Université Paris Cité

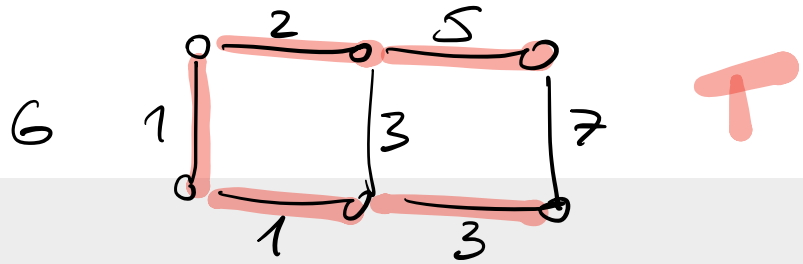
Arbres couvrants de poids minimum

CM nº6 — Algorithmique (AL5)

Matěj Stehlík

27/10/2023

Arbre couvrant de poids minimum (Minimum Spanning Tree)



Définition

Soit $G = (V, E)$ un graphe connexe avec une pondération $w \in \mathbb{R}^{|E|}$. Un *arbre couvrant de poids minimum* de G est un arbre couvrant $T \subseteq G$ qui minimise $w(T) = \sum_{e \in E} w_e$.

Exemple

Réalisation d'un réseau électrique ou informatique entre différents points, deux points quelconques doivent toujours être reliés entre eux (connexité) et on doit minimiser le coût de la réalisation.

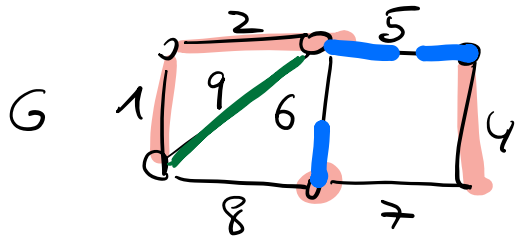
Remarque sur les poids des arêtes

- Pour rester simple, on supposera que tous les poids des arêtes sont *distincts* : $w_e \neq w_{e'}$ pour toute paire d'arêtes e et e'.
- Les poids distincts garantissent que l'arbre couvrant de poids minimum est unique.
- Sans cette condition, il peut y avoir plusieurs arbres couvrants de poids minimum différents.
- Si on dispose d'un algorithme qui suppose que les poids des arêtes sont distincts, on peut toujours l'utiliser sur des graphes où plusieurs arêtes ont le même poids, tant que nous disposons d'une méthode cohérente pour *casser les égalités*.

Arbre couvrant de poids minimum : un algorithme générique

- Soit G un graphe dont on veut calculer un arbre couvrant de poids minimum.
- Soit $F \subseteq G$ la forêt consistant de n sommets isolés.
- Tant que $F \subseteq G$ n'est pas un arbre, fusionner les arbres en ajoutant certaines arêtes entre eux.
- L'algorithme s'arrête lorsque F consiste en un seul arbre à n sommets, qui doit être l'arbre couvrant de poids minimum.
- Les règles de sélection des « certaines » arêtes conduisent à des algorithmes différents.

Arêtes sûres et arêtes inutiles



- Soit F une forêt couvrante « intermédiaire » de G.
- À chaque composante connexe de F, on associe une arête *sûre* : l'arête de poids minimum avec exactement une extrémité dans cette composante.
- Une arête est *inutile* si elle n'est pas une arête de F, mais ses deux extrémités sont dans la même composante de F.

Lemme

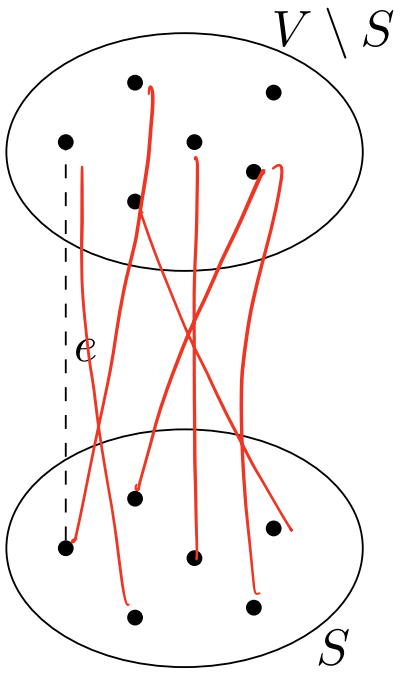
L'arbre couvrant de poids minimum de G contient toutes les arêtes sûres.

Lemme

L'arbre couvrant de poids minimum de G ne contient aucune arête inutile.

Preuve : Ajouter une arête inutile à F crée un cycle.

Démonstration du premier lemme (1/2)



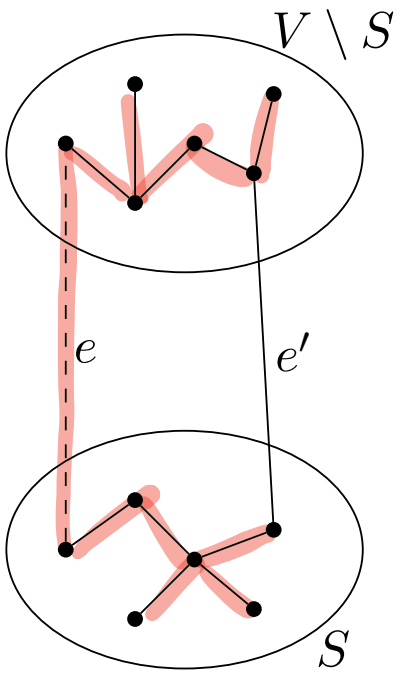
- On prouvera l'affirmation plus forte suivante :

Lemme

Pour tout sous-ensemble $S \subseteq V(G)$, l'arbre couvrant de poids minimum de G contient l'arête de poids minimum avec exactement une extrémité dans S.

- Soit S un sous-ensemble arbitraire de sommets de G.
- Soit e l'arête la plus légère avec exactement une extrémité dans S.
- Supposons par l'absurde qu'il existe un arbre couvrant de poids minimum T qui ne contient pas e.

Démonstration du premier lemme (2/2)



- Puisque T est connexe, il contient une chaîne d'une extrémité de e à l'autre.
- Comme cette chaîne commence à un sommet de S et se termine à un sommet qui n'est pas dans S, il existe au moins une arête $e' \in E(T)$ avec une extrémité dans S et l'autre dans $V \setminus S$.
- $(T - e') \cup \{e\}$ est un arbre couvrant de G.
- Par la définition de e, on a $w_{e'} > w_e$, ce qui implique que $w(T') < w(T)$.
- On conclut que T n'est pas l'arbre couvrant de poids minimum.

