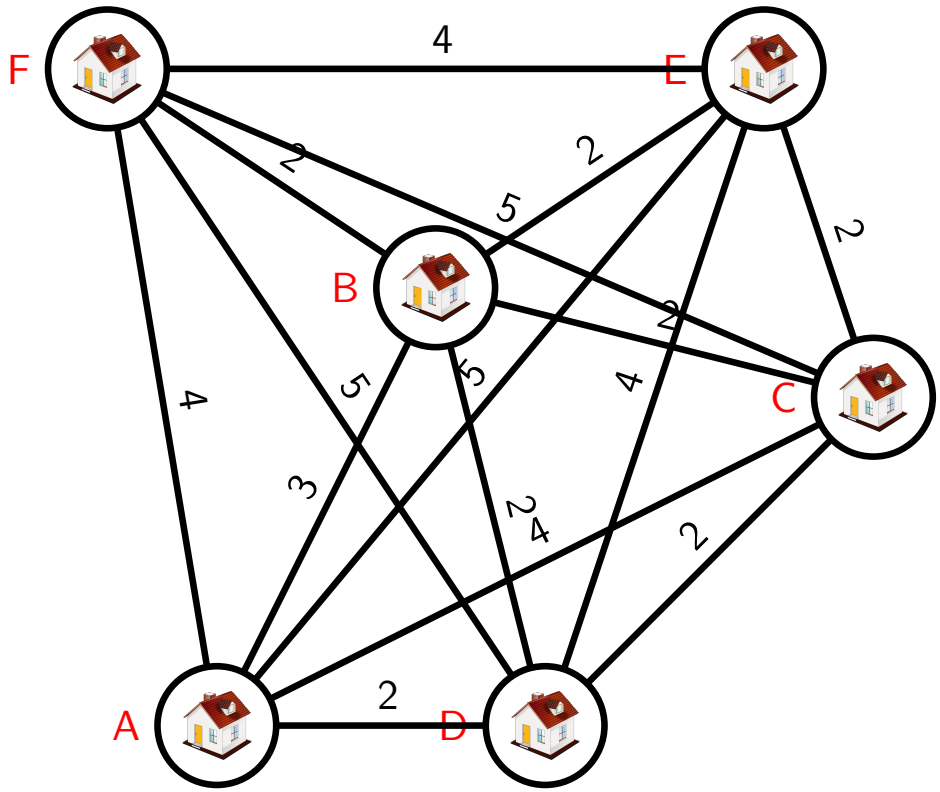
Motivations

On veut que chaque paire de maisons soit mutellement accessible.
Comment relier les maisons à moindre coût ?



Motivations

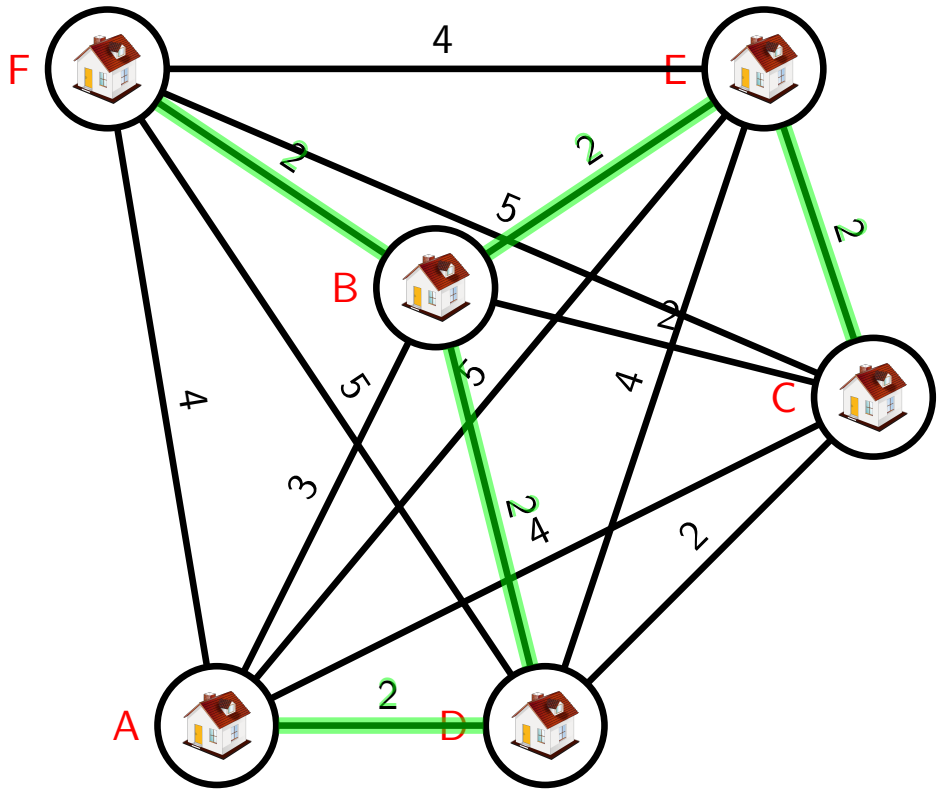
On veut que chaque paire de maisons soit mutellement accessible.
Comment relier les maisons à moindre coût ?



- On pourrait ajouter tous les liens possibles . . . mais c'est cher ;

Motivations

On veut que chaque paire de maisons soit mutellement accessible.
Comment relier les maisons à moindre coût ?



- On pourrait ajouter tous les liens possibles . . . mais c'est cher ;
- Ou ne garder qu'un sous-graphe connexe de poids minimum ;

Arbres couvrants de poids minimum

Définition 4

Un sous-graphe **couvrant** d'un graphe connexe donné $G = (V, E)$ est un graphe connexe de la forme $H = (V, F)$ où $F \subseteq E$.

Arbres couvrants de poids minimum

Définition 4

Un sous-graphe **couvrant** d'un graphe connexe donné $G = (V, E)$ est un graphe connexe de la forme $H = (V, F)$ où $F \subseteq E$.

- Les arbres (ou forêts) de parcours de la fois passée étaient couvrants ;

Arbres couvrants de poids minimum

Définition 4

Un sous-graphe **couvrant** d'un graphe connexe donné $G = (V, E)$ est un graphe connexe de la forme $H = (V, F)$ où $F \subseteq E$.

- Les arbres (ou forêts) de parcours de la fois passée étaient couvrants ;
- Maintenant qu'on a des poids à prendre en compte, on va chercher des arbres couvrants avec un autre objectif :

Arbres couvrants de poids minimum

Définition 4

Un sous-graphe **couvrant** d'un graphe connexe donné $G = (V, E)$ est un graphe connexe de la forme $H = (V, F)$ où $F \subseteq E$.

- Les arbres (ou forêts) de parcours de la fois passée étaient couvrants ;
- Maintenant qu'on a des poids à prendre en compte, on va chercher des arbres couvrants avec un autre objectif :

Définition 5

Un **arbre couvrant de poids minimum** (ou **ACPM**) pour un graphe pondéré G est un arbre couvrant T pour G tel que pour tout arbre couvrant T' pour G, on a $w(T) \leq w(T')$.

Algorithmes de calcul d'ACPM

- Les deux algorithmes que nous verrons rajoutent progressivement des arêtes à un sous-graphe T de G ;

Algorithmes de calcul d'ACPM

- Les deux algorithmes que nous verrons rajoutent progressivement des arêtes à un sous-graphe T de G ;
- On fera la distinction entre les catégories suivantes d'arêtes ; une arête e de G est :

Algorithmes de calcul d'ACPM

- Les deux algorithmes que nous verrons rajoutent progressivement des arêtes à un sous-graphe T de G ;
- On fera la distinction entre les catégories suivantes d'arêtes ; une arête e de G est :
  - **candidate** si elle a au moins une extrémité dans T ;

Algorithmes de calcul d'ACPM

- Les deux algorithmes que nous verrons rajoutent progressivement des arêtes à un sous-graphe T de G ;
- On fera la distinction entre les catégories suivantes d'arêtes ; une arête e de G est :
  - **candidate** si elle a au moins une extrémité dans T ;
  - **valide** si e est candidate et $T \cup e$ est acyclique ;

Algorithmes de calcul d'ACPM

- Les deux algorithmes que nous verrons rajoutent progressivement des arêtes à un sous-graphe T de G ;
- On fera la distinction entre les catégories suivantes d'arêtes ; une arête e de G est :
  - **candidate** si elle a au moins une extrémité dans T ;
  - **valide** si e est candidate et $T \cup e$ est acyclique ;
  - **sûre** si elle est valide et de poids minimum parmi toutes les arêtes valides ;

Algorithmes de calcul d'ACPM

- Les deux algorithmes que nous verrons rajoutent progressivement des arêtes à un sous-graphe T de G ;
- On fera la distinction entre les catégories suivantes d'arêtes ; une arête e de G est :
  - **candidate** si elle a au moins une extrémité dans T ;
  - **valide** si e est candidate et $T \cup e$ est acyclique ;
  - **sûre** si elle est valide et de poids minimum parmi toutes les arêtes valides ;
- Les deux algorithmes sont simples, mais nécessiteront des structures de données efficaces ;

L'algorithme de Prim

- L'algorithme de Prim construit un ACPM T de la manière suivante :

L'algorithme de Prim

- L'algorithme de Prim construit un ACPM T de la manière suivante :
  1. on part d'un sommet arbitraire, qu'on ajoute à un ensemble S de sommets explorés ;

L'algorithme de Prim

- L'algorithme de Prim construit un ACPM T de la manière suivante :
  1. on part d'un sommet arbitraire, qu'on ajoute à un ensemble S de sommets explorés ;
  2. à chaque étape, on rajoute à T une arête e sûre ; c'est-à-dire que :

L'algorithme de Prim

- L'algorithme de Prim construit un ACPM T de la manière suivante :
  1. on part d'un sommet arbitraire, qu'on ajoute à un ensemble S de sommets explorés ;
  2. à chaque étape, on rajoute à T une arête e sûre ; c'est-à-dire que :
     1. e possède une extrémité dans T ;

L'algorithme de Prim

- L'algorithme de Prim construit un ACPM T de la manière suivante :
  1. on part d'un sommet arbitraire, qu'on ajoute à un ensemble S de sommets explorés ;
  2. à chaque étape, on rajoute à T une arête e sûre ; c'est-à-dire que :
     1. e possède une extrémité dans T ;
     2. e est de poids minimum ;

L'algorithme de Prim

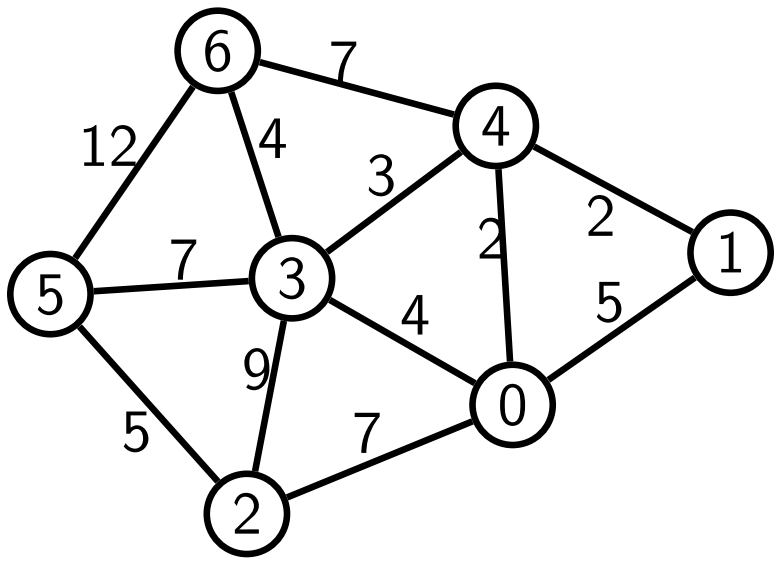
- L'algorithme de Prim construit un ACPM T de la manière suivante :
  1. on part d'un sommet arbitraire, qu'on ajoute à un ensemble S de sommets explorés ;
  2. à chaque étape, on rajoute à T une arête e sûre ; c'est-à-dire que :
     1. e possède une extrémité dans T ;
     2. e est de poids minimum ;
     3. e ne crée pas de cycle dans T ;

L'algorithme de Prim

- L'algorithme de Prim construit un ACPM T de la manière suivante :
  1. on part d'un sommet arbitraire, qu'on ajoute à un ensemble S de sommets explorés ;
  2. à chaque étape, on rajoute à T une arête e sûre ; c'est-à-dire que :
     1. e possède une extrémité dans T ;
     2. e est de poids minimum ;
     3. e ne crée pas de cycle dans T ;
- On résoud les ambigüités arbitrairement ;

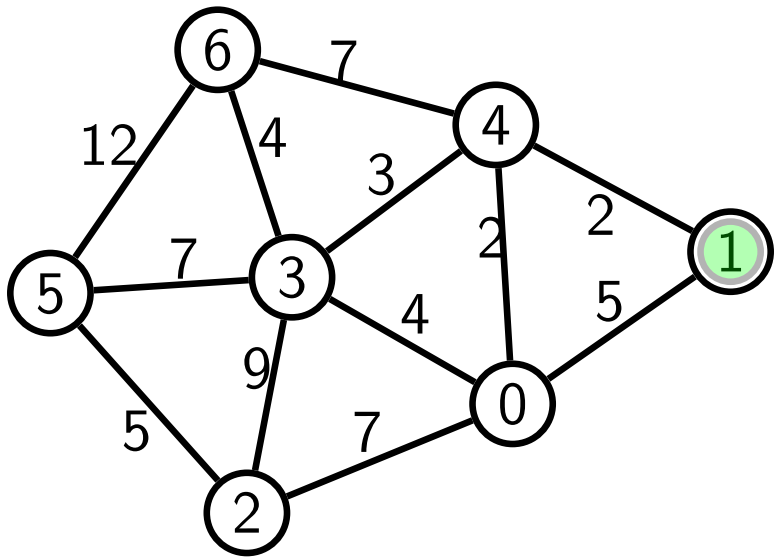
Déroulement de l'algorithme de Prim

Exemple 2 (départ = 1)



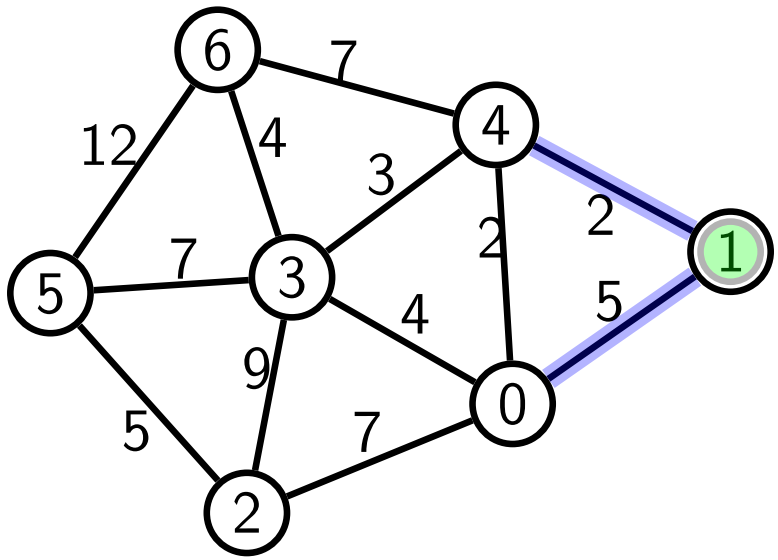
Déroulement de l'algorithme de Prim

Exemple 2 (départ = 1)



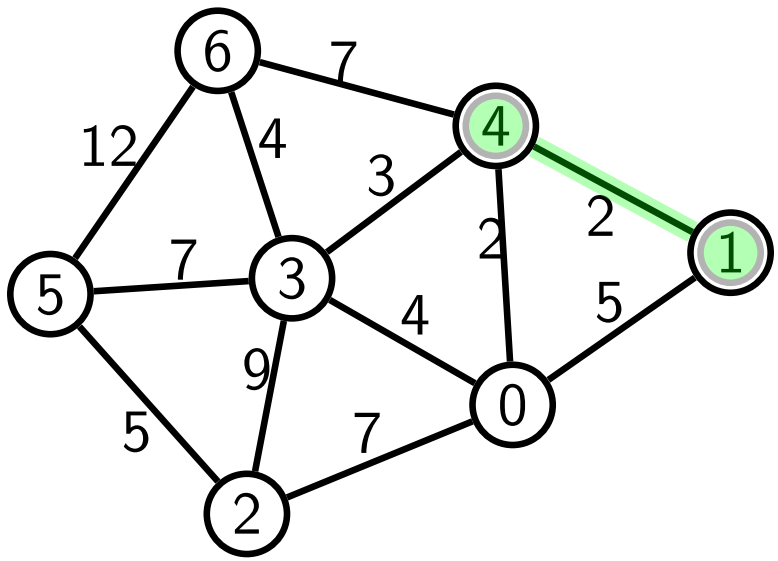
Déroulement de l'algorithme de Prim

Exemple 2 (départ = 1)



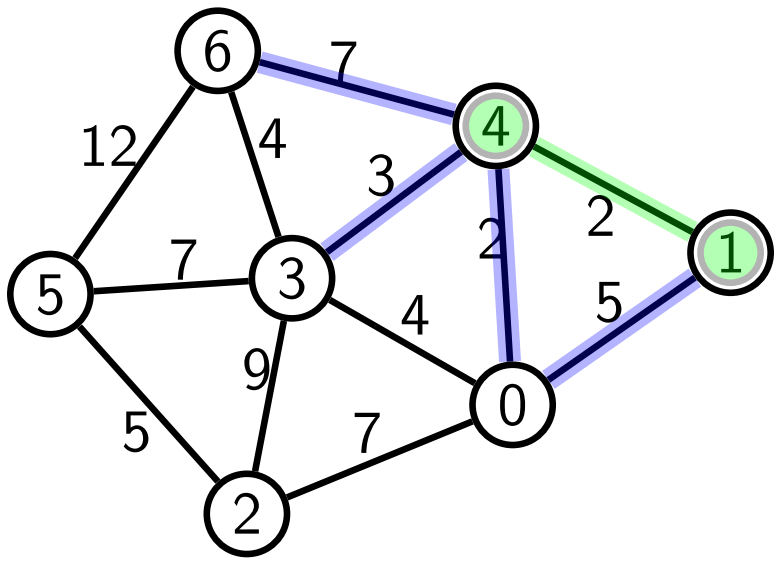
Déroulement de l'algorithme de Prim

Exemple 2 (départ = 1)