Les trois algorithmes d'arbre couvrant de poids minimum

- Voici le comportement, à chaque itération, des trois algorithmes que nous étudierons :

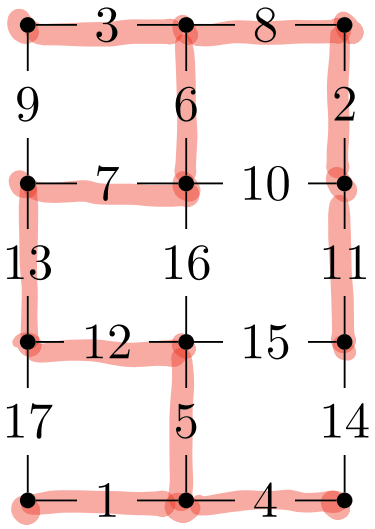
Kruskal : Ajouter l'arête sûre la plus légère à F.

Prim : Ajouter l'arête sûre de T à T.

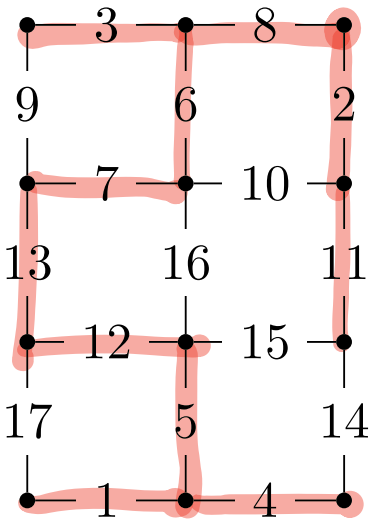
Borůvka : Ajouter toutes les arêtes sûres à F.

- Ces algorithmes sont exemples des algorithmes *gloutons*.

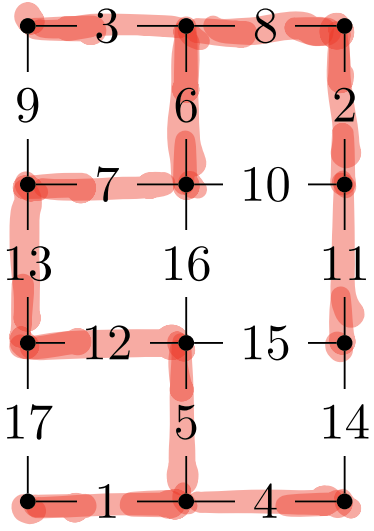
Illustration des trois algorithmes



Kruskal



Prim



Borůvka

Petite parenthèse sur les algorithmes gloutons

- Pour gagner aux échecs, il faut calculer beaucoup de coups à l'avance.
- Un joueur qui calcule seulement un coup à l'avance n'aura pas beaucoup de succès.
- Pourtant, dans certains problèmes, cette stratégie myope peut conduire à des bons algorithmes, comme c'est le cas pour les arbres couvrants de poids minimum.
- Les *algorithmes gloutons* construisent une solution pièce par pièce, ajoutant à chaque étape la pièce qui donne les bénéfices les plus importants.

L'algorithme de Kruskal (vue d'ensemble)

Entrées : Un graphe connexe
$G = (V, E)$ avec des poids w_e
sur les arêtes

Sorties : Ensemble d'arêtes $X \subseteq E$
d'un arbre couvrant de G

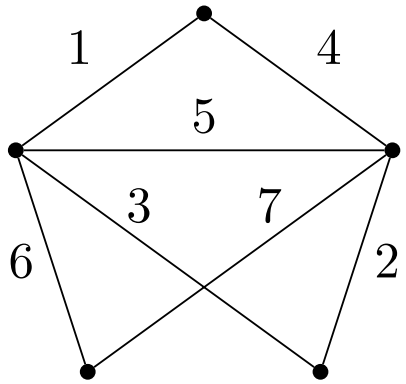
$X \leftarrow \varnothing$

Trier les arêtes E par poids croissant

pour tous les $e \in E$ **faire**

  si $(V, X \cup \{e\})$ *est acyclique* **alors**

    $X \leftarrow X \cup \{e\}$



L'algorithme de Kruskal (vue d'ensemble)

Entrées : Un graphe connexe
$G = (V, E)$ avec des poids w_e
sur les arêtes

Sorties : Ensemble d'arêtes $X \subseteq E$
d'un arbre couvrant de G

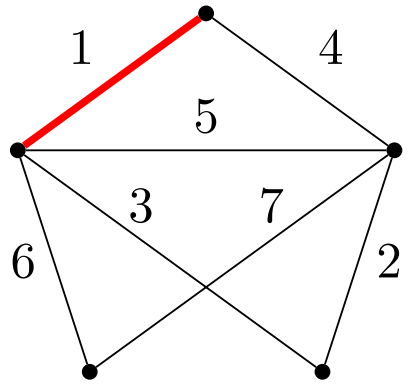
$X \leftarrow \varnothing$

Trier les arêtes E par poids croissant

pour tous les $e \in E$ **faire**

  si $(V, X \cup \{e\})$ *est acyclique* **alors**

    $X \leftarrow X \cup \{e\}$



L'algorithme de Kruskal (vue d'ensemble)

Entrées : Un graphe connexe
$G = (V, E)$ avec des poids w_e
sur les arêtes

Sorties : Ensemble d'arêtes $X \subseteq E$
d'un arbre couvrant de G

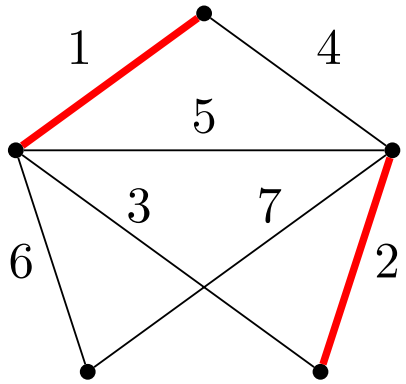
$X \leftarrow \varnothing$

Trier les arêtes E par poids croissant

pour tous les $e \in E$ **faire**

  si $(V, X \cup \{e\})$ *est acyclique* **alors**

    $X \leftarrow X \cup \{e\}$



L'algorithme de Kruskal (vue d'ensemble)

Entrées : Un graphe connexe
$G = (V, E)$ avec des poids w_e
sur les arêtes

Sorties : Ensemble d'arêtes $X \subseteq E$
d'un arbre couvrant de G

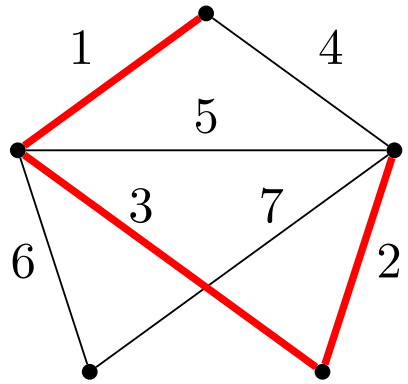
$X \leftarrow \varnothing$

Trier les arêtes E par poids croissant

pour tous les $e \in E$ **faire**

  si $(V, X \cup \{e\})$ *est acyclique* **alors**

    $X \leftarrow X \cup \{e\}$



L'algorithme de Kruskal (vue d'ensemble)

Entrées : Un graphe connexe
$G = (V, E)$ avec des poids w_e
sur les arêtes

Sorties : Ensemble d'arêtes $X \subseteq E$
d'un arbre couvrant de G

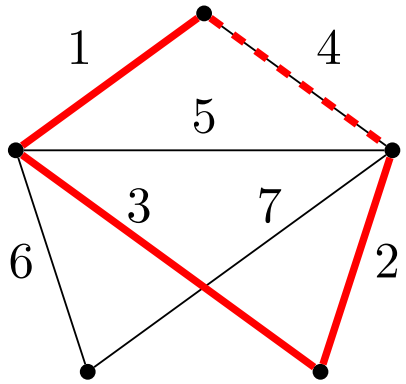
$X \leftarrow \varnothing$

Trier les arêtes E par poids croissant

pour tous les $e \in E$ **faire**

  si $(V, X \cup \{e\})$ *est acyclique* **alors**

    $X \leftarrow X \cup \{e\}$



L'algorithme de Kruskal (vue d'ensemble)

Entrées : Un graphe connexe
$G = (V, E)$ avec des poids w_e
sur les arêtes

Sorties : Ensemble d'arêtes $X \subseteq E$
d'un arbre couvrant de G

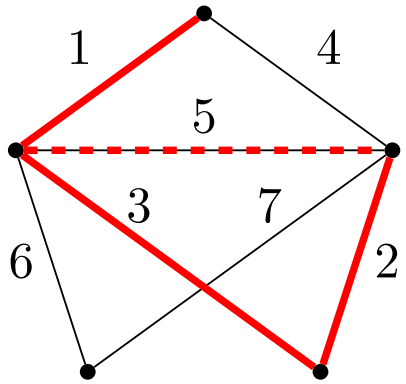
$X \leftarrow \varnothing$

Trier les arêtes E par poids croissant

pour tous les $e \in E$ **faire**

  si $(V, X \cup \{e\})$ *est acyclique* **alors**

    $X \leftarrow X \cup \{e\}$



L'algorithme de Kruskal (vue d'ensemble)

Entrées : Un graphe connexe
$G = (V, E)$ avec des poids w_e
sur les arêtes

Sorties : Ensemble d'arêtes $X \subseteq E$
d'un arbre couvrant de G

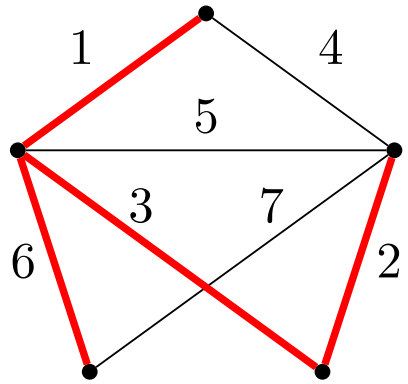
$X \leftarrow \varnothing$

Trier les arêtes E par poids croissant

pour tous les $e \in E$ **faire**

  si $(V, X \cup \{e\})$ *est acyclique* **alors**

    $X \leftarrow X \cup \{e\}$



L'algorithme de Kruskal (vue d'ensemble)

Entrées : Un graphe connexe
$G = (V, E)$ avec des poids w_e
sur les arêtes

Sorties : Ensemble d'arêtes $X \subseteq E$
d'un arbre couvrant de G

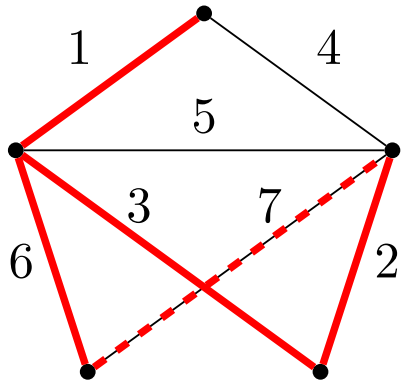
$X \leftarrow \varnothing$

Trier les arêtes E par poids croissant

pour tous les $e \in E$ **faire**

  si $(V, X \cup \{e\})$ *est acyclique* **alors**

    $X \leftarrow X \cup \{e\}$



L'algorithme de Kruskal (vue d'ensemble)

Entrées : Un graphe connexe
$G = (V, E)$ avec des poids w_e
sur les arêtes

Sorties : Ensemble d'arêtes $X \subseteq E$
d'un arbre couvrant de G

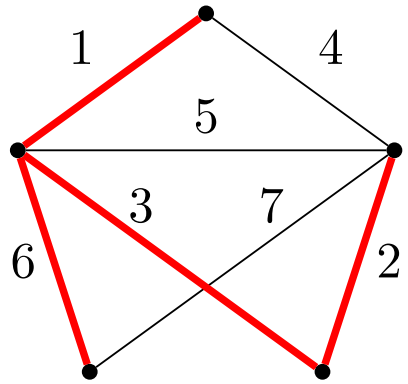
$X \leftarrow \varnothing$

Trier les arêtes E par poids croissant

pour tous les $e \in E$ **faire**

  si $(V, X \cup \{e\})$ *est acyclique* **alors**

    $X \leftarrow X \cup \{e\}$



Justification de l'algorithme de Kruskal

- Puisque nous examinons les arêtes de la plus légère à la plus lourde, dans cet ordre, toute arête que nous examinons est sûre si et seulement si ses points d'extrémité se trouvent dans des composants différents de la forêt F.
- Supposons par l'absurde que e relie deux composantes connexes F_1 et F_2, mais e n'est pas sûre.
- Il existe alors une arête e' plus légère avec exactement une extrémité dans F_1.
- Cela est impossible, car e' aurait été examiné (et donc ajouté à F) avant e.

Complexité de Kruskal

Question

Comment décider si l'ajout d'une arête crée un cycle ? Est-il possible de le faire en temps constant ?

Version naïve

Faire un parcours en largeur ou profondeur pour détecter le cycle — coût $O(n+m)$ à chacun des appels, c'est-à-dire, coût $O(nm+m^2)$, potentiellement $O(n^4)$.

Une astuce pour détecter des cycles

- Nous allons modéliser l'état de l'algorithme par une collection d'ensembles *disjoints*.
- Chaque ensemble correspond aux sommets d'une composante connexe.
- Au début chaque sommet est isolé, c'est-à-dire, chaque sommet est une composante connexe.
- makeset(x) : créer l'ensemble $\{x\}$.
- Nous aurons besoin de vérifier si deux sommets sont dans la même composante connexe.
- find(x) : à quel ensemble appartient x ?
- Lorsque nous rajoutons une arête, nous fusionnons deux composantes connexes.
- union(x, y) : fusionner les deux ensembles qui contiennent x et y.