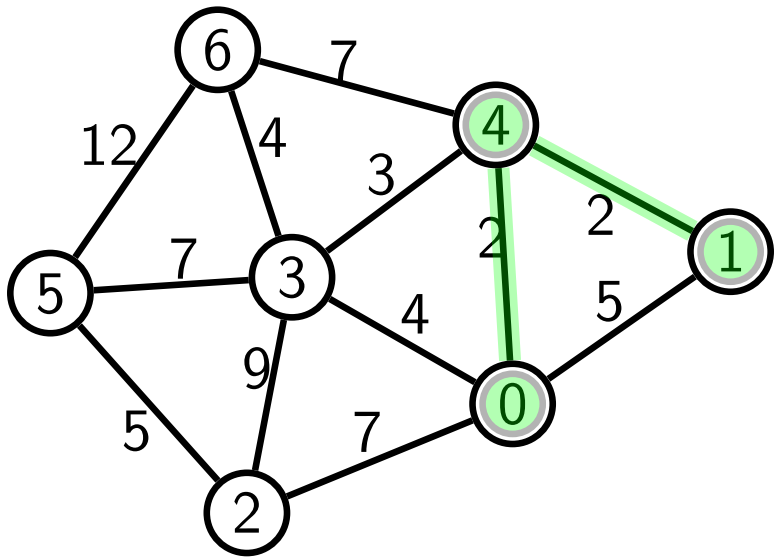
Déroulement de l'algorithme de Prim

Exemple 2 (départ = 1)



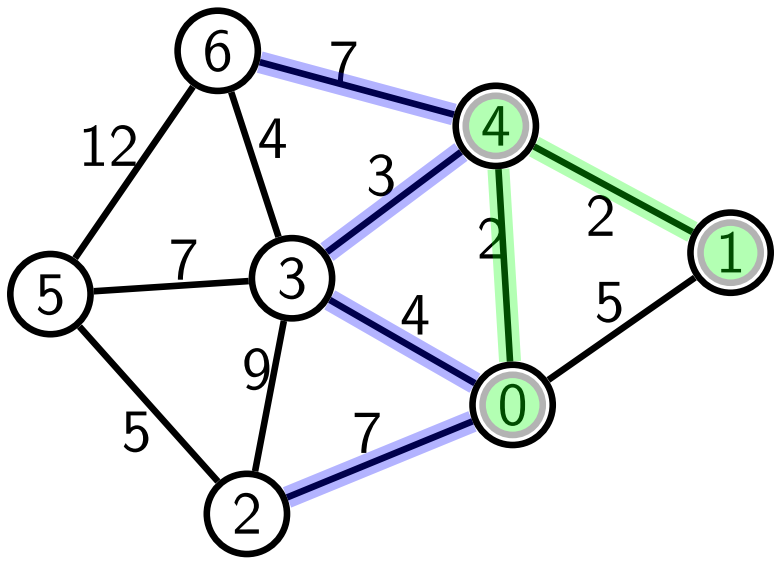
Déroulement de l'algorithme de Prim

Exemple 2 (départ = 1)



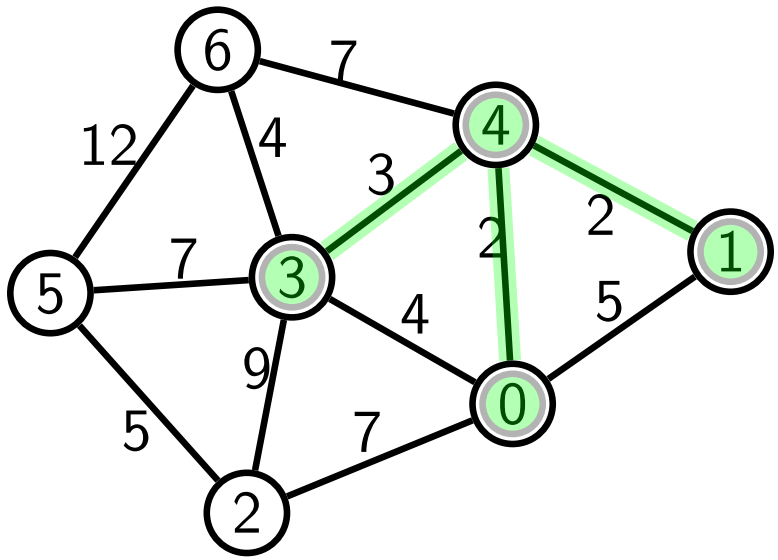
Déroulement de l'algorithme de Prim

Exemple 2 (départ = 1)



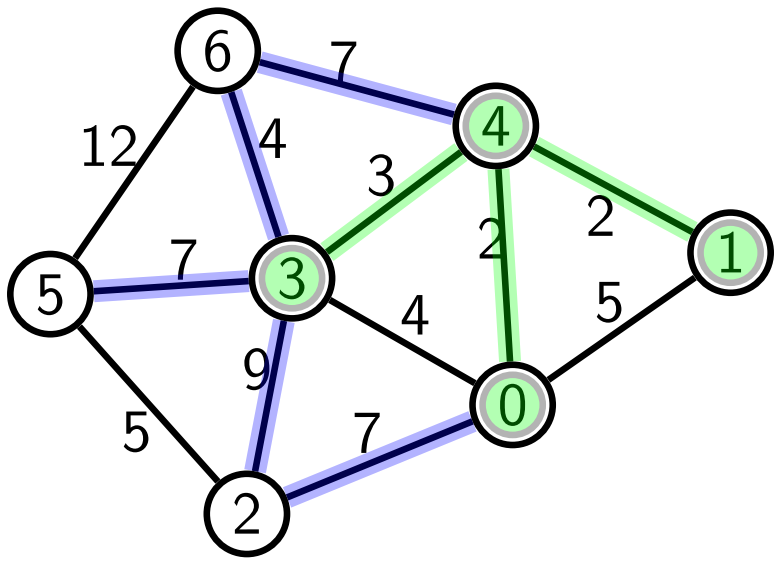
Déroulement de l'algorithme de Prim

Exemple 2 (départ = 1)



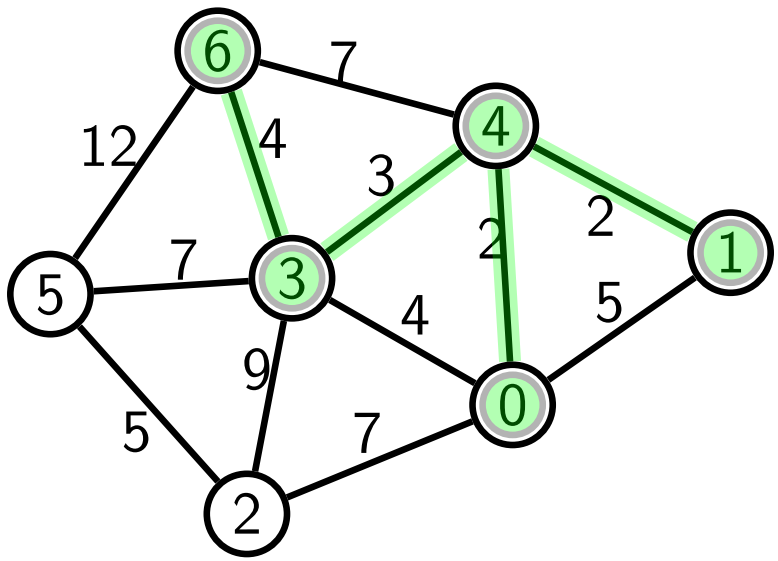
Déroulement de l'algorithme de Prim

Exemple 2 (départ = 1)



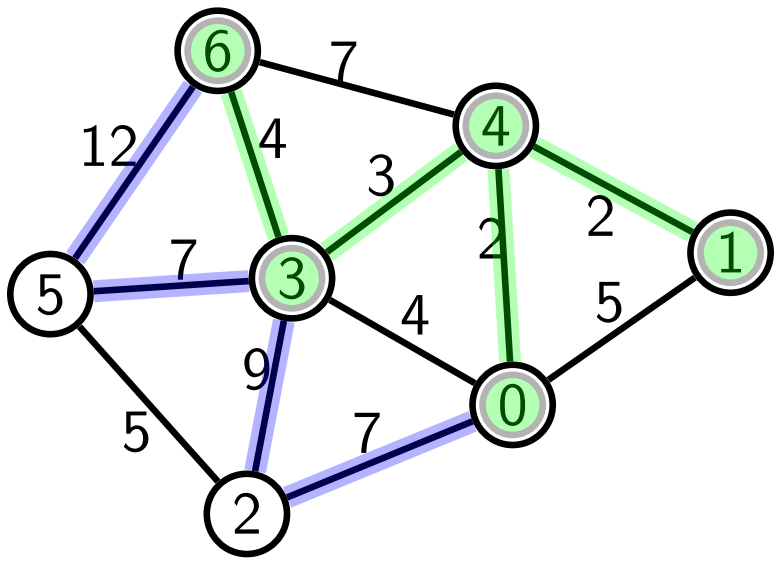
Déroulement de l'algorithme de Prim

Exemple 2 (départ = 1)



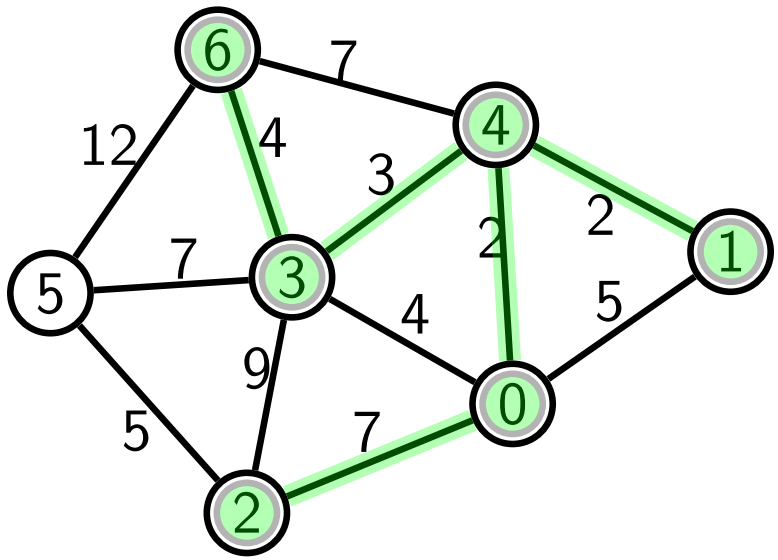
Déroulement de l'algorithme de Prim

Exemple 2 (départ = 1)



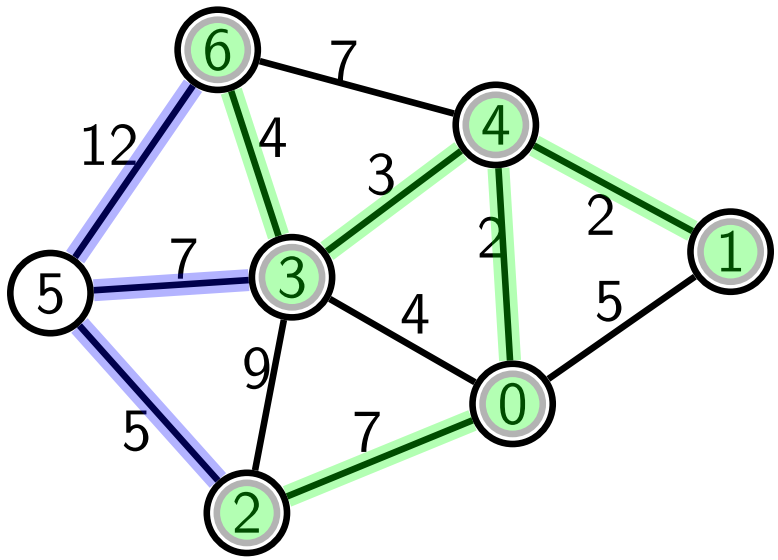
Déroulement de l'algorithme de Prim

Exemple 2 (départ = 1)



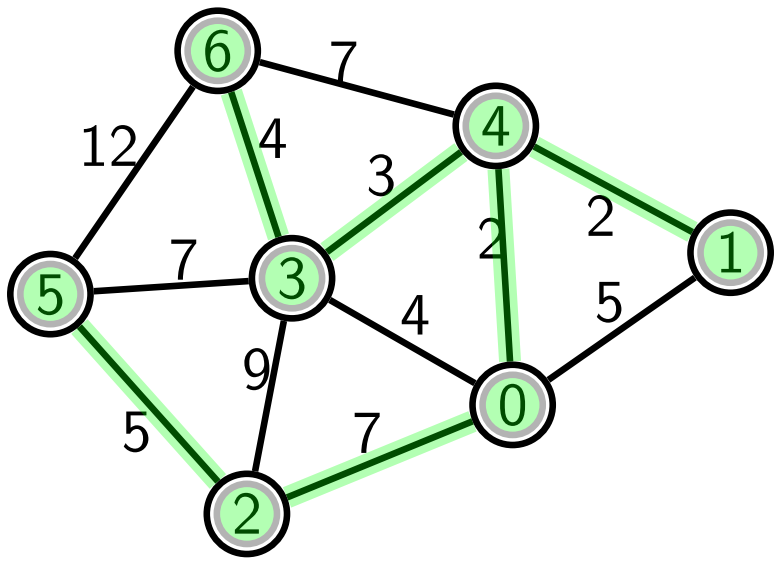
Déroulement de l'algorithme de Prim

Exemple 2 (départ = 1)



Déroulement de l'algorithme de Prim

Exemple 2 (départ = 1)



Prim : validité des arêtes

- Étant donnée une arête $e = \{u, v\}$ de G : comment sait-on si elle est valide par rapport à l'arbre T ?

Prim : validité des arêtes

- Étant donnée une arête $e = \{u, v\}$ de G : comment sait-on si elle est valide par rapport à l'arbre T ?
- $\{u, v\}$ est valide si et seulement si $T \cup e$ est acyclique ;

Prim : validité des arêtes

- Étant donnée une arête $e = \{u, v\}$ de G : comment sait-on si elle est valide par rapport à l'arbre T ?
- $\{u, v\}$ est valide si et seulement si $T \cup e$ est acyclique ;
- $\Leftrightarrow$ $u \in V(T)$ ou $v \in V(T)$ — mais pas les deux ;

Prim : validité des arêtes

- Étant donnée une arête $e = \{u, v\}$ de G : comment sait-on si elle est valide par rapport à l'arbre T ?
- $\{u, v\}$ est valide si et seulement si $T \cup e$ est acyclique ;
- $\Leftrightarrow$ $u \in V(T)$ ou $v \in V(T)$ — mais pas les deux ;
- Il suffit donc de marquer les sommets de T (ou hors de T) ;

Prim : sélection parmi les arêtes valides

- Comment sélectionner une arête valide de poids minimum ?

Prim : sélection parmi les arêtes valides

- Comment sélectionner une arête valide de poids minimum ?
- Idée : trier les arêtes par poids croissant ;

Prim : sélection parmi les arêtes valides

- Comment sélectionner une arête valide de poids minimum ?
- Idée : trier les arêtes par poids croissant ;
- Oui, mais la validité des arêtes change à mesure que T évolue ;

Prim : sélection parmi les arêtes valides

- Comment sélectionner une arête valide de poids minimum ?
- Idée : trier les arêtes par poids croissant ;
- Oui, mais la validité des arêtes change à mesure que T évolue ;
- Le tri ne nous empêcherait pas de devoir parcourir la structure à chaque itération $\Rightarrow O(|E|)$ pour la sélection ;

Prim : sélection parmi les arêtes valides

- Comment sélectionner une arête valide de poids minimum ?
- Idée : trier les arêtes par poids croissant ;
- Oui, mais la validité des arêtes change à mesure que T évolue ;
- Le tri ne nous empêcherait pas de devoir parcourir la structure à chaque itération $\Rightarrow O(|E|)$ pour la sélection ;

- Solution plus efficace : utiliser un **tas** d'arêtes valides ;

Tas

Un **tas** (ou *heap*) est un arbre binaire enraciné dont tout sommet s ayant pour fils gauche g et pour fils droit d vérifie :

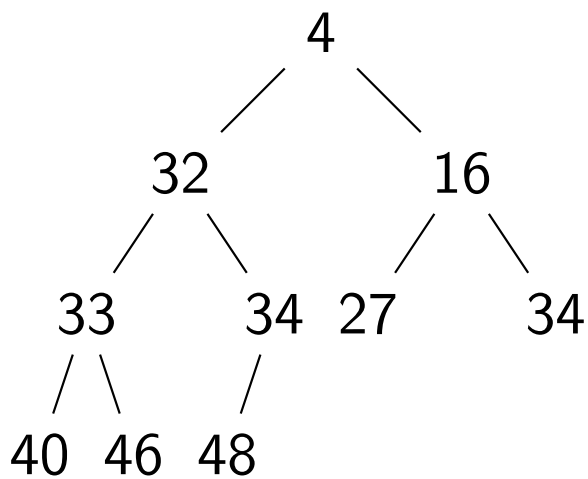
$$s.\text{valeur} = \min(s.\text{valeur}, g.\text{valeur}, d.\text{valeur}).$$

Tas

Un **tas** (ou *heap*) est un arbre binaire enraciné dont tout sommet s ayant pour fils gauche g et pour fils droit d vérifie :

$$s.\text{valeur} = \min(s.\text{valeur}, g.\text{valeur}, d.\text{valeur}).$$

Exemple 3



4	32	16	33	34	27	34	40	46	48
0	1	2	3	4	5	6	7	8	9

La classe Tas

- On supposera qu'une classe Tas est disponible avec les méthodes suivantes :

La classe Tas

- On supposera qu'une classe Tas est disponible avec les méthodes suivantes :
  - un constructeur Tas(S), qui organise les données de S sous la forme d'un tas en $O(|S|)$;

La classe Tas

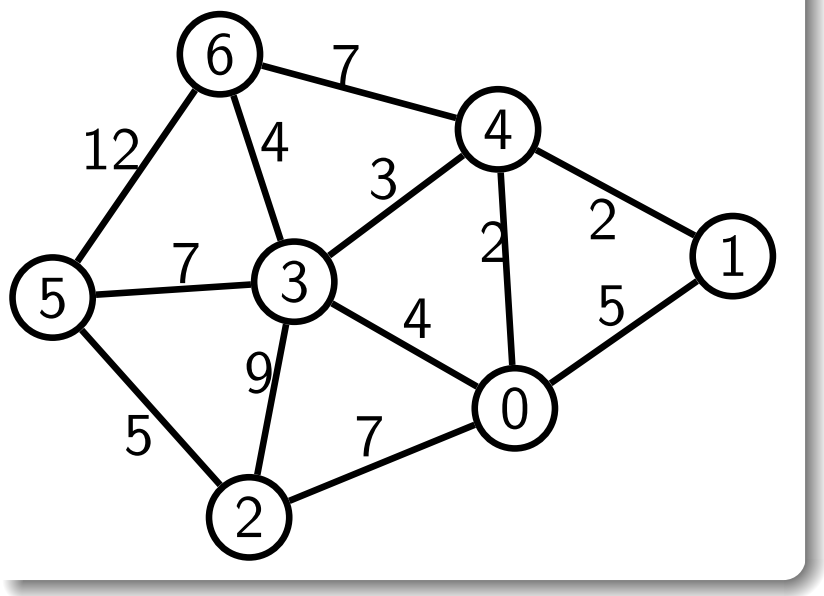
- On supposera qu'une classe Tas est disponible avec les méthodes suivantes :
  - un constructeur Tas(S), qui organise les données de S sous la forme d'un tas en $O(|S|)$;
  - une méthode insérer(élément), qui ajoute un élément au tas T en $O(\log|T|)$ **et** garantit que le résultat après insertion est toujours un tas ;

La classe Tas

- On supposera qu'une classe Tas est disponible avec les méthodes suivantes :
  - un constructeur Tas(S), qui organise les données de S sous la forme d'un tas en $O(|S|)$;
  - une méthode insérer(élément), qui ajoute un élément au tas T en $O(\log|T|)$ **et** garantit que le résultat après insertion est toujours un tas ;
  - une méthode extraire_minimum(), qui extrait et renvoie le minimum du tas T en $O(\log|T|)$ **et** garantit que le résultat après extraction est toujours un tas.