L'algorithme de Kruskal (version "union–find")

Entrées : Un graphe connexe $G = (V, E)$ avec des poids w_e sur les arêtes

Sorties : Ensemble d'arêtes $X \subseteq E$ d'un arbre couvrant de G

pour tous les $u \in V$ **faire**
  makeset(u)

$X \leftarrow \varnothing$

Trier les arêtes E par poids croissant

pour tous les $uv \in E$, *dans l'ordre croissant de poids* **faire**
  si *find*$(u) \neq$ *find*(v) **alors**
    $X \leftarrow X \cup \{uv\}$
    union(u, v)

Remarque

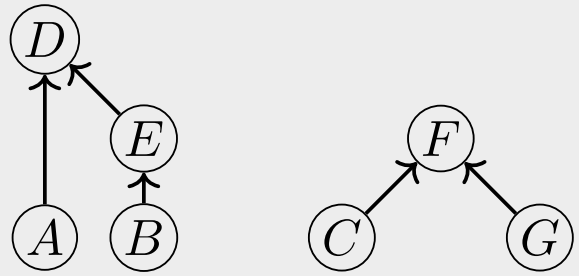
L'algorithme appelle `makeset` $|V|$ fois, `find` $2|E|$ fois, et `union` $|V| - 1$ fois.

Structure des données “union–find”

- Nous pouvons stocker un ensemble S comme une arborescence

Exemple

Une représentation de $\{A, B, D, E\}$ et $\{C, F, G\}$ par des arborescences :



rank(A) = 0
rank(E) = 1
rank(D) = 2

Pointeurs et rank

- Les sommets de cette arborescence sont les éléments de S (dans un ordre quelconque)
- À chaque élément x on associe un pointeur $\pi(x)$ vers son parent.
- En suivant le chemin $(x, \pi(x), \pi(\pi(x)), \ldots)$, on arrive finalement à la racine r.
- On peut considérer r comme le *représentant* de S.
- Cet élément est distingué par le fait que $\pi(r) = r$.
- À chaque sommet on associe aussi un *rank* qui mesure la hauteur du sous-arborescebce dont le sommet est la racine.

Les fonctions makeset et find

Fonction makeset(x)

$\pi(x) \leftarrow x$
$\texttt{rank}(x) \leftarrow 0$

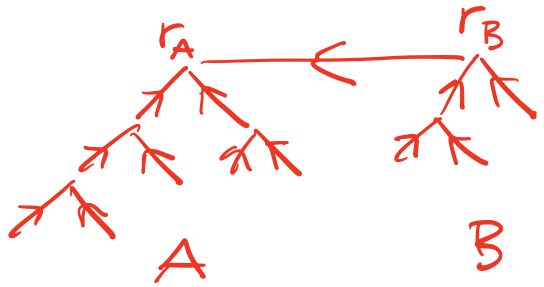
Complexité constante

Fonction find(x)

tant que $x \neq \pi(x)$ **faire**
  $x \leftarrow \pi(x)$
retourner x

Complexité dépend de la hauteur de l'arborescence, donc il est important de limiter la hauteur de l'arborescence.

Fusionner deux ensembles efficacement

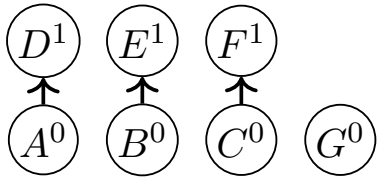


- Soient A et B deux ensembles disjoints qu'on souhaite fusionner.
- Si r_A est la racine de A et r_B est la racine de B, il suffit de définir $\pi(r_A) = r_B$ ou $\pi(r_B) = r_A$.
- Si la hauteur de A est supérieure a celle de B, et on définit $\pi(r_A) = r_B$, alors la hauteur du nouveau arbre augmente de 1.
- Par contre, si on définit $\pi(r_B) = r_A$, alors la hauteur n'augmente pas.
- Avec cette stratégie, la hauteur augmente uniquement lorsque les deux arborescences ont la même hauteur (rank).

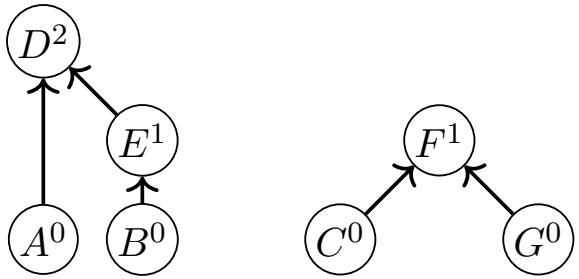
Une illustration

$\text{makeset}(A), \ldots, \text{makeset}(G)$:

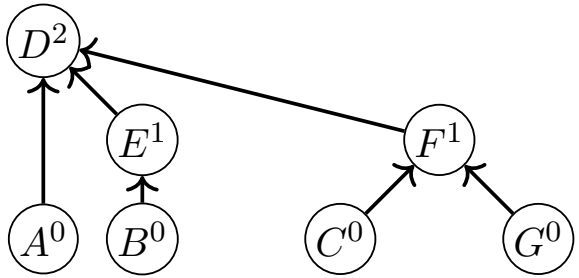
$\text{union}(A, D), \text{union}(B, E), \text{union}(C, F)$:



$\text{union}(E, A), \text{union}(C, G)$:



$\text{union}(B, G)$:



La fonction union

Fonction union(x, y)

$r_x \leftarrow \text{find}(x)$
$r_y \leftarrow \text{find}(y)$
si $r_x = r_y$ **alors**
  Retour
si $\mathit{rank}(r_x) > \mathit{rank}(r_y)$ **alors**
  $\pi(r_y) \leftarrow r_x$
sinon
  $\pi(r_x) \leftarrow r_y$
  si $\mathit{rank}(r_x) = \mathit{rank}(r_y)$ **alors**
    $\text{rank}(r_y) \leftarrow \text{rank}(r_y) + 1$

- $\text{rank}(x)$ est la hauteur de la sous-arborescence avec racine x.
- Cela implique, par exemple, que $\text{rank}(x) < \text{rank}(\pi(x))$, pour tout sommet x.

Nombre de sommets en terme de rank

Lemme

Soit x un sommet d'une arborescence A. Si $\text{rank}(x) = k$, alors la sous-arborescence avec racine x a au moins 2^k sommets.

Démonstration

- Vrai pour $k = 0$.
- Supposons que la proposition est vraie pour un entier $k \geq 0$, et soit x un sommet d'une arborescence A telle que $\texttt{rank}(x) = k + 1$.
- On a $\texttt{rank}(x) = k + 1$ parce que l'on a fusionné deux arborescences A_1, A_2, la racine de chacune ayant rang k.
- Par l'hypothèse de récurrence, A_1 et A_2 ont donc chacune au moins 2^k sommets.
- A a donc au moins $2 \cdot 2^k = 2^{k+1}$ sommets.