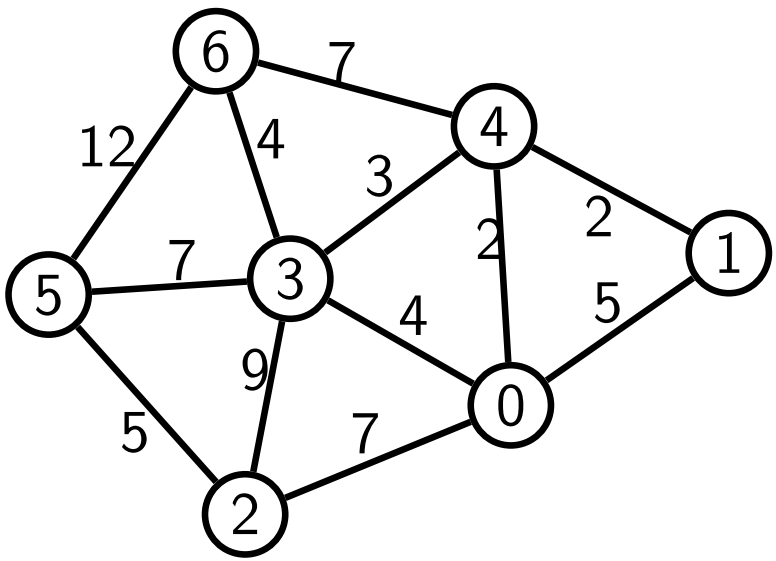
Déroulement de l'algorithme de Prim

Exemple 4 (départ = 1)



Déroulement de l'algorithme de Prim

Exemple 4 (départ = 1)



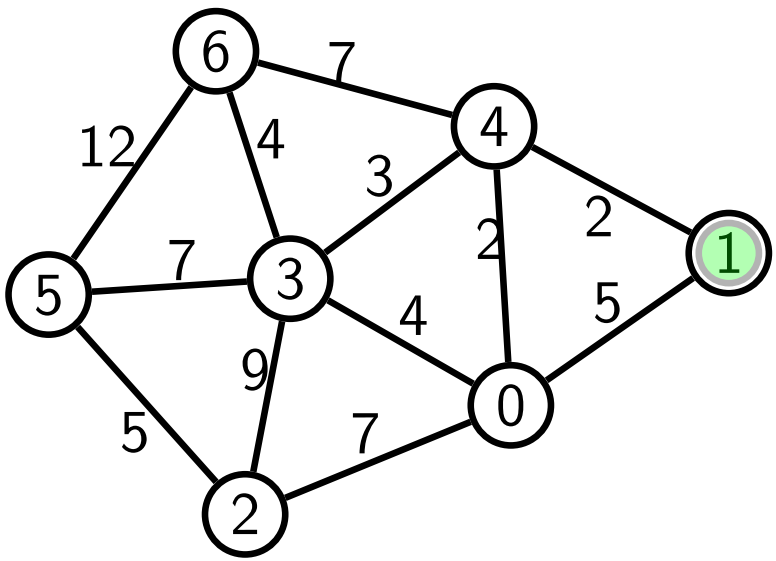
Les coulisses

	0	1	2	3	4	5	6
hors_arbre :	✓	✓	✓	✓	✓	✓	✓

tas :

Déroulement de l'algorithme de Prim

Exemple 4 (départ = 1)



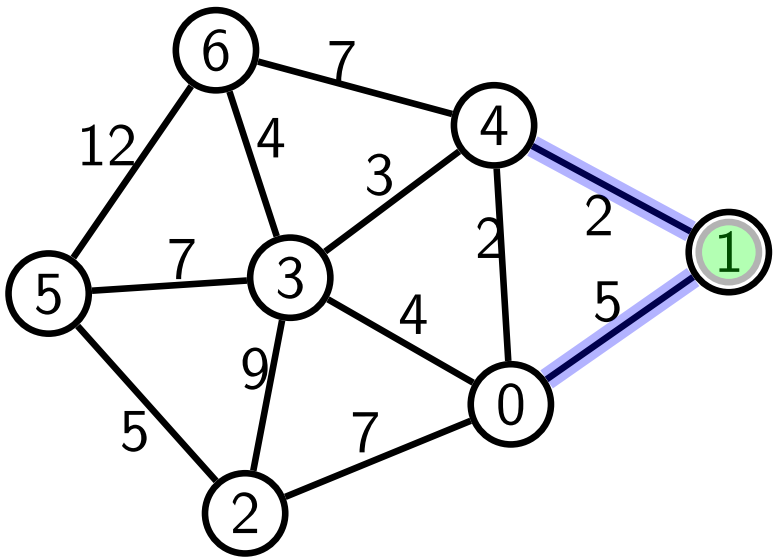
Les coulisses

	0	1	2	3	4	5	6
hors_arbre :	✓		✓	✓	✓	✓	✓

tas :

Déroulement de l'algorithme de Prim

Exemple 4 (départ = 1)



Les coulisses

	0	1	2	3	4	5	6
hors_arbre :	✓		✓	✓	✓	✓	✓

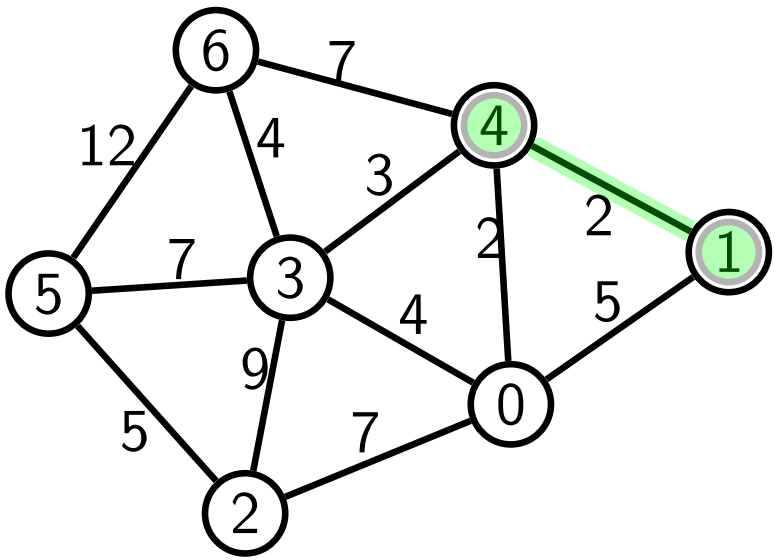
tas :

({1, 4}, 2)

({0, 1}, 5)

Déroulement de l'algorithme de Prim

Exemple 4 (départ = 1)



Les coulisses

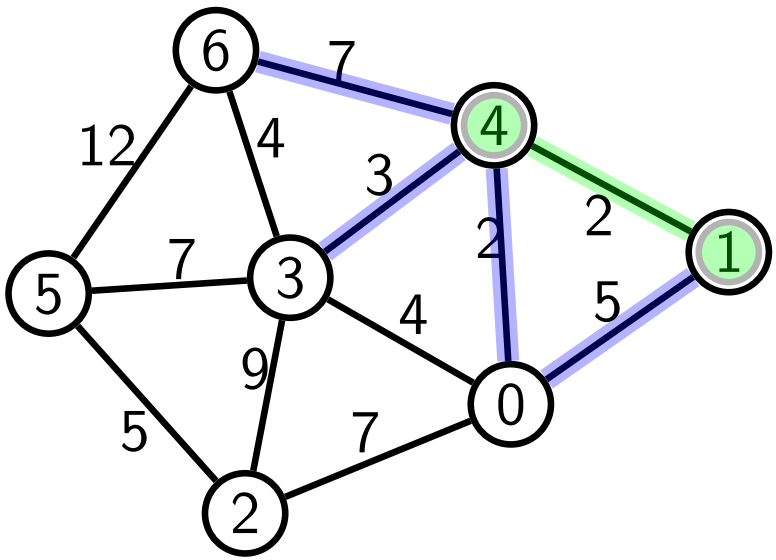
	0	1	2	3	4	5	6
hors_arbre :	✓		✓	✓		✓	✓

tas :

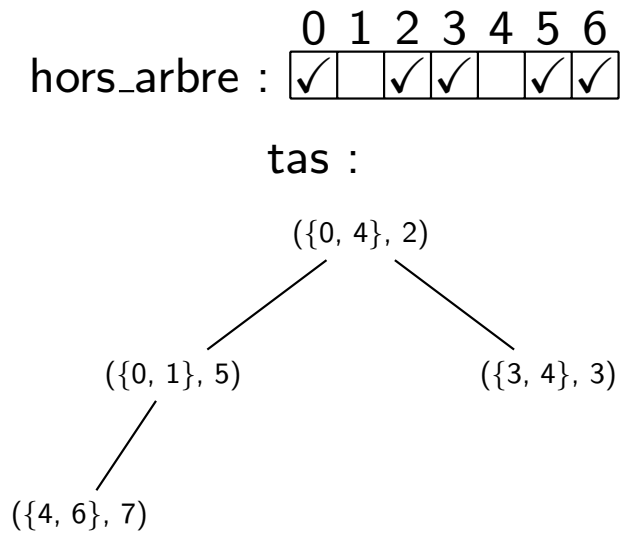
({0, 1}, 5)

Déroulement de l'algorithme de Prim

Exemple 4 (départ = 1)

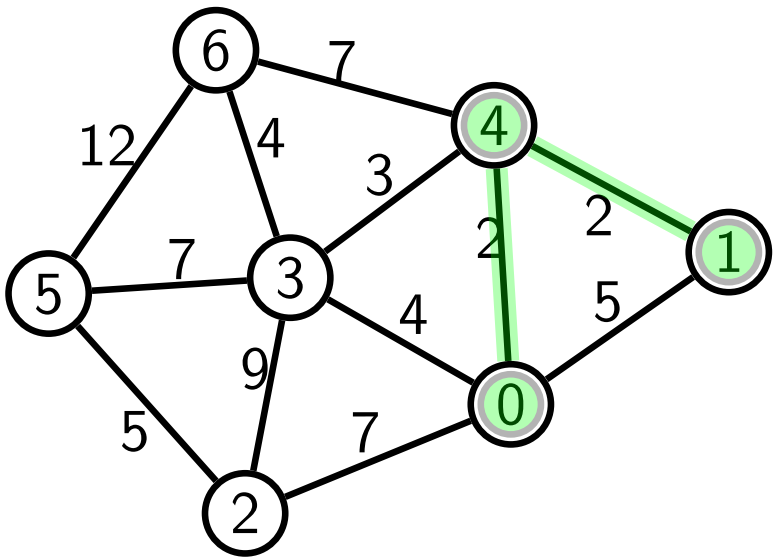


Les coulisses

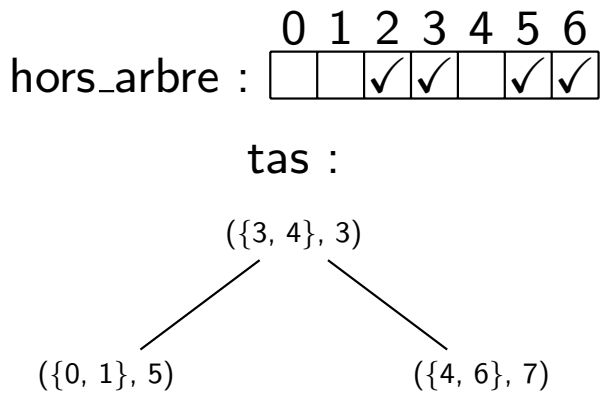


Déroulement de l'algorithme de Prim

Exemple 4 (départ = 1)

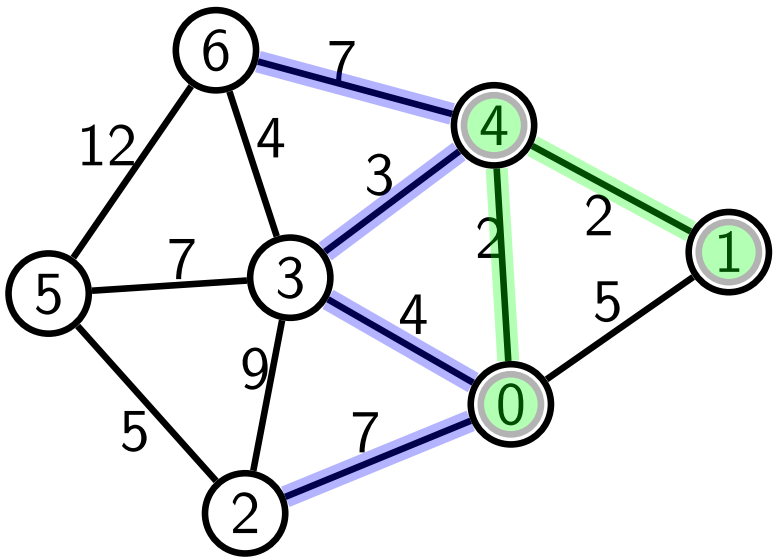


Les coulisses

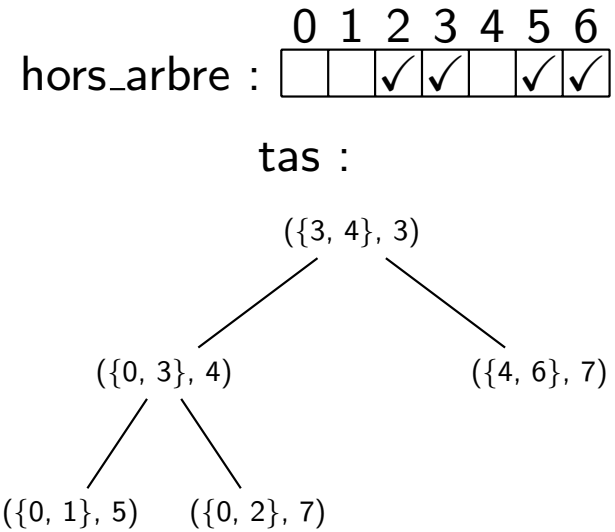


Déroulement de l'algorithme de Prim

Exemple 4 (départ = 1)

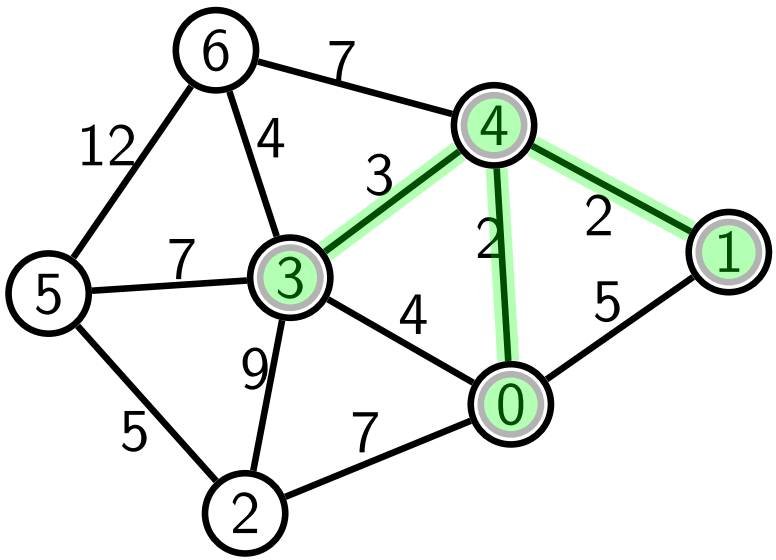


Les coulisses

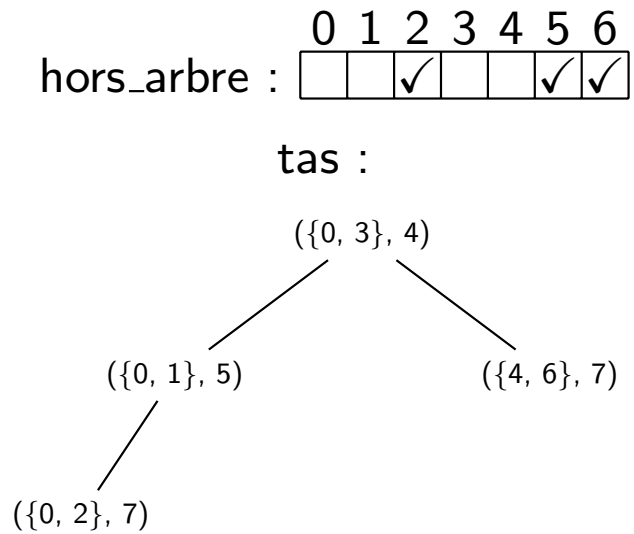


Déroulement de l'algorithme de Prim

Exemple 4 (départ = 1)

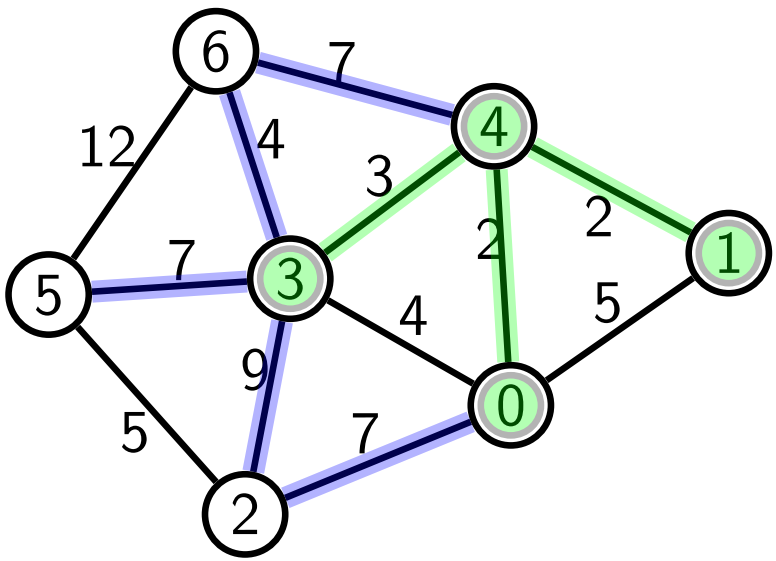


Les coulisses

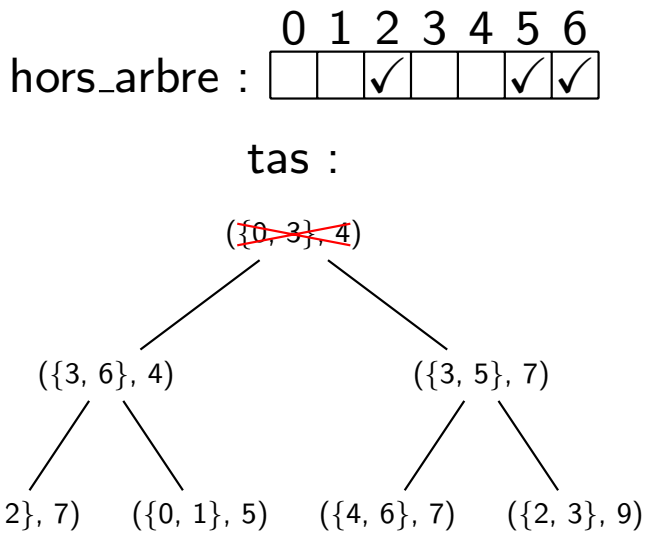


Déroulement de l'algorithme de Prim

Exemple 4 (départ = 1)

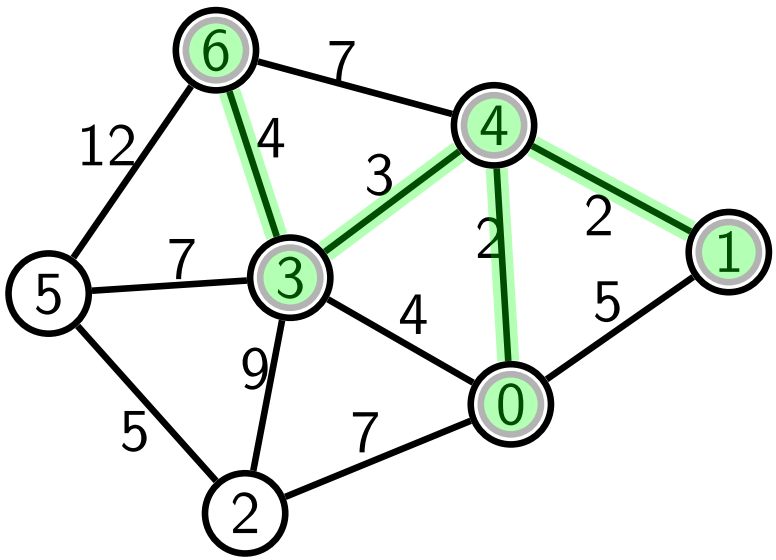


Les coulisses

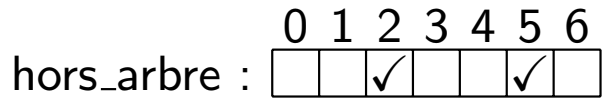


Déroulement de l'algorithme de Prim

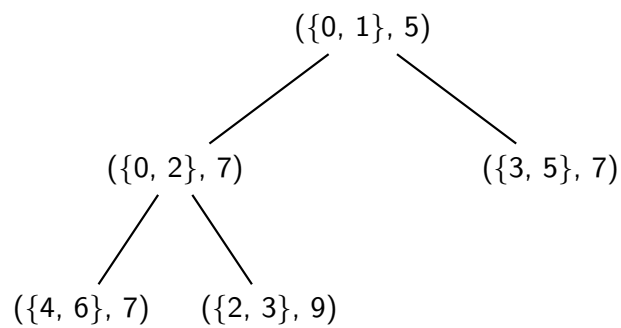
Exemple 4 (départ = 1)



Les coulisses

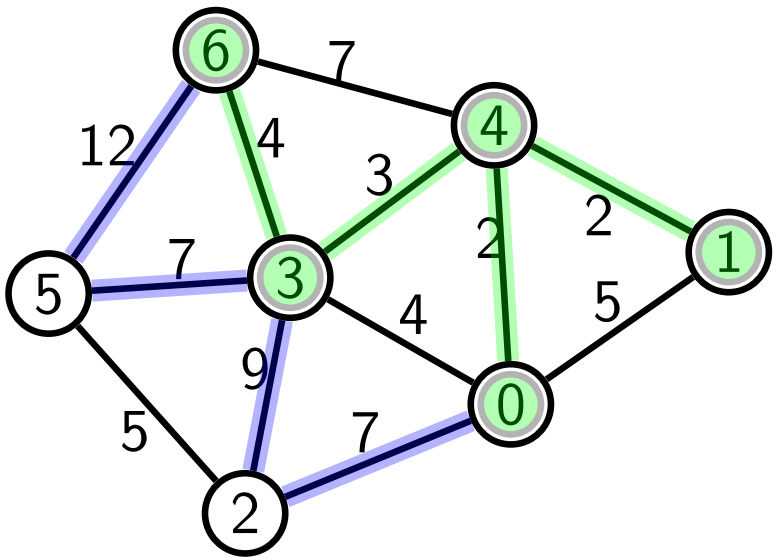


tas :

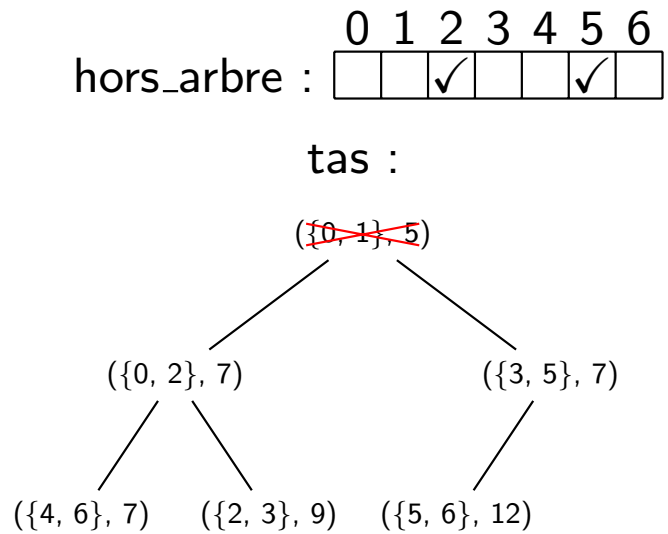


Déroulement de l'algorithme de Prim

Exemple 4 (départ = 1)

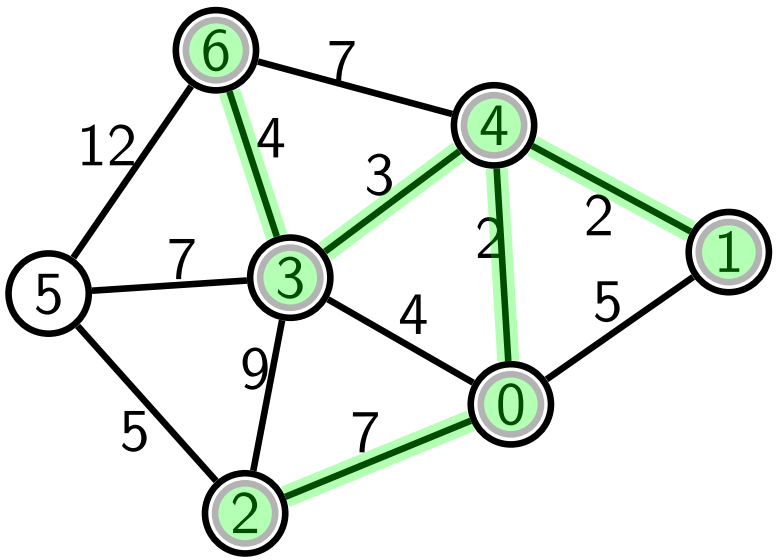


Les coulisses

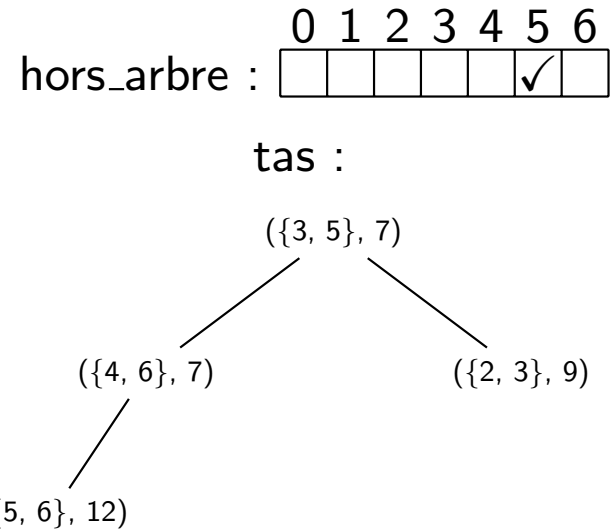


Déroulement de l'algorithme de Prim

Exemple 4 (départ = 1)

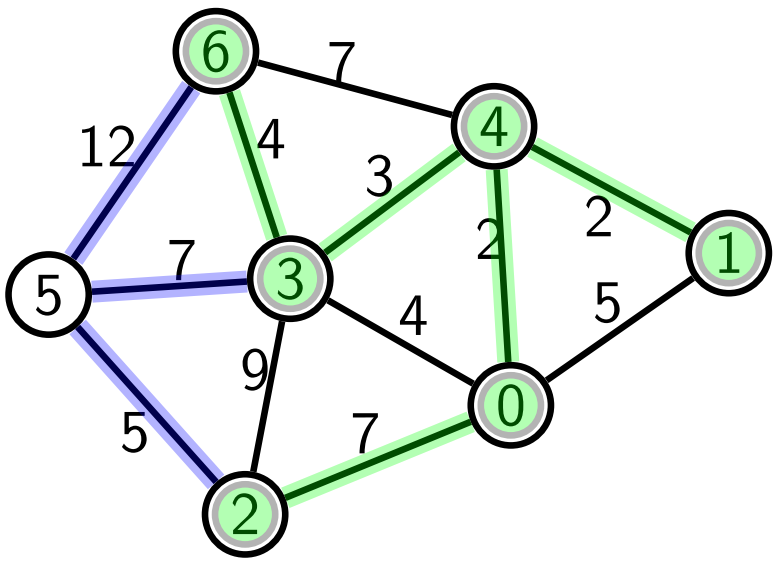


Les coulisses

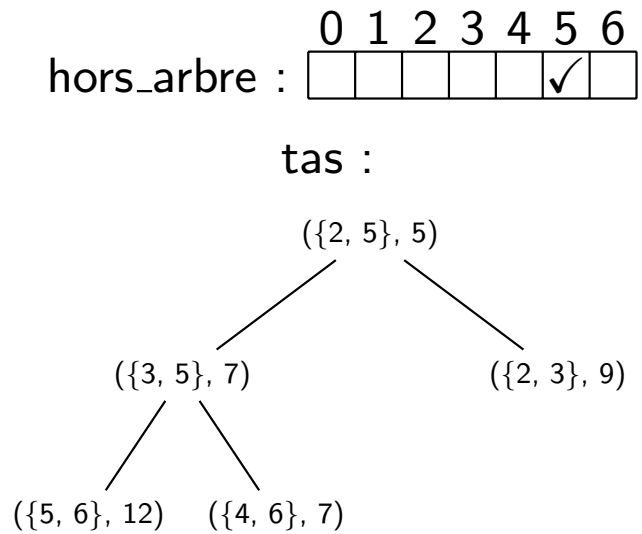


Déroulement de l'algorithme de Prim

Exemple 4 (départ = 1)

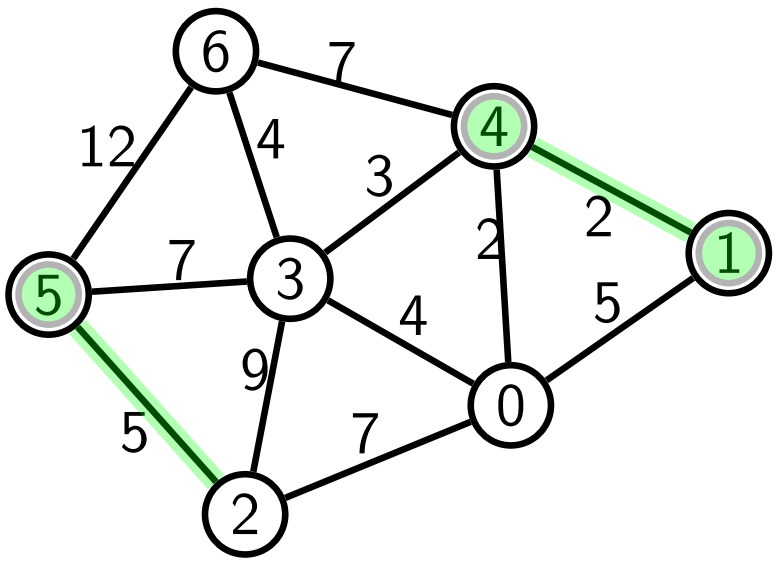


Les coulisses

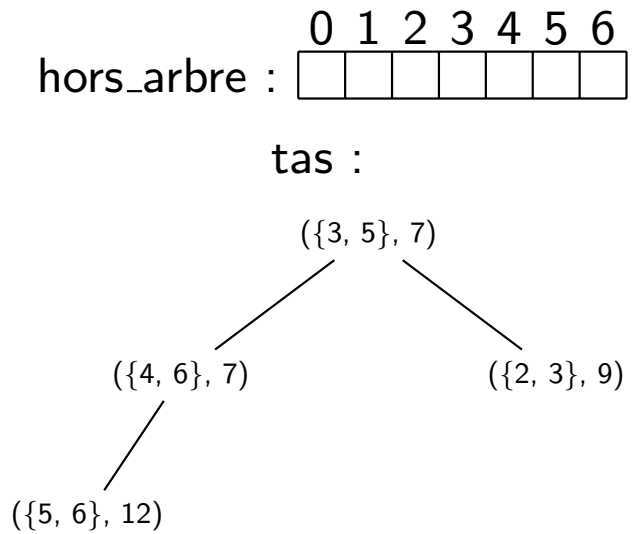


Déroulement de l'algorithme de Prim

Exemple 4 (départ = 1)



Les coulisses



Stockage des arêtes

Lorsqu'on découvre un nouveau sommet, on enregistre les nouvelles arêtes valides ;

Algorithme 1 : StockerAretesValides(G, u, S, hors_arbre)

Entrées : un graphe pondéré non orienté G, un sommet u de G, un tas d'arêtes S et un tableau booléen hors_arbre.
Résultat : les arêtes valides incidentes à u sont ajoutées à S.

```
1 pour chaque v ∈ G.voisins(u) faire
2     si hors_arbre[v] alors S.insérer((u, v, G.poids_arête(u, v))) ;
```

Extraction des arêtes

Attention : les arêtes sont valides quand on les insère, mais pas nécessairement quand on les extrait !