Complexité de l'algorithme de Kruskal

- Par conséquent, $\text{rank}(x) \leq \log_2 n$, pour tout sommet x.
- find et union sont donc de complexité $O(\log n)$.
- Nous pouvons conclure que l'algorithme de Kruskal est de complexité $O(m \log m) = O(m \log n)$, par exemple, en utilisant mergesort pour trier les arêtes.

mergesort

L'algorithme de Prim

Entrées : Un graphe connexe $G = (V, E)$ avec des poids w_e sur les arêtes

Sorties : Ensemble d'arêtes $X \subseteq E$ d'un arbre couvrant de G

$X \leftarrow \emptyset$

Choisir arbitrairement un sommet s et le marquer

tant que $\exists$ *un sommet non marqué adjacent à un sommet marqué* **faire**

- Trouver un sommet v non marqué adjacent à un sommet marqué u minimisant le poids de l'arête uv
- $X \leftarrow X \cup \{uv\}$
- Marquer v

Illustration de l'algorithme de Prim

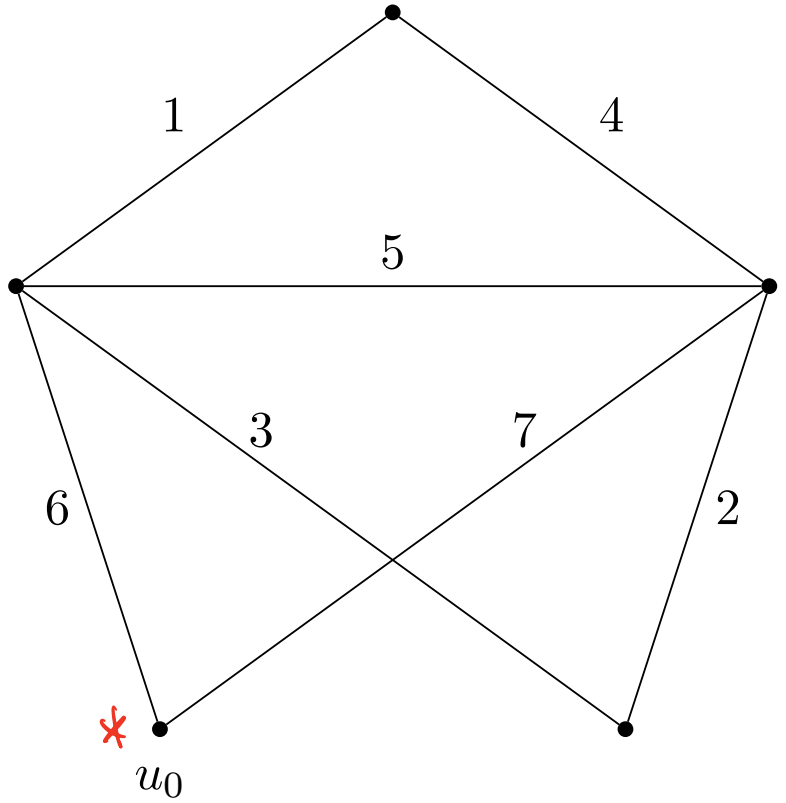


Illustration de l'algorithme de Prim

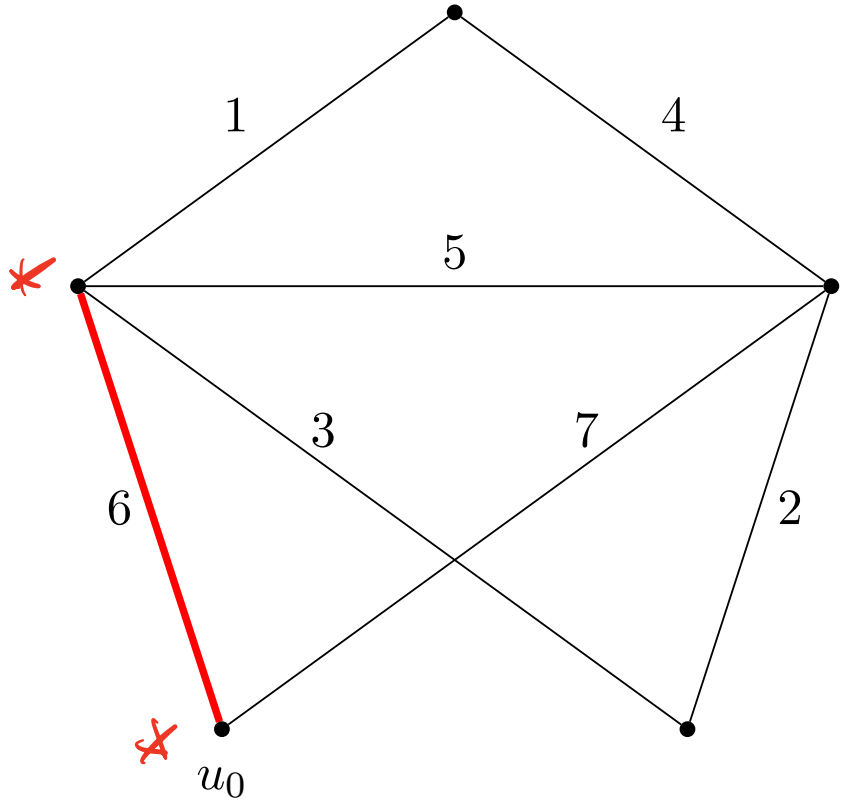


Illustration de l'algorithme de Prim

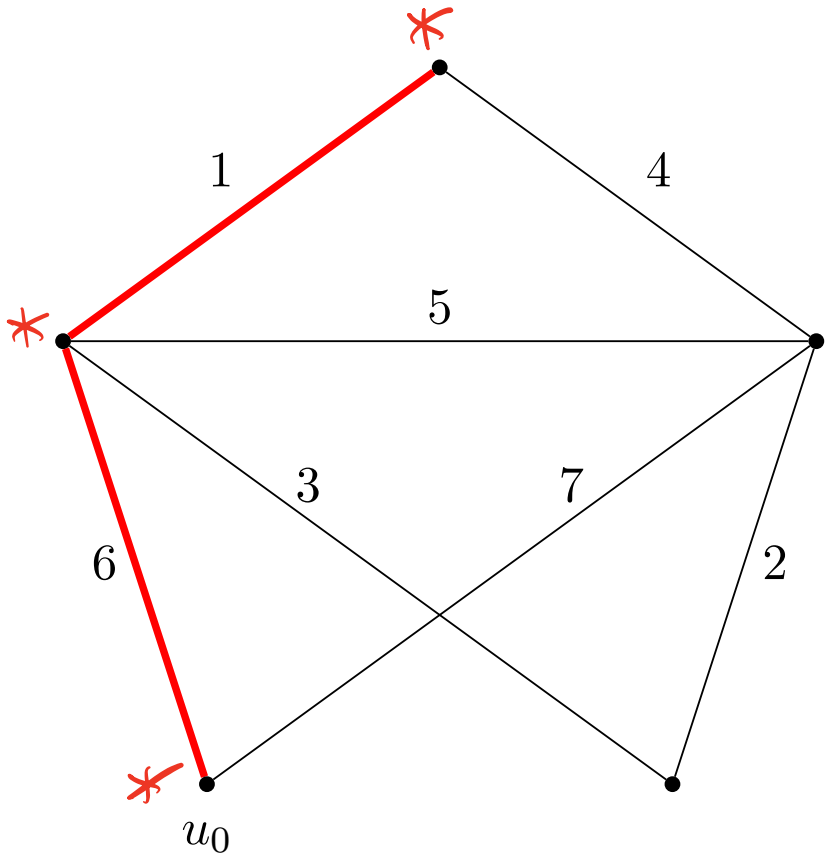


Illustration de l'algorithme de Prim

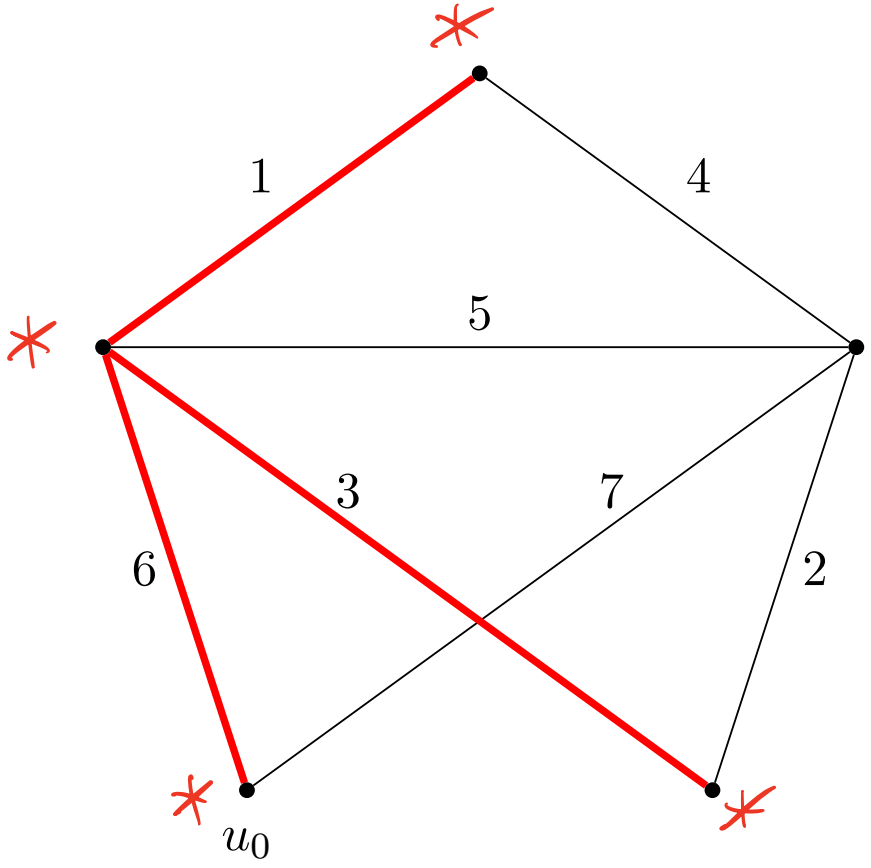
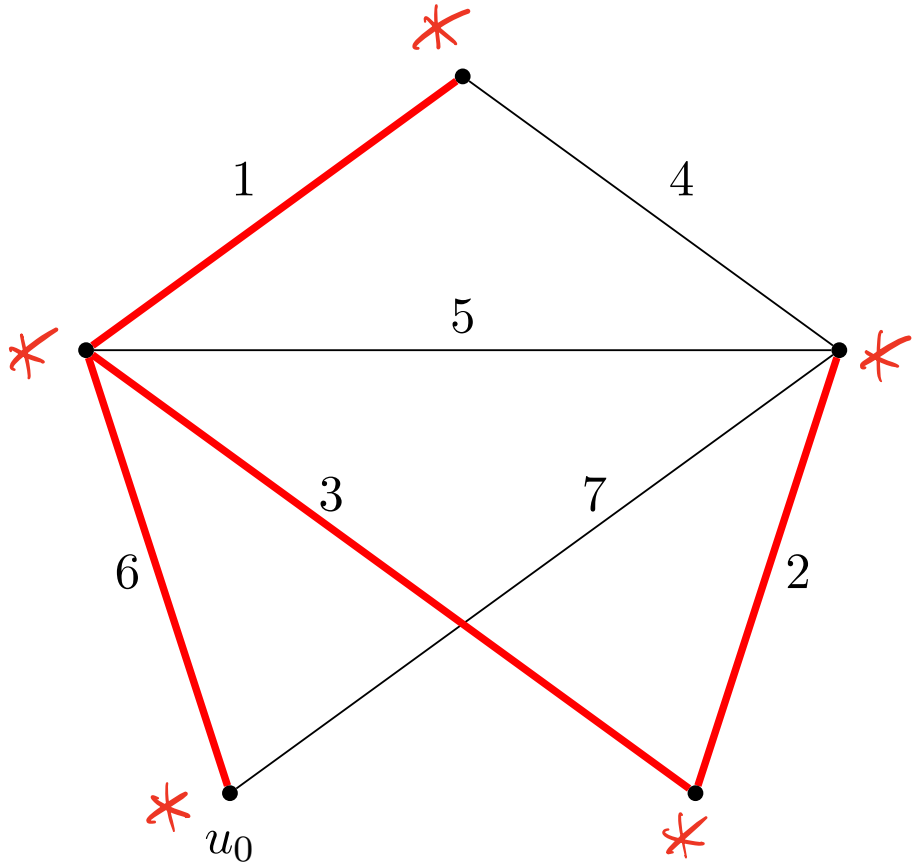