Algorithme 2 : ExtraireAreteSure(S, hors_arbre)

Entrées : un tas S d'arêtes et un tableau booléen hors_arbre.
Résultat : une arête sûre (ou factice s'il n'y en a pas) est extraite de S et renvoyée ; les arêtes invalides éventuellement rencontrées sont éliminées.

```
1 tant que S.pas_vide() faire
2 |   (u, v, p) ← S.extraire_minimum();
3 |   si hors_arbre[u] ≠ hors_arbre[v] alors renvoyer (u, v, p) ;
4 renvoyer (NIL, NIL, +∞);
```

L'algorithme de Prim proprement dit

Algorithme 3 : PRIM(G, départ)

Entrées : un graphe pondéré non orienté G, un sommet de départ.
Sortie : un ACPM pour la composante connexe de G contenant départ.

```
1  arbre ← GraphePondéré();
2  arbre.ajouter_sommet(départ);
3  hors_arbre ← tableau(G.nombre_sommets(), VRAI);
4  hors_arbre[départ] ← FAUX;
5  candidates ← Tas() ;
6  StockerAretesValides(G, départ, candidates, hors_arbre) ;
7  tant que VRAI faire
8  |   (u, v, p) ← ExtraireAreteSure(candidates, hors_arbre) ;
9  |   si u = NIL alors renvoyer arbre ;
10 |   si ¬ hors_arbre[u] alors échanger u et v;
11 |   arbre.ajouter_arête(u, v, p);
12 |   hors_arbre[u] ← FAUX;
13 |   StockerAretesValides(G, u, candidates, hors_arbre) ;
14 renvoyer arbre;
```

Complexité de l'algorithme de Prim

- Supposons que G est implémenté à l'aide de listes d'adjacence ($\Rightarrow$ G.voisins(v) est en $O(\deg(v))$;

Complexité de l'algorithme de Prim

- Supposons que G est implémenté à l'aide de listes d'adjacence ($\Rightarrow$ G.voisins(v) est en $O(\deg(v))$;
- On passe $O(|V|)$ fois dans la boucle principale ;

Complexité de l'algorithme de Prim

- Supposons que G est implémenté à l'aide de listes d'adjacence ($\Rightarrow$ G.voisins(v) est en $O(\deg(v))$;
- On passe $O(|V|)$ fois dans la boucle principale ;
- Le tas contient au pire $|E|$ arêtes ;

Complexité de l'algorithme de Prim

- Supposons que G est implémenté à l'aide de listes d'adjacence ($\Rightarrow$ G.voisins(v) est en $O(\deg(v))$;
- On passe $O(|V|)$ fois dans la boucle principale ;
- Le tas contient au pire $|E|$ arêtes ;
- Les insertions et extractions se font en temps $O(\log |E|)$;

Complexité de l'algorithme de Prim

- Supposons que G est implémenté à l'aide de listes d'adjacence ($\Rightarrow$ G.voisins(v) est en $O(\deg(v))$;
- On passe $O(|V|)$ fois dans la boucle principale ;
- Le tas contient au pire $|E|$ arêtes ;
- Les insertions et extractions se font en temps $O(\log |E|)$;

$\Rightarrow O((|V| + |E|) \log |E|) = O(|E| \log |V|)$;

Complexité de l'algorithme de Prim

- Supposons que G est implémenté à l'aide de listes d'adjacence ($\Rightarrow$ G.voisins(v) est en $O(\deg(v))$;
- On passe $O(|V|)$ fois dans la boucle principale ;
- Le tas contient au pire $|E|$ arêtes ;
- Les insertions et extractions se font en temps $O(\log |E|)$;

  $\Rightarrow O((|V| + |E|) \log |E|) = O(|E| \log |V|)$;
- Il est possible d'obtenir du $O(|E| + |V| \log |V|)$ avec des *tas de Fibonacci* [1] ;

Forêts couvrantes de poids minimum

- Attention : l'algorithme de Prim n'explore que la composante connexe contenant le sommet de départ ;

Forêts couvrantes de poids minimum

- Attention : l'algorithme de Prim n'explore que la composante connexe contenant le sommet de départ ;
- Si on veut une **forêt** couvrante de poids minimum (FCPM), on fait comme pour l'identification des composantes connexes ;

L'algorithme de Kruskal

- L'algorithme de Kruskal construit une FCPM F de la manière suivante :

L'algorithme de Kruskal

- L'algorithme de Kruskal construit une FCPM F de la manière suivante :
  1. tous les sommets du graphe font partie de F ;

L'algorithme de Kruskal

- L'algorithme de Kruskal construit une FCPM F de la manière suivante :
  1. tous les sommets du graphe font partie de F ;
  2. à chaque étape, on rajoute à F une arête e satisfaisant les conditions suivantes :

L'algorithme de Kruskal

- L'algorithme de Kruskal construit une FCPM F de la manière suivante :
  1. tous les sommets du graphe font partie de F ;
  2. à chaque étape, on rajoute à F une arête e satisfaisant les conditions suivantes :
     1. e est de poids minimum ;

L'algorithme de Kruskal

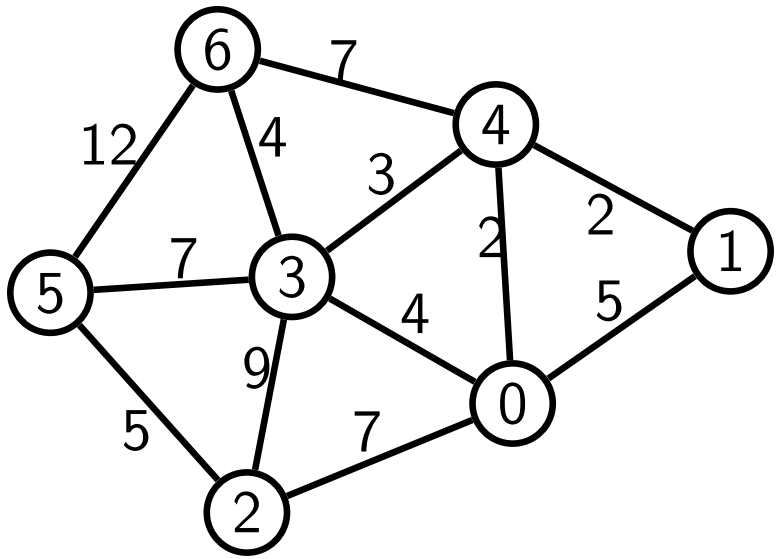
- L'algorithme de Kruskal construit une FCPM F de la manière suivante :
  1. tous les sommets du graphe font partie de F ;
  2. à chaque étape, on rajoute à F une arête e satisfaisant les conditions suivantes :
     1. e est de poids minimum ;
     2. e ne crée pas de cycle dans F ;

L'algorithme de Kruskal

- L'algorithme de Kruskal construit une FCPM F de la manière suivante :
  1. tous les sommets du graphe font partie de F ;
  2. à chaque étape, on rajoute à F une arête e satisfaisant les conditions suivantes :
     1. e est de poids minimum ;
     2. e ne crée pas de cycle dans F ;
- On résoud les ambigüités arbitrairement ;