Illustration de l'algorithme de Prim



Justification de l'algorithme de Prim

- Rappelons le lemme que nous avons prouvé au début du cours :

Lemme

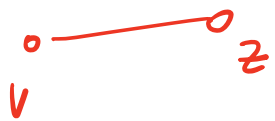
Pour tout sous-ensemble $S \subseteq V(G)$, l'arbre couvrant de poids minimum de G contient l'arête de poids minimum avec exactement une extrémité dans S.

- Si l'on note S l'ensemble de sommets marqués, alors le lemme garantit que l'arête que l'on ajoute à X est bien une arête de l'arbre couvrant de poids minimum.

L'algorithme de Prim (implémentation avec files de priorité)

Entrées : Un graphe connexe $G = (V, E)$ avec des poids w_e sur les arêtes
Sorties : Ensemble d'arêtes $X \subseteq E$ d'un arbre couvrant de G

pour tous les $u \in V$ **faire**
    coût$[u] \leftarrow \infty$
    `prev`$[u] \leftarrow \emptyset$
Choisir une source u_0
coût$[u_0] \leftarrow 0$
$H \leftarrow$ makequeue(V) `// file de priorité avec coûts comme clés`
tant que $H \neq \emptyset$ **faire**
    $v \leftarrow$ deletemin(H)
    pour tous les $vz \in E$ **faire**
        si *coût*$[z] > w_{vz}$ **alors**
            coût$[z] \leftarrow w_{vz}$
            `prev`$[z] \leftarrow v$
            decreasekey(H, z)

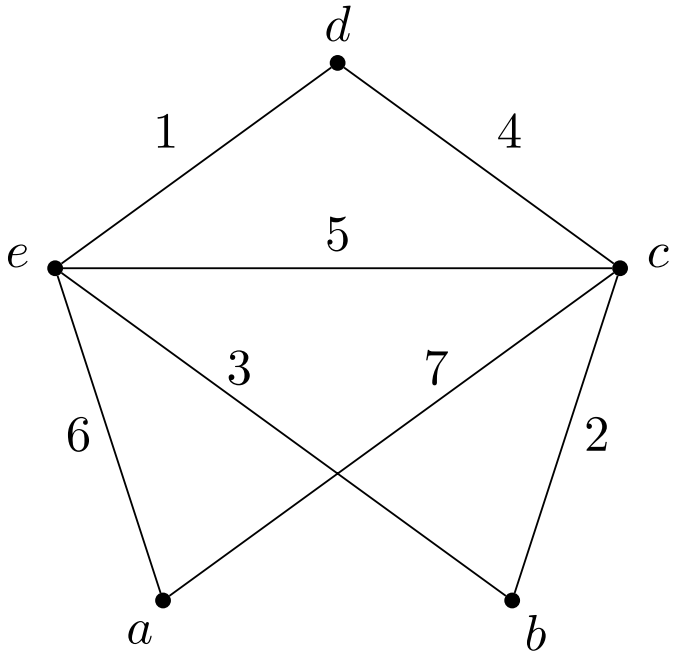


Complexité de l'algorithme de Prim

- L'algorithme de Prim est très proche à l'algorithme de Dijkstra.
- La seule différence est dans les valeurs des clés :
  - Dans l'algorithme de Prim, la valeur d'un sommet est le poids de l'arête entrante la plus légère
  - Dans l'algorithme de Dijkstra, c'est la longueur d'un chemin de la source vers ce sommet.
- La complexité est la même que celle de l'algorithme de Dijkstra, selon le type de tas utilisé :

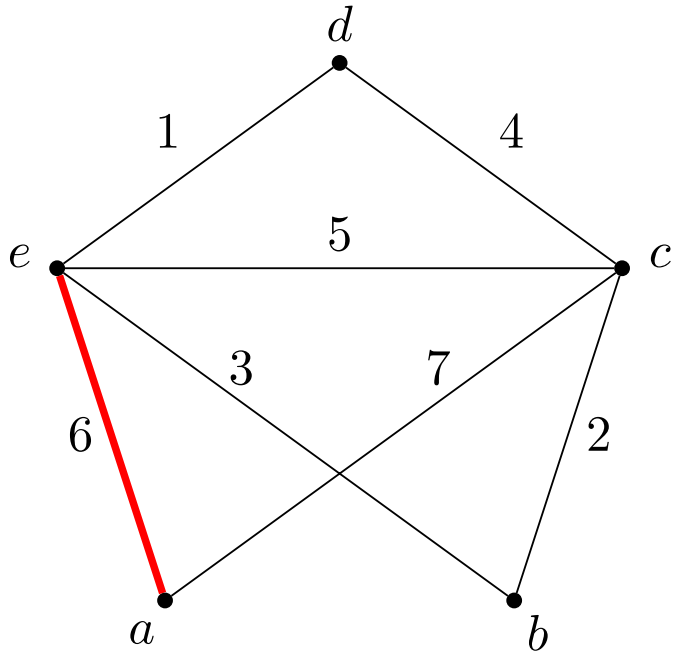
liste	$O(n^2)$
tas binaire	$O((n+m)\log n) = O(m \log n)$
tas de Fibonacci	$O(m + n \log n)$

Illustration de l'algorithme de Prim (version avec files de priorité)



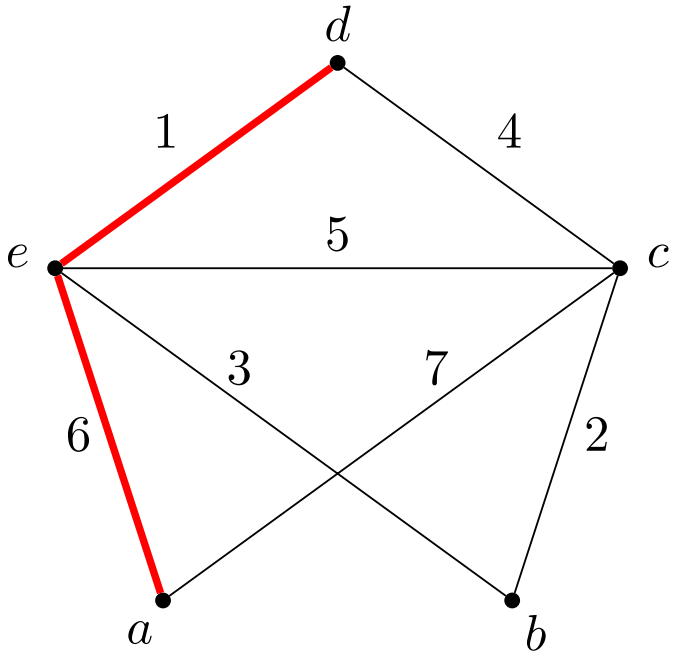
S	a	b	c	d	e
$\emptyset$	$0/\emptyset$	$\infty/\emptyset$	$\infty/\emptyset$	$\infty/\emptyset$	$\infty/\emptyset$

Illustration de l'algorithme de Prim (version avec files de priorité)



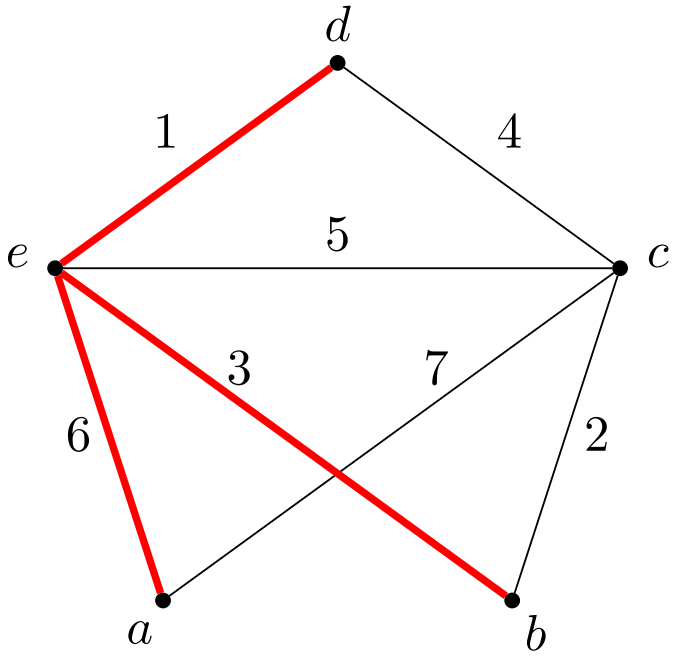
S	a	b	c	d	e
$\emptyset$	$0/\emptyset$	$\infty/\emptyset$	$\infty/\emptyset$	$\infty/\emptyset$	$\infty/\emptyset$
$\{a\}$		$\infty/\emptyset$	$7/a$	$\infty/\emptyset$	$6/a$