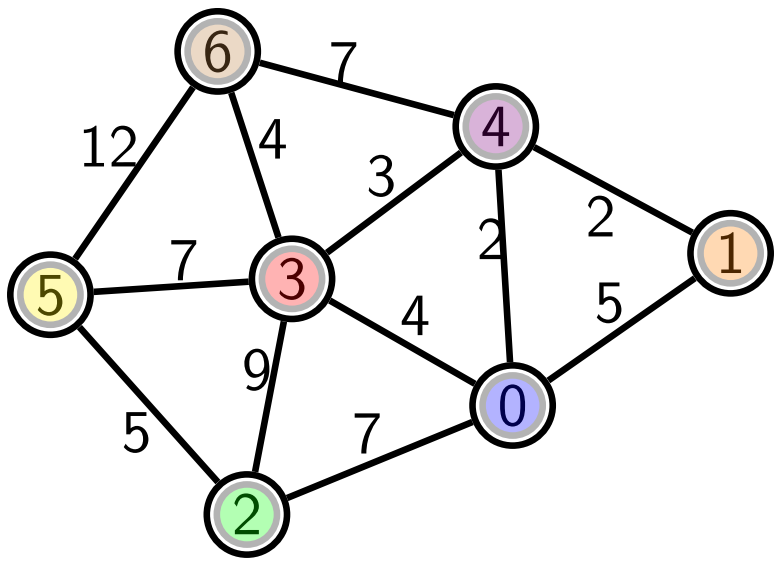
Déroulement de l'algorithme de Kruskal

Exemple 5



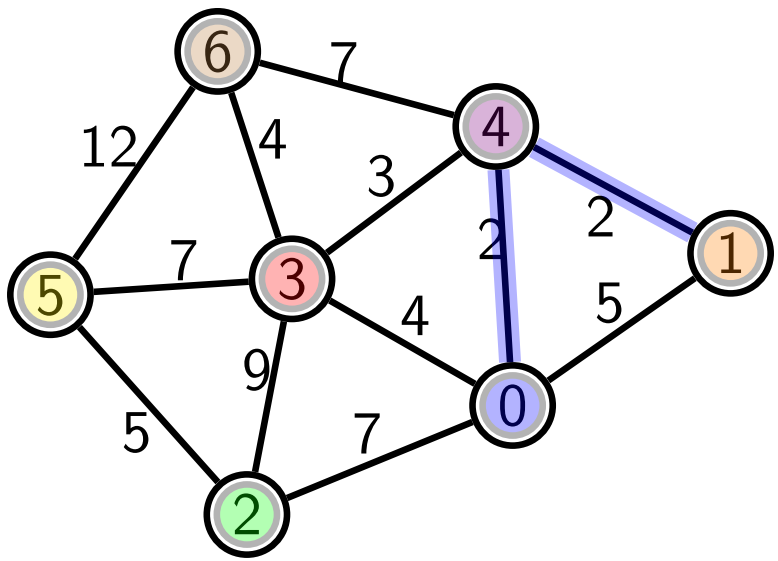
Déroulement de l'algorithme de Kruskal

Exemple 5



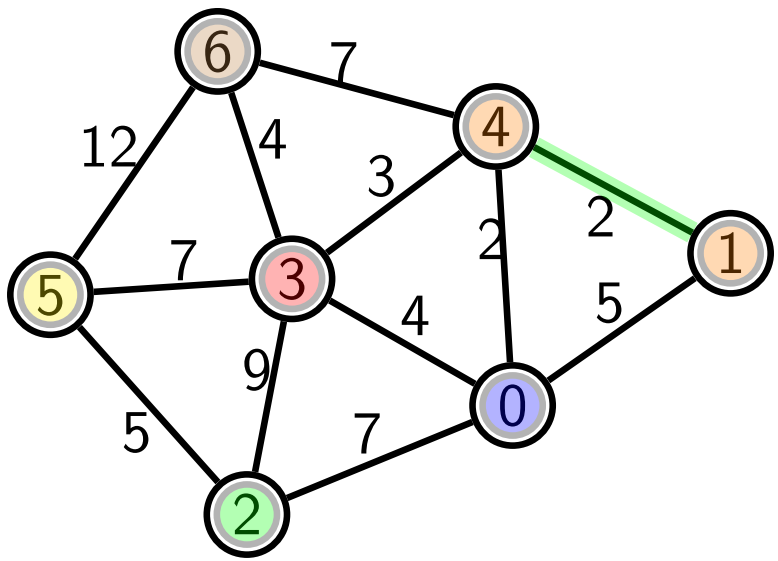
Déroulement de l'algorithme de Kruskal

Exemple 5



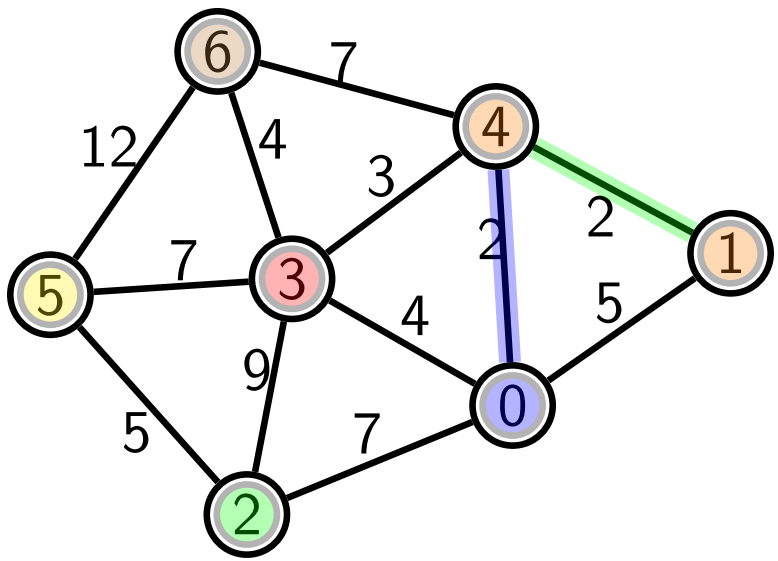
Déroulement de l'algorithme de Kruskal

Exemple 5



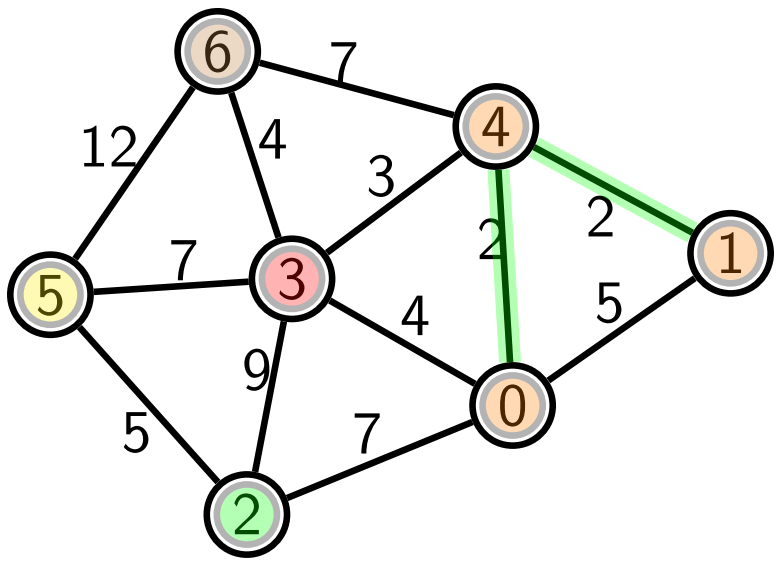
Déroulement de l'algorithme de Kruskal

Exemple 5



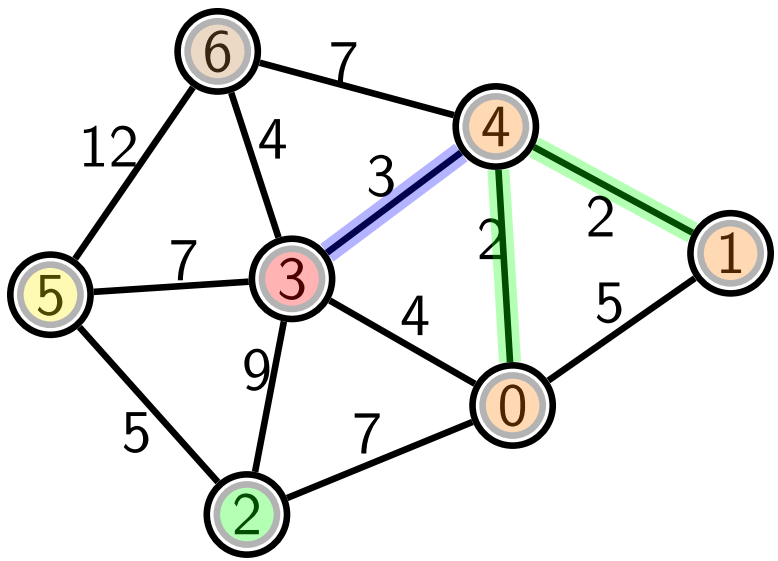
Déroulement de l'algorithme de Kruskal

Exemple 5



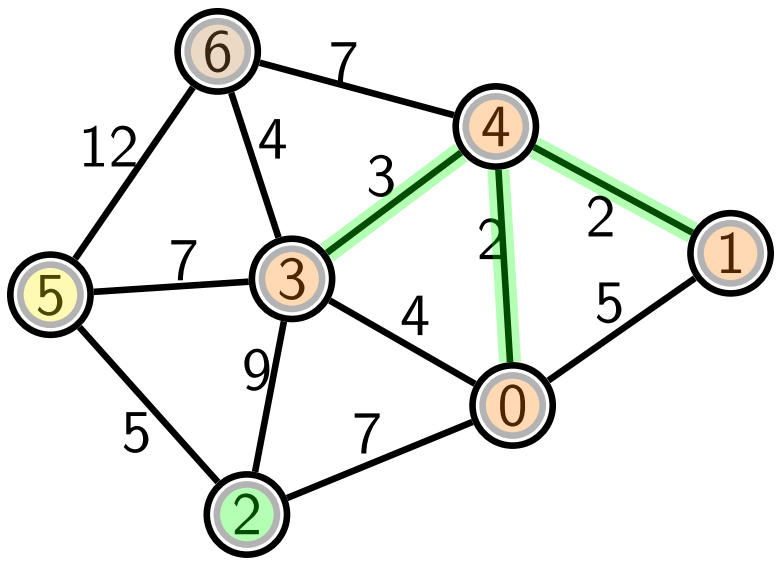
Déroulement de l'algorithme de Kruskal

Exemple 5



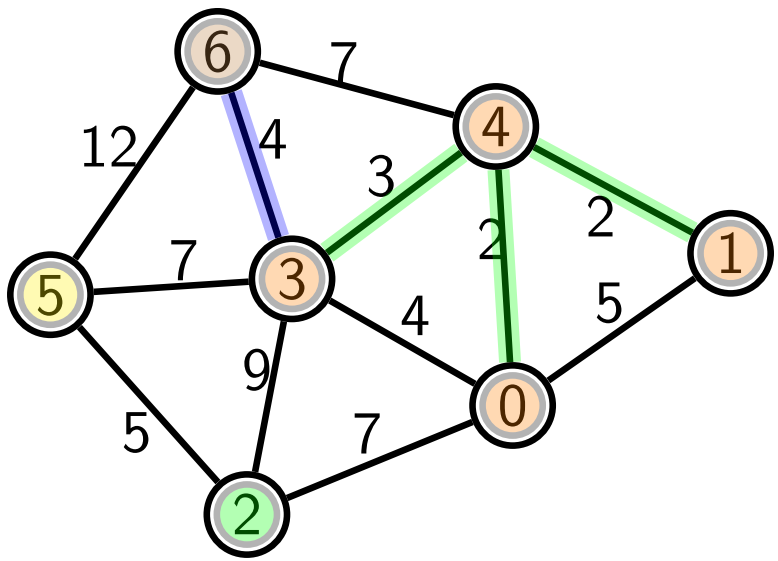
Déroulement de l'algorithme de Kruskal

Exemple 5



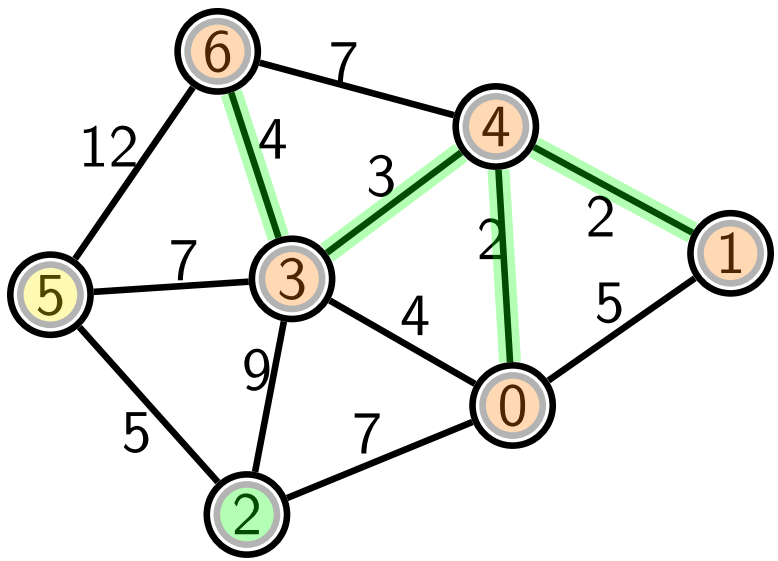
Déroulement de l'algorithme de Kruskal

Exemple 5



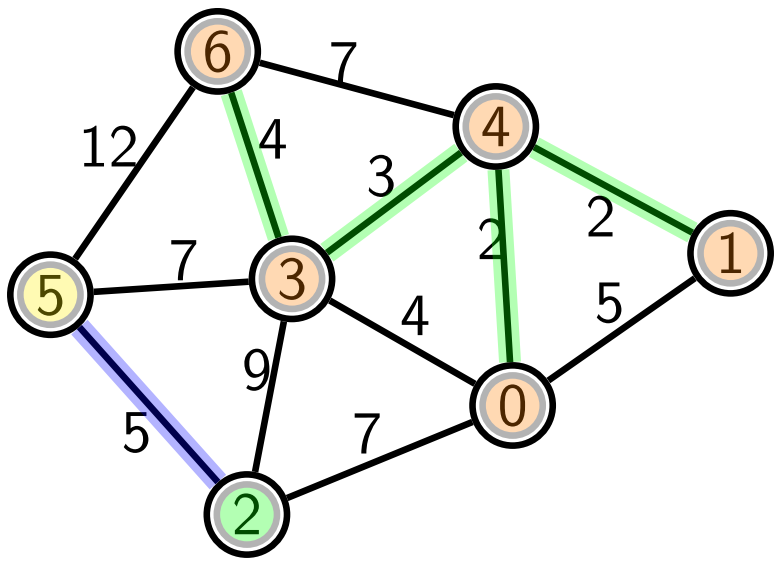
Déroulement de l'algorithme de Kruskal

Exemple 5



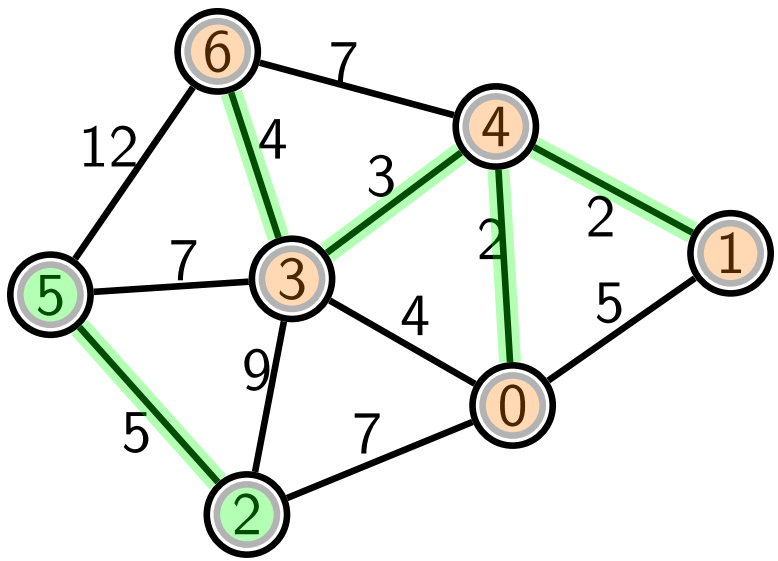
Déroulement de l'algorithme de Kruskal

Exemple 5



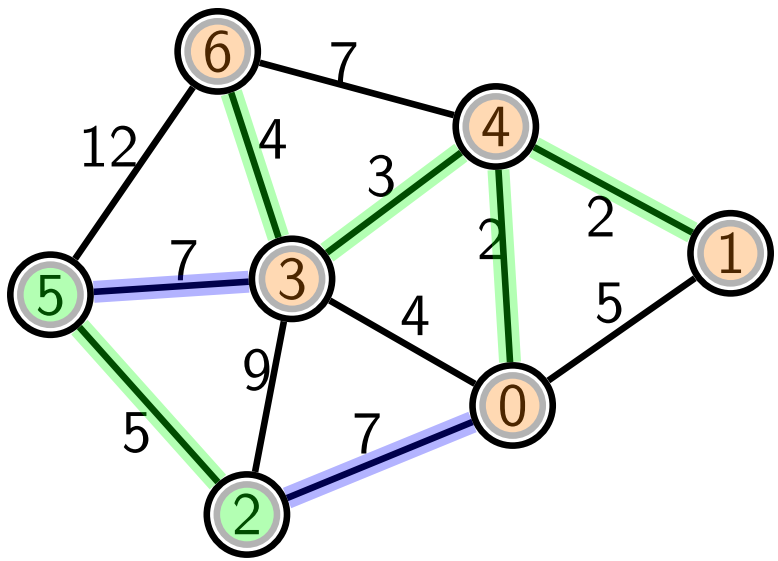
Déroulement de l'algorithme de Kruskal

Exemple 5



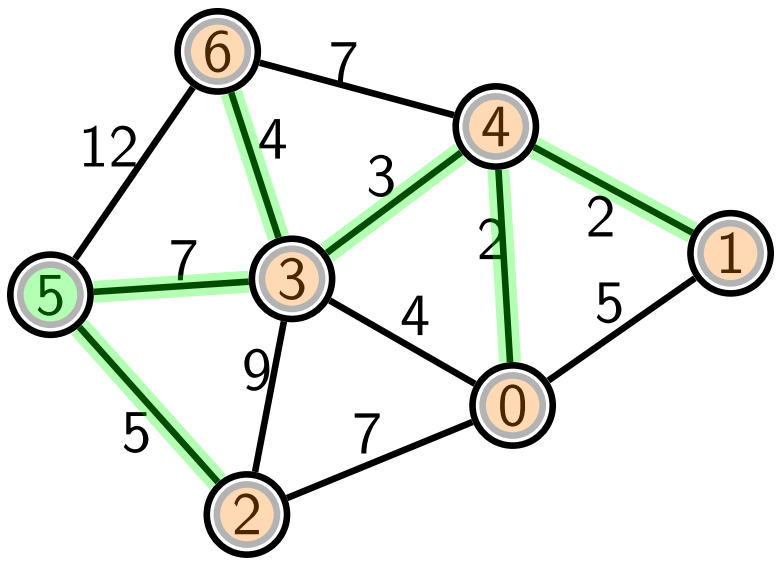
Déroulement de l'algorithme de Kruskal

Exemple 5



Déroulement de l'algorithme de Kruskal

Exemple 5



Kruskal : validité des arêtes

- Étant donnée une arête $e = \{u, v\}$ de G : comment sait-on si elle est valide par rapport à la forêt F ?

Kruskal : validité des arêtes

- Étant donnée une arête $e = \{u, v\}$ de G : comment sait-on si elle est valide par rapport à la forêt F ?
- Dans l'algorithme de Prim, il suffisait de vérifier si les deux sommets étaient dans l'**arbre** T ;

Kruskal : validité des arêtes

- Étant donnée une arête $e = \{u, v\}$ de G : comment sait-on si elle est valide par rapport à la forêt F ?
- Dans l'algorithme de Prim, il suffisait de vérifier si les deux sommets étaient dans l'**arbre** T ;
- Mais ici, on construit plusieurs composantes en même temps !

Kruskal : validité des arêtes

- Étant donnée une arête $e = \{u, v\}$ de G : comment sait-on si elle est valide par rapport à la forêt F ?
- Dans l'algorithme de Prim, il suffisait de vérifier si les deux sommets étaient dans l'**arbre** T ;
- Mais ici, on construit plusieurs composantes en même temps !

- Solution : $\{u, v\}$ est valide si et seulement si u et v appartiennent à des composantes différentes de F ;

Kruskal : gestion des composantes

- Comment maintenir les informations sur les composantes ?

Kruskal : gestion des composantes

- Comment maintenir les informations sur les composantes ?
- Deux techniques naïves :

Kruskal : gestion des composantes

- Comment maintenir les informations sur les composantes ?
- Deux techniques naïves :
  1. stocker les classes de la partition sous la forme d'une collection d'ensembles ; dans ce cas :

Kruskal : gestion des composantes

- Comment maintenir les informations sur les composantes ?
- Deux techniques naïves :
  1. stocker les classes de la partition sous la forme d'une collection d'ensembles ; dans ce cas :
     - fusionner les classes A et B se fait en $O(|A| + |B|)$;

Kruskal : gestion des composantes

- Comment maintenir les informations sur les composantes ?
- Deux techniques naïves :
  1. stocker les classes de la partition sous la forme d'une collection d'ensembles ; dans ce cas :
     - fusionner les classes A et B se fait en $O(|A| + |B|)$;
     - identifier la classe du sommet v se fait en $O(|V|)$;

Kruskal : gestion des composantes

- Comment maintenir les informations sur les composantes ?
- Deux techniques naïves :
  1. stocker les classes de la partition sous la forme d'une collection d'ensembles ; dans ce cas :
     - fusionner les classes A et B se fait en $O(|A| + |B|)$;
     - identifier la classe du sommet v se fait en $O(|V|)$;
  2. utiliser un tableau `marqueurs` dont la case `marqueurs[`v`]` contient la classe à laquelle appartient v ;

Kruskal : gestion des composantes

- Comment maintenir les informations sur les composantes ?
- Deux techniques naïves :
  1. stocker les classes de la partition sous la forme d'une collection d'ensembles ; dans ce cas :
     - fusionner les classes A et B se fait en $O(|A| + |B|)$;
     - identifier la classe du sommet v se fait en $O(|V|)$;
  2. utiliser un tableau `marqueurs` dont la case `marqueurs[`v`]` contient la classe à laquelle appartient v ;
     - fusionner les classes A et B se fait en $O(|V|)$;

Kruskal : gestion des composantes

- Comment maintenir les informations sur les composantes ?
- Deux techniques naïves :
  1. stocker les classes de la partition sous la forme d'une collection d'ensembles ; dans ce cas :
     - fusionner les classes A et B se fait en $O(|A| + |B|)$;
     - identifier la classe du sommet v se fait en $O(|V|)$;
  2. utiliser un tableau `marqueurs` dont la case `marqueurs[`v`]` contient la classe à laquelle appartient v ;
     - fusionner les classes A et B se fait en $O(|V|)$;
     - identifier la classe du sommet v se fait en $O(1)$;

Kruskal : gestion des composantes

- Comment maintenir les informations sur les composantes ?
- Deux techniques naïves :
  1. stocker les classes de la partition sous la forme d'une collection d'ensembles ; dans ce cas :
     - fusionner les classes A et B se fait en $O(|A| + |B|)$;
     - identifier la classe du sommet v se fait en $O(|V|)$;
  2. utiliser un tableau `marqueurs` dont la case `marqueurs[`v`]` contient la classe à laquelle appartient v ;
     - fusionner les classes A et B se fait en $O(|V|)$;
     - identifier la classe du sommet v se fait en $O(1)$;
- La structure recommandée dans ce cas-ci est *Union-Find* ;

Structure *Union-Find* (ou ensembles disjoints)

- La structure **Union-Find** représente une partition d'un ensemble à l'aide d'une forêt orientée ;

Structure *Union-Find* (ou ensembles disjoints)

- La structure **Union-Find** représente une partition d'un ensemble à l'aide d'une forêt orientée ;
  1. chaque arbre représente une classe de la partition ;

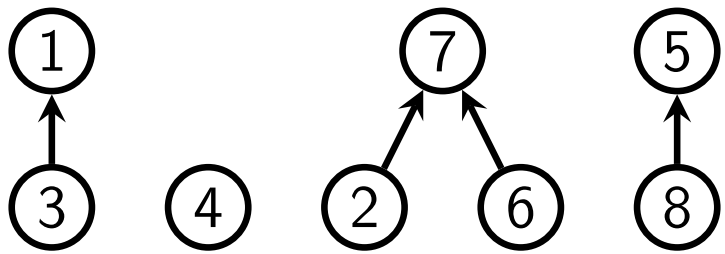
Structure *Union-Find* (ou ensembles disjoints)

- La structure **Union-Find** représente une partition d'un ensemble à l'aide d'une forêt orientée ;
  1. chaque arbre représente une classe de la partition ;
  2. le "numéro" d'une partie est le numéro de sa racine ;

Structure *Union-Find* (ou ensembles disjoints)

- La structure **Union-Find** représente une partition d'un ensemble à l'aide d'une forêt orientée ;
  1. chaque arbre représente une classe de la partition ;
  2. le "numéro" d'une partie est le numéro de sa racine ;

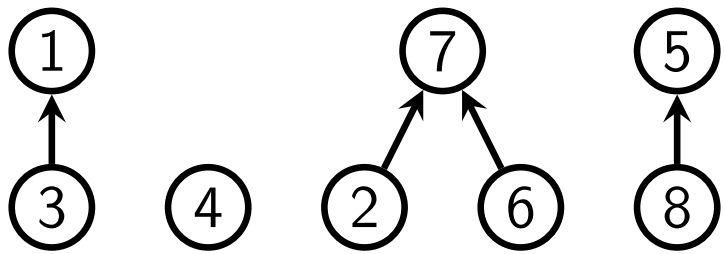
Exemple 6 (représentation de $\{\{1,3\},\{2,6,7\},\{4\},\{5,8\}\}$)



Structure *Union-Find* (ou ensembles disjoints)

- La structure **Union-Find** représente une partition d'un ensemble à l'aide d'une forêt orientée ;
  1. chaque arbre représente une classe de la partition ;
  2. le "numéro" d'une partie est le numéro de sa racine ;

Exemple 6 (représentation de $\{\{1,3\},\{2,6,7\},\{4\},\{5,8\}\}$)

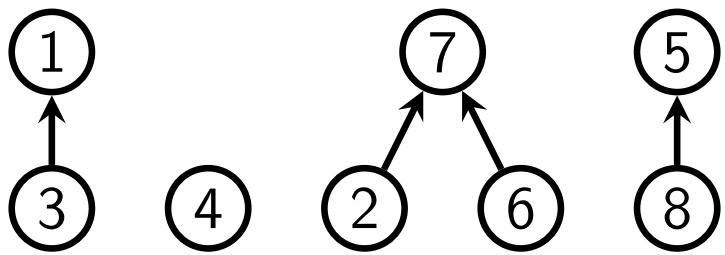


- Les deux opérations disponibles sont :

Structure *Union-Find* (ou ensembles disjoints)

- La structure **Union-Find** représente une partition d'un ensemble à l'aide d'une forêt orientée ;
  1. chaque arbre représente une classe de la partition ;
  2. le "numéro" d'une partie est le numéro de sa racine ;

Exemple 6 (représentation de $\{\{1,3\},\{2,6,7\},\{4\},\{5,8\}\}$)

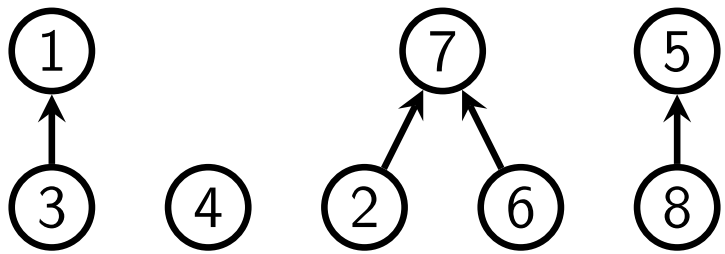


- Les deux opérations disponibles sont :
  1. *union(A, B)* : fusionne les classes A et B ;

Structure *Union-Find* (ou ensembles disjoints)

- La structure **Union-Find** représente une partition d'un ensemble à l'aide d'une forêt orientée ;
  1. chaque arbre représente une classe de la partition ;
  2. le "numéro" d'une partie est le numéro de sa racine ;

Exemple 6 (représentation de $\{\{1,3\},\{2,6,7\},\{4\},\{5,8\}\}$)



- Les deux opérations disponibles sont :
  1. *union(A, B)* : fusionne les classes A et B ;
  2. *find(x)* : renvoie la classe de l'élément x ;

Structure *Union-Find* : problèmes potentiels

- De mauvaises fusions peuvent mener mener à une structure dégénérée ;
- On se retrouve alors avec une opération *find* en $O(n)$;
- On va avoir recours à deux optimisations pour éviter les problèmes :
  1. la compression de chemins ;
  2. l'utilisation de rangs ;

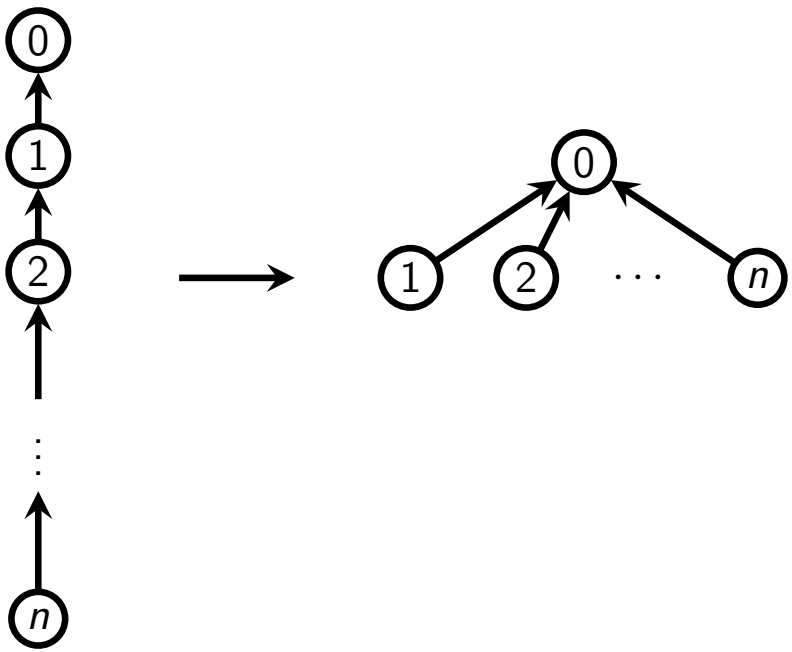
Exemple 7

Optimisation de *Union-Find* : compression de chemin

Les informations calculées lors de l'appel à *find* permettent de réduire la hauteur de la structure.

Exemple 8 (appel à *find(n)*)

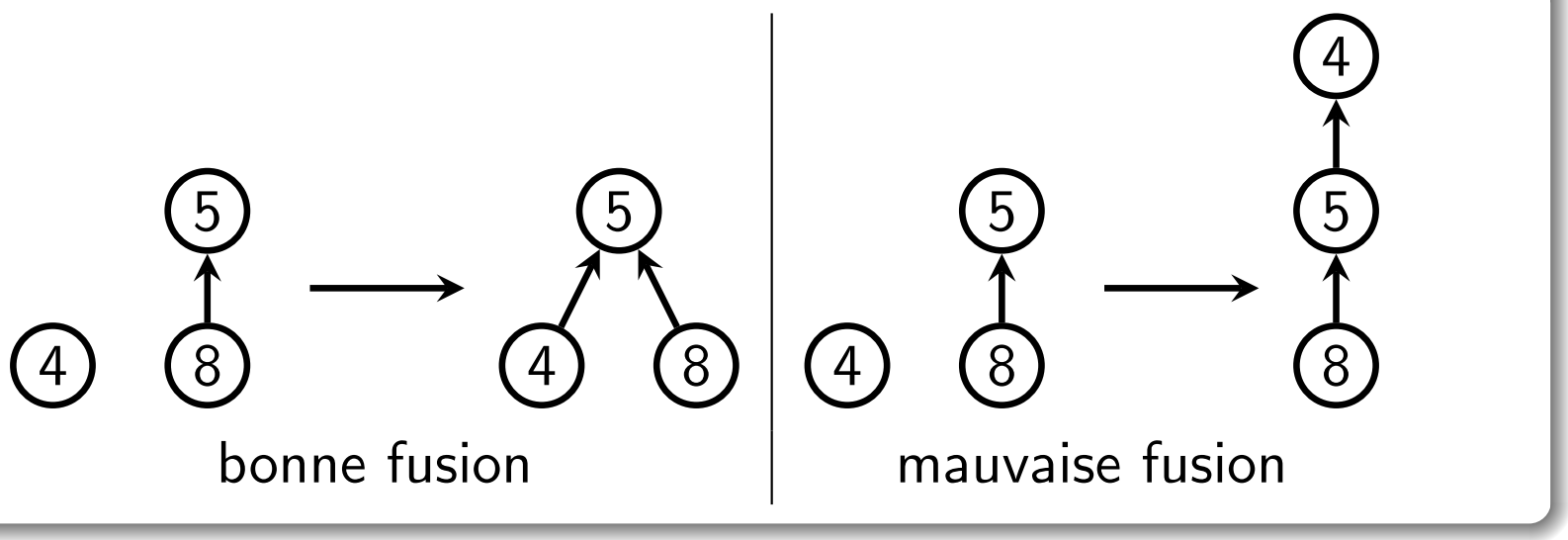
find donne 0 pour tous les sommets du chemin de n à 0.



Optimisation de *Union-Find* : rangs

Les "mauvaises" fusions augmentent la hauteur de l'arbre et donc la complexité de *find*.

Exemple 9



Rangs et fusions

- Calculer explicitement la hauteur des arbres est trop coûteux en temps ;

Rangs et fusions

- Calculer explicitement la hauteur des arbres est trop coûteux en temps ;
- On stocke plutôt pour chaque arbre un champ `rang` initialement nul ;

Rangs et fusions

- Calculer explicitement la hauteur des arbres est trop coûteux en temps ;
- On stocke plutôt pour chaque arbre un champ `rang` initialement nul ;
- Quand on doit effectuer une fusion de deux arbres A et B de rangs r et s :

Rangs et fusions

- Calculer explicitement la hauteur des arbres est trop coûteux en temps ;
- On stocke plutôt pour chaque arbre un champ `rang` initialement nul ;
- Quand on doit effectuer une fusion de deux arbres A et B de rangs r et s :
  - si $r < s$, B devient le parent de A ;

Rangs et fusions

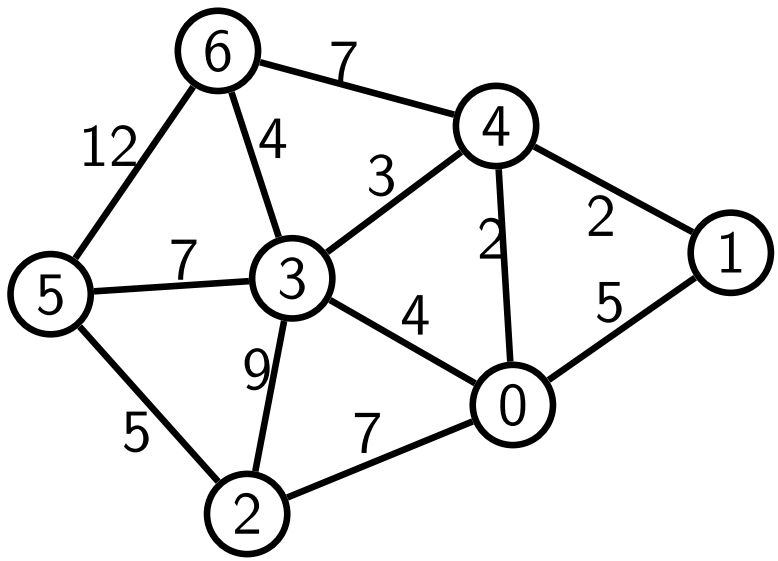
- Calculer explicitement la hauteur des arbres est trop coûteux en temps ;
- On stocke plutôt pour chaque arbre un champ `rang` initialement nul ;
- Quand on doit effectuer une fusion de deux arbres A et B de rangs r et s :
  - si $r < s$, B devient le parent de A ;
  - si $r > s$, A devient le parent de B ;

Rangs et fusions

- Calculer explicitement la hauteur des arbres est trop coûteux en temps ;
- On stocke plutôt pour chaque arbre un champ `rang` initialement nul ;
- Quand on doit effectuer une fusion de deux arbres A et B de rangs r et s :
  - si $r < s$, B devient le parent de A ;
  - si $r > s$, A devient le parent de B ;
  - si $r = s$: choisir arbitrairement, et incrémenter le rang de l'arbre sous lequel on place l'autre ;

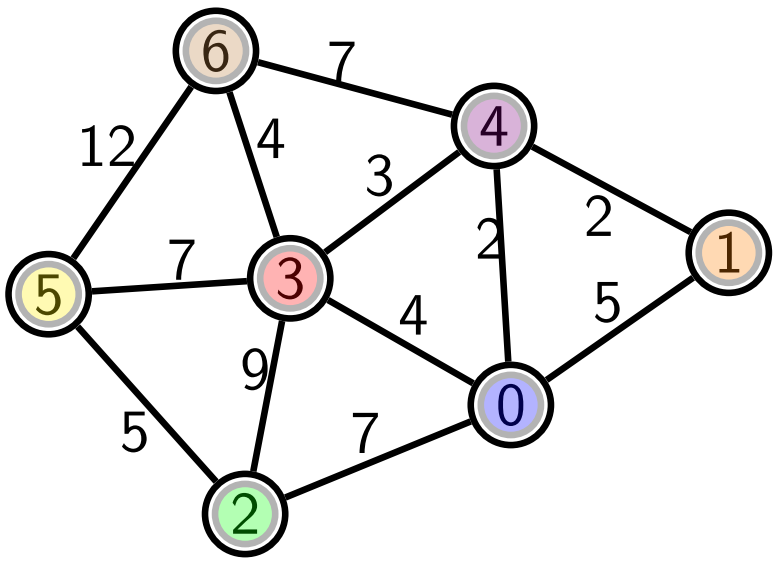
Déroulement de l'algorithme de Kruskal

Exemple 10



Déroulement de l'algorithme de Kruskal

Exemple 10



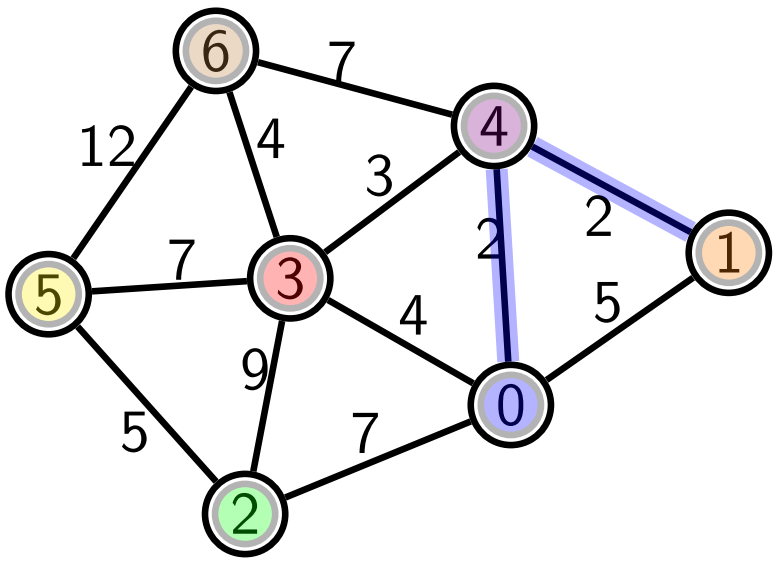
Les coulisses

parents :

0	1	2	3	4	5	6

Déroulement de l'algorithme de Kruskal

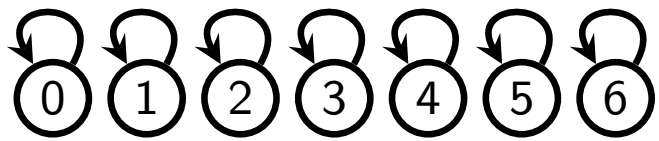
Exemple 10



Les coulisses

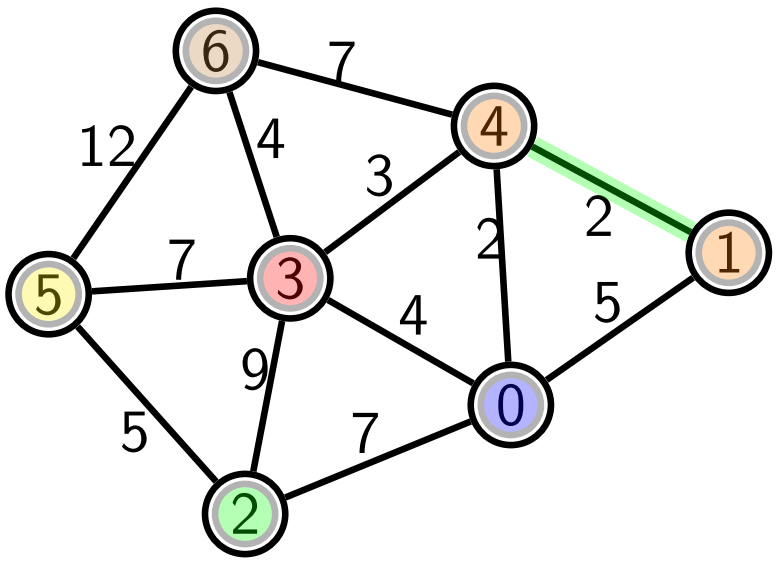
parents :