Illustration de l'algorithme de Prim (version avec files de priorité)



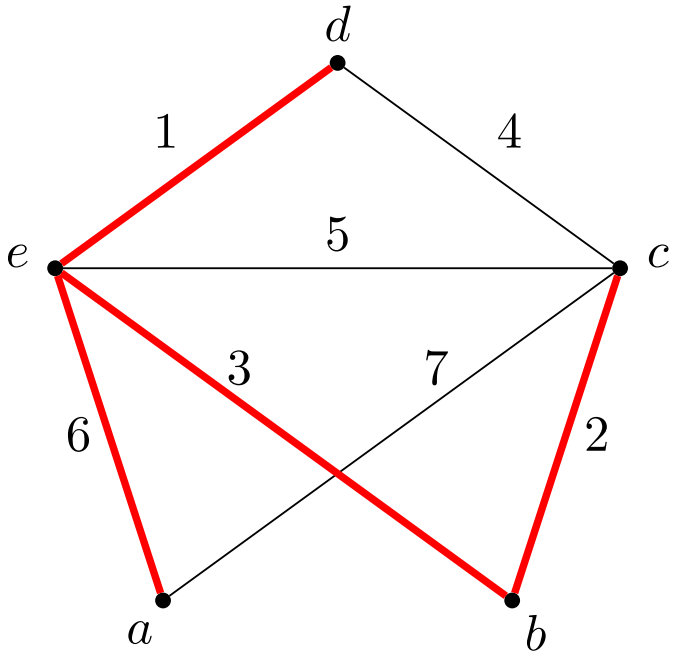
S	a	b	c	d	e
$\emptyset$	$0/\emptyset$	$\infty/\emptyset$	$\infty/\emptyset$	$\infty/\emptyset$	$\infty/\emptyset$
$\{a\}$		$\infty/\emptyset$	$7/a$	$\infty/\emptyset$	$6/a$
$\{a,e\}$		$3/e$	$5/e$	$1/e$	

Illustration de l'algorithme de Prim (version avec files de priorité)



S	a	b	c	d	e
$\emptyset$	$0/\emptyset$	$\infty/\emptyset$	$\infty/\emptyset$	$\infty/\emptyset$	$\infty/\emptyset$
$\{a\}$		$\infty/\emptyset$	$7/a$	$\infty/\emptyset$	$6/a$
$\{a,e\}$		$3/e$	$5/e$	$1/e$	
$\{a,e,d\}$		$3/e$	$4/d$		

Illustration de l'algorithme de Prim (version avec files de priorité)

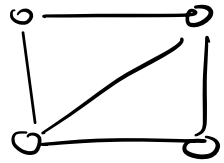


S	a	b	c	d	e
$\emptyset$	$0/\emptyset$	$\infty/\emptyset$	$\infty/\emptyset$	$\infty/\emptyset$	$\infty/\emptyset$
$\{a\}$		$\infty/\emptyset$	$7/a$	$\infty/\emptyset$	$6/a$
$\{a,e\}$		$3/e$	$5/e$	$1/e$	
$\{a,e,d\}$		$3/e$	$4/d$		
$\{a,e,d,b\}$			$2/c$		

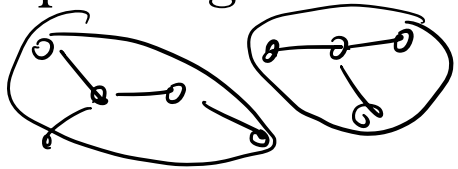
Pour finir : quelques mots sur l'algorithme de Borůvka

- Historiquement, le premier algorithme d'arbre couvrant de poids minimum.
- Soit $F \subseteq G$ la forêt consistant de n sommets isolés.
- Tant que F n'est pas un arbre, ajouter toutes les arêtes sûres de poids minimum.
- L'algorithme s'arrête lorsque F consiste en un seul arbre à n sommets, qui doit être l'arbre couvrant de poids minimum.
- Chaque itération réduit le nombre d'arbres d'au moins la moitié.
- L'algorithme se termine donc au bout de $\log n$ itérations.
- Une implémentation avec la structure de données union find est de complexité $O(m \log n)$; égale à celle de Kruskal et de Prim (implémentation avec tas binaire).

Bonus : un algorithme randomisé pour Min-Cut (1/3)

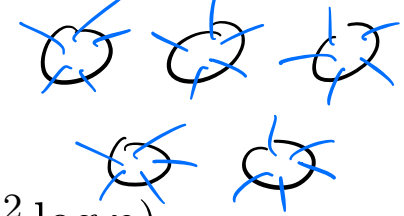


- Soit G un graphe connexe, non-pondéré à n sommets.
- Rappel : si $U \subseteq V(G)$, $\delta(U)$ est l'ensemble d'arêtes avec *exactement une extrémité* dans U.
- Une *coupe minimum* de G est un ensemble $\delta(U)$ de taille minimum, parmi toutes les sous-ensembles $U \subseteq V(G)$.
- Supposons que l'on retire la dernière arête ajouté par l'algorithme de Kruskal.

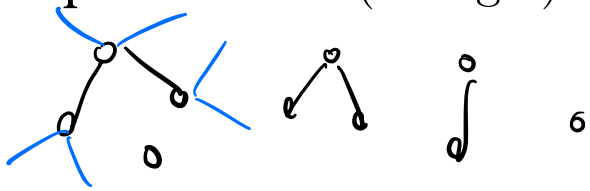


- On obtient ainsi deux arbres T_1 et T_2.
- Fait remarquable : $\delta(V(T_1))$ est une coupe minimum avec probabilité $2/(n(n-1))$.
- Conséquence : si l'on répète ce processus $O(n^2)$ fois, et si les arêtes sont triées uniformément au hasard, alors la plus petite coupe trouvée est une coupe minimum avec une forte probabilité.

Bonus : un algorithme randomisé pour Min-Cut (2/3)



- La complexité de cet algorithme randomisé est de $O(mn^2 \log n)$.
- L'algorithme peut être affiné pour obtenir une complexité de $O(n^2 \log n)$ (l'algorithme de Karger).
- Soit C la taille d'une coupe minimum de G.



- À chaque étape de l'algorithme de Kruskal, chaque composante connexe de (V, X) est incidente à au moins C arêtes sûres.
- Si (V, X) consiste de k composantes connexes, il doit y avoir $kC/2$ arêtes sûres.
- Comme les arêtes sont triées uniformément au hasard, la probabilité que l'arête choisie par l'algorithme de Kruskal est dans la coupe minimum est au plus $C/(kC/2) = 2/k$.

Bonus : un algorithme randomisé pour Min-Cut (3/3)

- Donc, la probabilité que que l'arête choisie par l'algorithme de Kruskal *n'est pas* dans la coupe est de $1 - 2/k = (k-2)/k$.
- Ainsi, la probabilité que l'algorithme de Kruskal ne choisie aucune arête de la coupe minimum est de

$$\frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdots \frac{2}{4} \cdot \frac{1}{3} = \frac{2}{n(n-1)}.$$