0	1	2	3	4	5	6
0	1	2	3	4	5	6

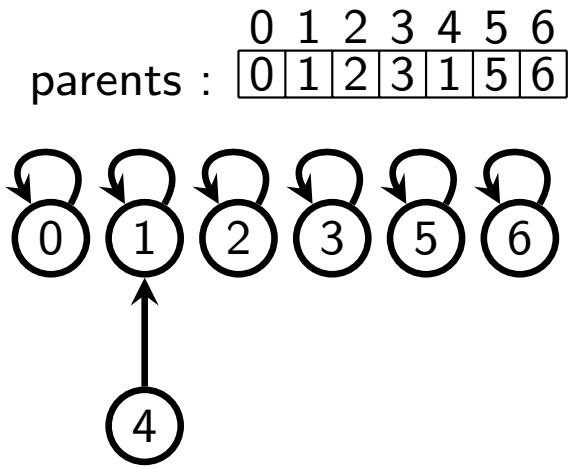


Déroulement de l'algorithme de Kruskal

Exemple 10

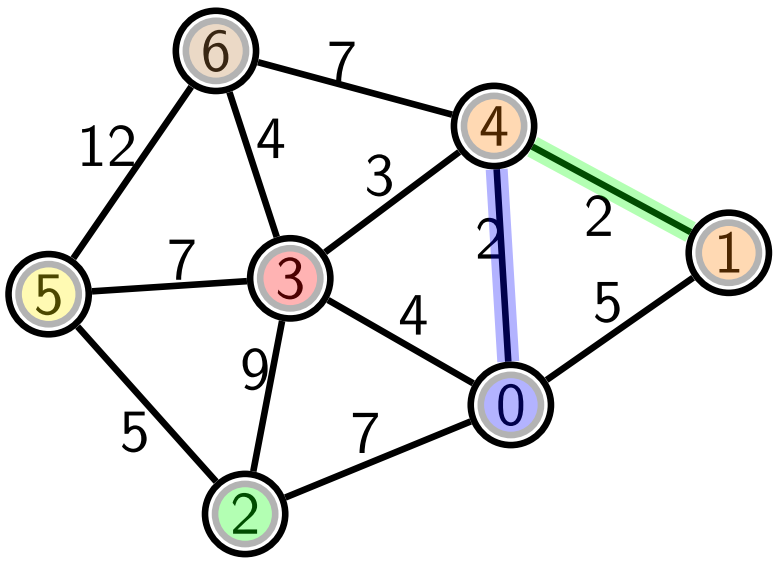


Les coulisses

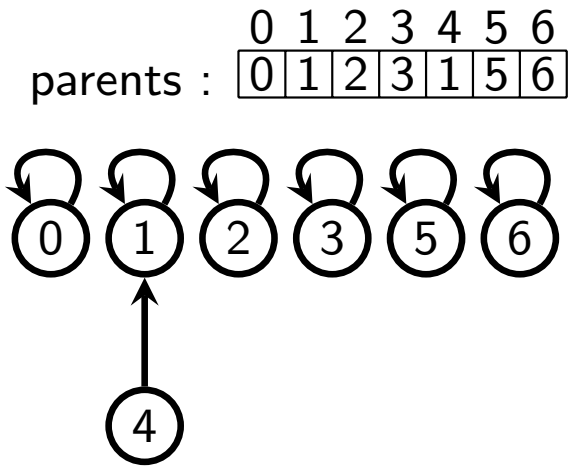


Déroulement de l'algorithme de Kruskal

Exemple 10

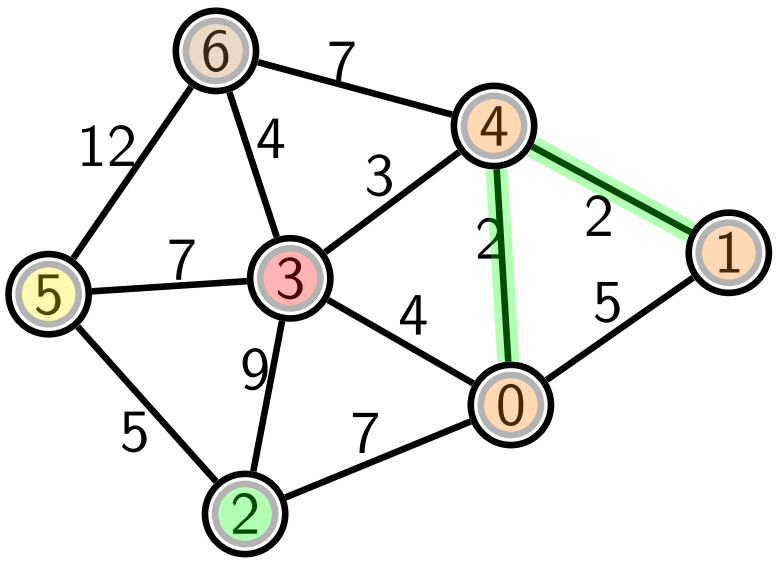


Les coulisses

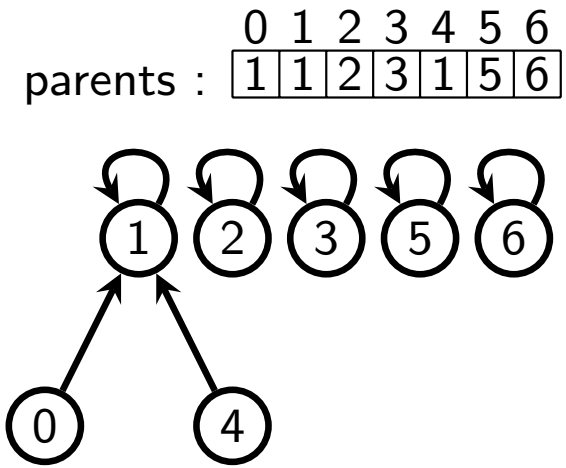


Déroulement de l'algorithme de Kruskal

Exemple 10

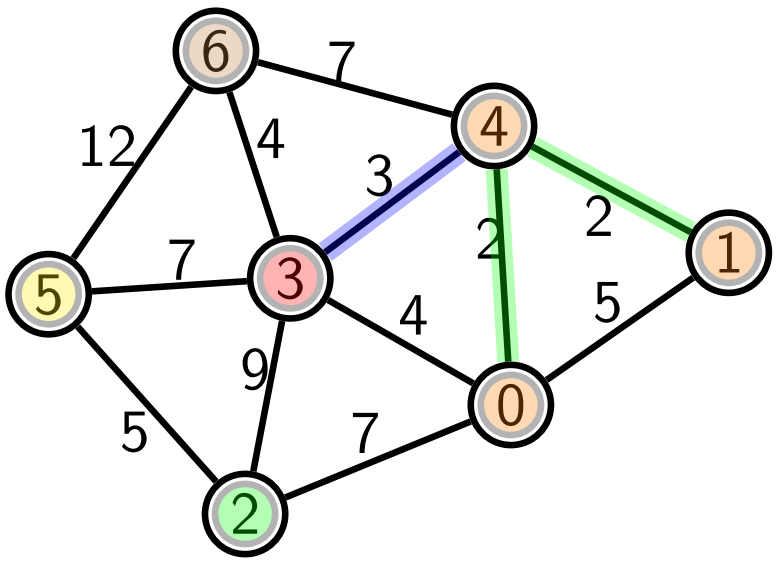


Les coulisses

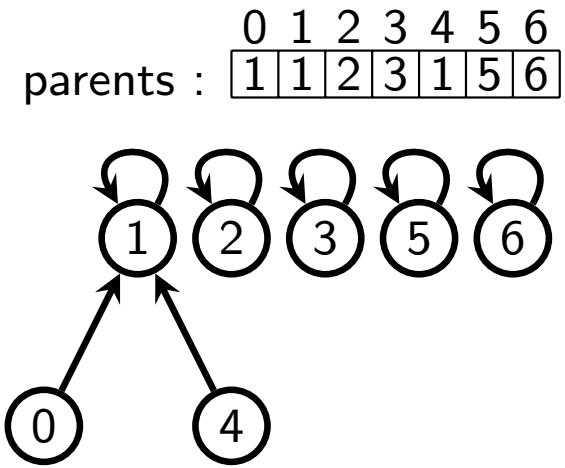


Déroulement de l'algorithme de Kruskal

Exemple 10

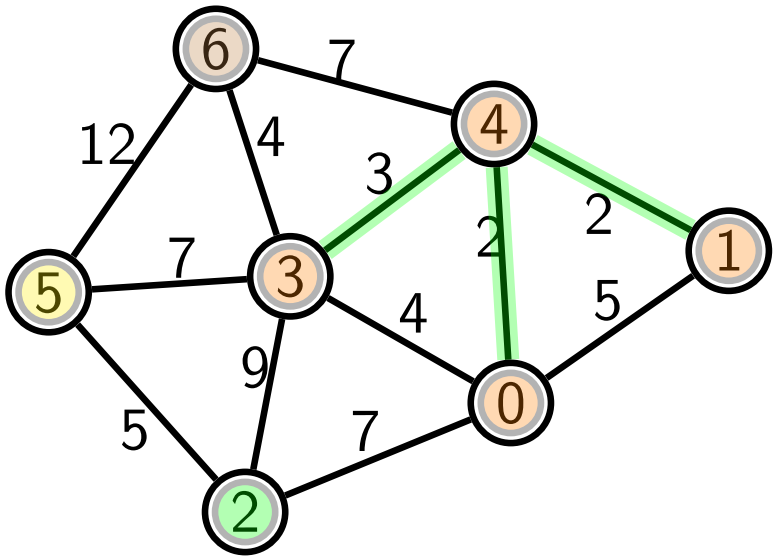


Les coulisses

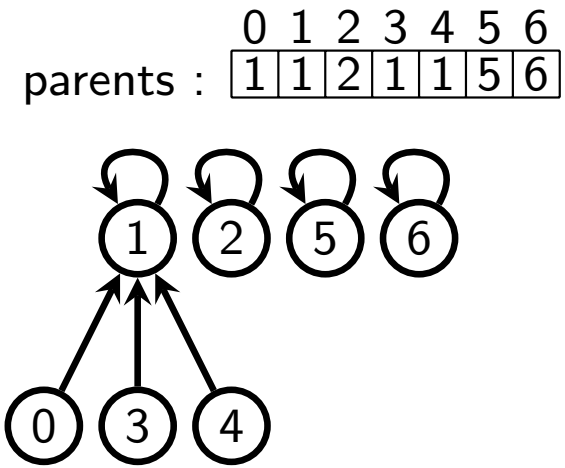


Déroulement de l'algorithme de Kruskal

Exemple 10

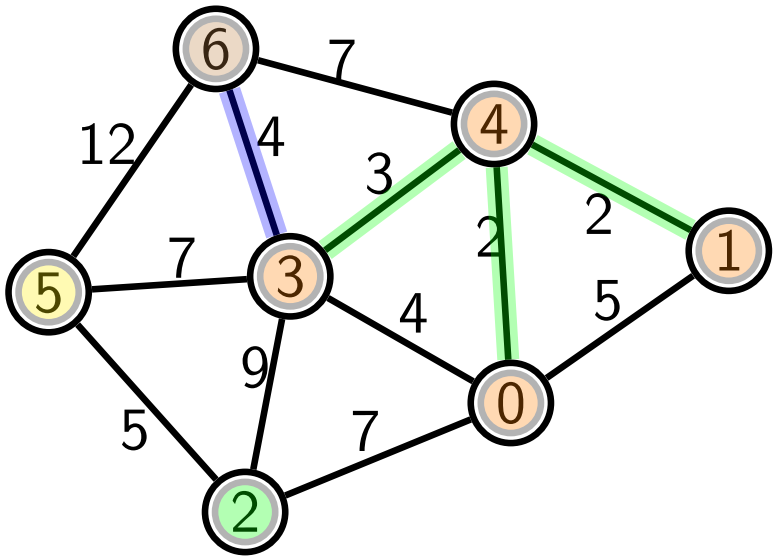


Les coulisses

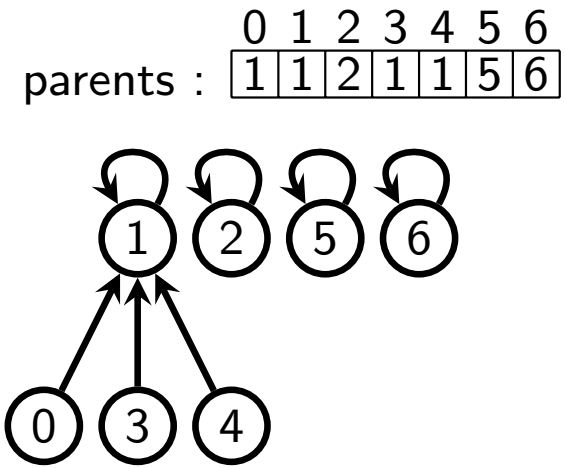


Déroulement de l'algorithme de Kruskal

Exemple 10

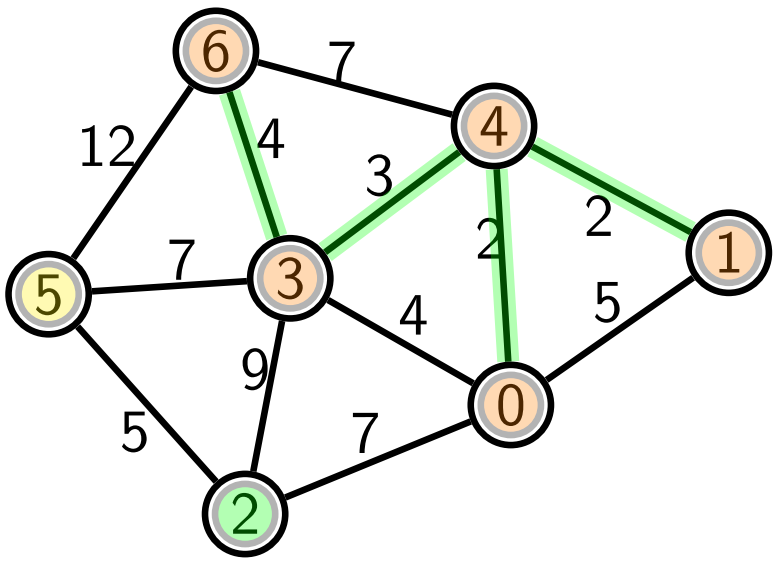


Les coulisses

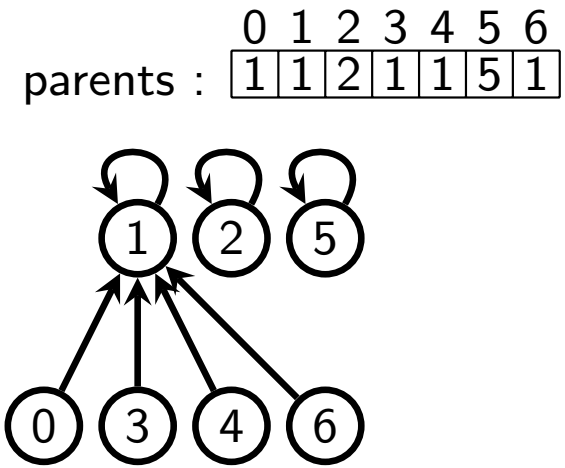


Déroulement de l'algorithme de Kruskal

Exemple 10

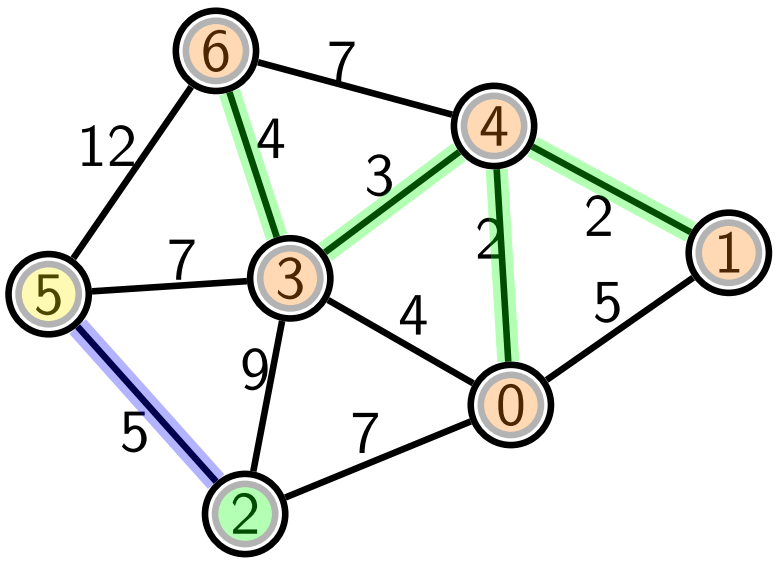


Les coulisses

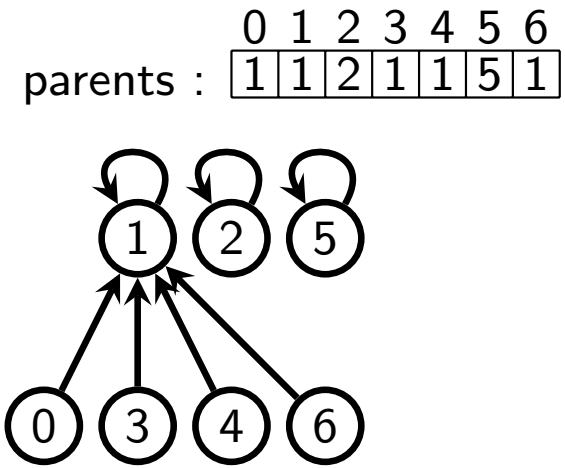


Déroulement de l'algorithme de Kruskal

Exemple 10

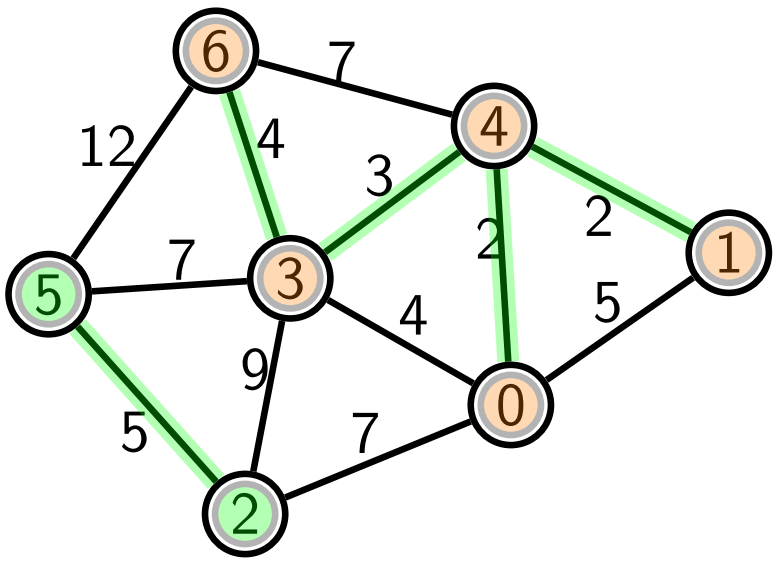


Les coulisses



Déroulement de l'algorithme de Kruskal

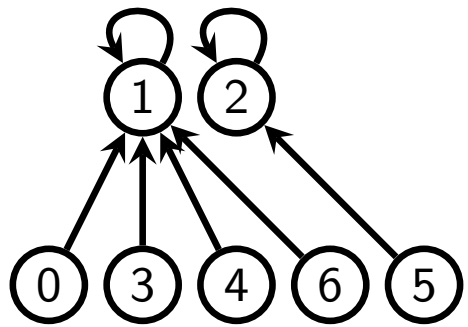
Exemple 10



Les coulisses

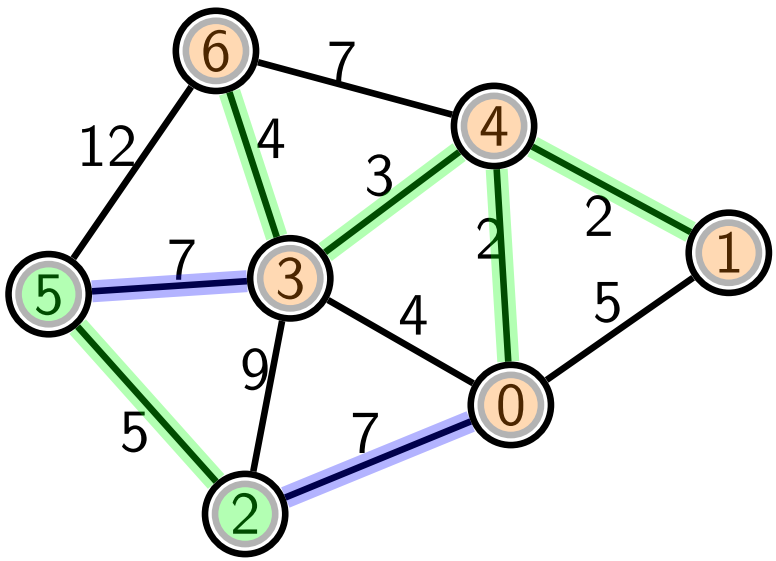
parents :

0	1	2	3	4	5	6
1	1	2	1	1	2	1



Déroulement de l'algorithme de Kruskal

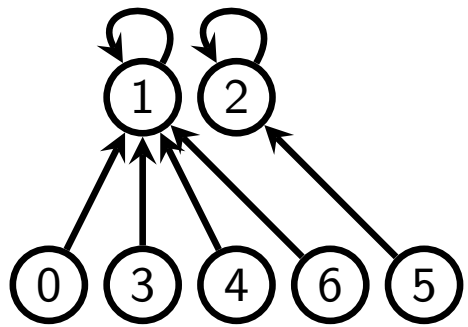
Exemple 10



Les coulisses

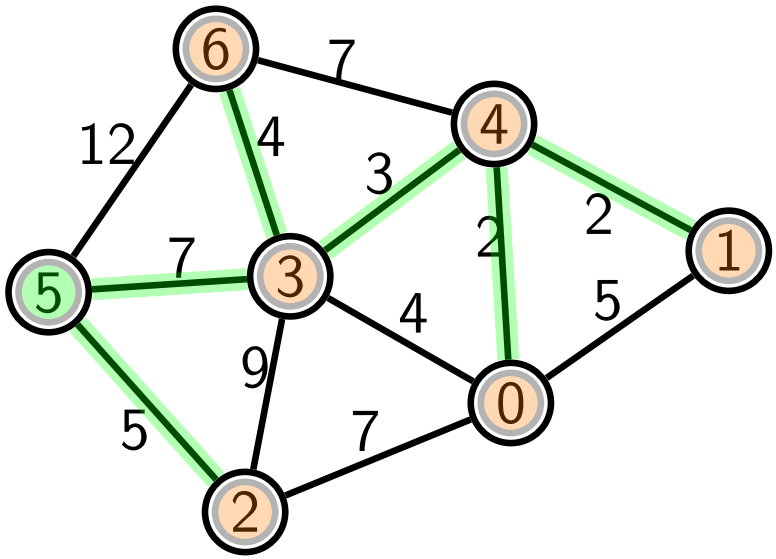
parents :

0	1	2	3	4	5	6
1	1	2	1	1	2	1



Déroulement de l'algorithme de Kruskal

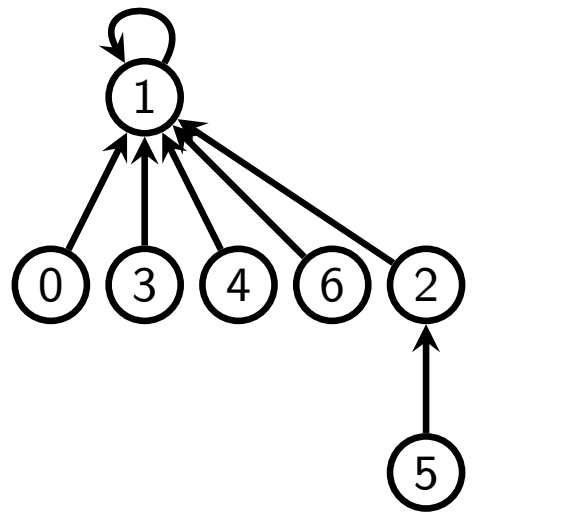
Exemple 10



Les coulisses

parents :

0	1	2	3	4	5	6
1	1	1	1	1	2	1



L'algorithme de Kruskal proprement dit

Algorithme 4 : KRUSKAL(G)

Entrées : un graphe pondéré non orienté G.
Sortie : une forêt couvrante de poids minimum pour G consistant en un arbre couvrant de poids minimum pour chaque composante connexe de G.

```
1 forêt ← GraphePondéré(G.sommets());
2 classes ← UnionFind(G.sommets());
3 pour chaque (u, v, p) ∈ tri_par_poids_croissant(G.arêtes()) faire
4     si classes.find(u) ≠ classes.find(v) alors
5         forêt.ajouter_arête(u, v, p);
6         classes.union(classes.find(u), classes.find(v));
7 renvoyer forêt;
```

Complexité de l'algorithme de Kruskal (listes d'adjacence)

- Initialisation de la forêt : $O(|V|)$;

Complexité de l'algorithme de Kruskal (listes d'adjacence)

- Initialisation de la forêt : $O(|V|)$;
- Initialisation de Union-Find : $O(|V|)$;

Complexité de l'algorithme de Kruskal (listes d'adjacence)

- Initialisation de la forêt : $O(|V|)$;
- Initialisation de Union-Find : $O(|V|)$;
- Tri des arêtes : $O(|E| \log |E|) = O(|E| \log |V|)$;

Complexité de l'algorithme de Kruskal (listes d'adjacence)

- Initialisation de la forêt : $O(|V|)$;
- Initialisation de Union-Find : $O(|V|)$;
- Tri des arêtes : $O(|E| \log |E|) = O(|E| \log |V|)$;
- Parcours des arêtes : $O(|E|)$) ;

Complexité de l'algorithme de Kruskal (listes d'adjacence)

- Initialisation de la forêt : $O(|V|)$;
- Initialisation de Union-Find : $O(|V|)$;
- Tri des arêtes : $O(|E| \log |E|) = O(|E| \log |V|)$;
- Parcours des arêtes : $O(|E|)$) ;
- Opérations sur Union-Find : "à peu près $O(1)$" ;

Complexité de l'algorithme de Kruskal (listes d'adjacence)

- Initialisation de la forêt : $O(|V|)$;
- Initialisation de Union-Find : $O(|V|)$;
- Tri des arêtes : $O(|E| \log |E|) = O(|E| \log |V|)$;
- Parcours des arêtes : $O(|E|)$;
- Opérations sur Union-Find : "à peu près $O(1)$" ;

$$\Rightarrow O(|V| + |E| + |E| \log |V|) = O(|E| \log |V|) ;$$

Bibliographie

[1] Michael L. Fredman and Robert Endre Tarjan.

Fibonacci heaps and their uses in improved network optimization algorithms.

Journal of the ACM, 34(3) :596–615, 1987.