Comment trouver la repondération ?

- Il suffit de prouver l'existence d'une pondération $\ell' \in \mathbb{R}^m$ t.q. $\ell \geq 0$.
- Soit G' le graphe construit à partir de G en ajoutant un nouveau sommet s et les arcs $\{(s, v) : v \in V\}$.
- On étend la pondération ℓ à une pondération de G' en posant $\ell_{(s,v)} = 0$ pour tout $v \in V$.
- G' ne contient aucun cycle négatif ssi G ne contient aucun cycle négatif.
- Supposons que G et G' ne contiennent aucun cycle négatif.
- On définit $h_v = \text{dist}(s, v)$ pour tout sommet $v \in V(G')$.
- On a $h_v \leq h_u + \ell_{(u,v)}$ pour tout arc $(u, v) \in E(G')$.
- Donc, $\ell'_{(u,v)} = \ell_{(u,v)} + h_u - h_v \geq 0$.

Algorithme de Johnson

1. Calculer G'.
2. Appliquer Bellman–Ford à G', avec source s, pour calculer $h_v := \mathrm{dist}(s, v)$ pour tout $v \in V(G)$ (ou trouver un cycle négatif)
3. Repondérer chaque arc $(u, v) \in E(G)$ par $\ell'_{(u,v)} = \ell_{(u,v)} + h(u) - h(v)$.
4. Pour chaque $u \in V(G)$, exécuter Dijkstra pour calculer $\mathrm{dist}_{\ell'}(u, v)$ pour tout $v \in V(G)$.
5. Pour chaque couple u, v, on a $\mathrm{dist}_{\ell}(u, v) = \mathrm{dist}_{\ell'}(u, v) + h(v) - h(u)$.

Complexité de l'algorithme de Johnson

- L'étape 1 : $O(n)$
- L'étape 2 : $O(nm)$
- L'étape 3 : $O(m)$
- L'étape 4 : $O(nm + n^2 \log n)$
- L'étape 5 : $O(n^2)$
- Donc, l'algorithme de Johnson est de complexité $O(nm + n^2 \log n)$.
- Pour des graphes *peu denses*, l'algorithme de Johnson est donc plus rapide que l'algorithme de Floyd–Warshall.

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.

L'essence de l'algorithme de Kruskal

- Le problème de trouver un arbre couvrant de poids minimum de $G = (V, E)$ peut être résolu avec *l'algorithme de Kruskal*.
- L'algorithme construit un arbre couvrant de poids minimum à partir du graphe sans arêtes $(V, \varnothing)$ en ajoutant des arêtes de E une par une selon la règle suivante :

  Ajouter l'arête la plus légère qui ne crée pas de cycle.

- C'est un exemple d'un *algorithme glouton*.

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

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

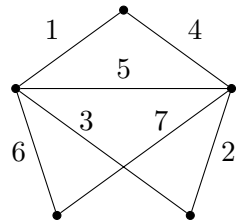
$X \leftarrow \varnothing$;

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

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

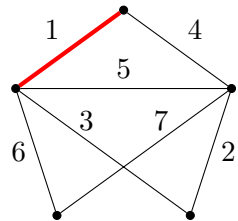
$X \leftarrow \varnothing$;

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

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

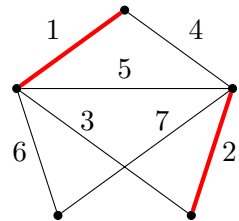
$X \leftarrow \varnothing$;

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

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

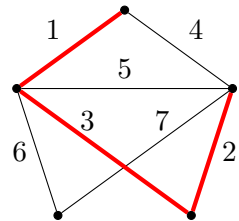
$X \leftarrow \varnothing$;

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

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

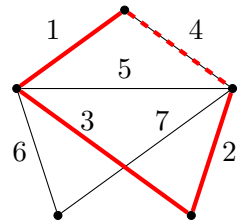
$X \leftarrow \varnothing$;

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

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

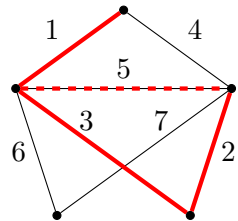
$X \leftarrow \varnothing$;

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

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

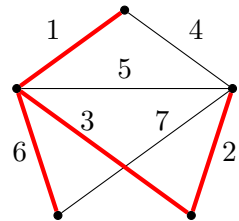
$X \leftarrow \varnothing$;

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

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

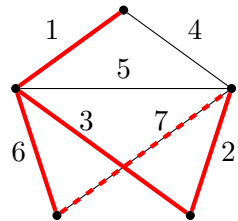
$X \leftarrow \varnothing$;

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

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

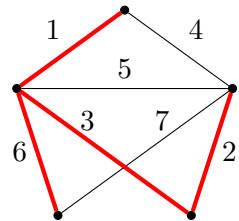
$X \leftarrow \varnothing$;

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 (1/3)

- L'algorithme s'arrête au pire lorsque toutes les arêtes dans E ont été considérées.
- Comme E est fini, l'algorithme s'arrête au bout d'un nombre fini d'opérations élémentaires.
- Soit $X \subseteq E$ la sortie de l'algorithme.
- $T = (V, X)$ est un sous-graphe couvrant de G, puisqu'il contient tous les sommets de G.
- Supposons par l'absurde que T n'est pas connexe.
- Soient T_1 et T_2 deux composantes connexes de T, et soit e une arête dans X entre T_1 et T_2 de poids minimum (l'arête e existe parce que G est connexe).
- Comme $(V, X \cup \{e\})$ est acyclique, l'algorithme aurait ajouté e à X.

Justification de l'algorithme (2/3)

- Donc, T est connexe.
- De plus, T est acyclique par la condition de la boucle.
- Cela montre que T est un arbre couvrant de G.
- Il reste à montrer que T est de poids minimum.
- Soit $T_0 = (V, X_0)$ un arbre couvrant de G de poids minimum, tel que $|X \cap X_0|$ soit maximum.
- Nous allons prouver que $|X \cap X_0| = |X|$, c'est-a-dire, $T = T_0$.
- Supposons par l'absurde que $|X \cap X_0| < |X|$, et soit e_1 l'arête la plus légère dans $X_0 \setminus X$.

Justification de l'algorithme (3/3)

- Le graphe $(V, X \cup \{e_1\})$ contient un cycle C (voir la caractérisation des arbres).
- Comme T_0 est acyclique, il existe au moins une arête $e_2 \in E(C) \setminus X_0$.
- Si $w_{e_1} < w_{e_2}$, l'algorithme de Kruskal aurait choisi l'arête e_1 au lieu de e_2.
- Donc, $w_{e_1} \geq w_{e_2}$.
- Soit $X_1 = (X_0 \setminus \{e_1\}) \cup \{e_2\}$
- Comme $w(X_1) \leq w(X_0)$, (V, X_1) est un (autre) arbre couvrant de G de poids minimum, tel que $|X \cap X_1| > |X \cap X_0|$, contradiction avec l'hypothèse.
- Donc, $X = X_0$, et on conclut que $T = (V, X)$ est un arbre couvrant de G de poids minimum.

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)

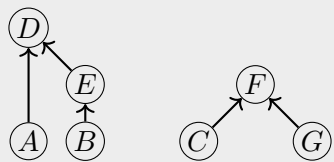
Remarque
L'algorithme appelle `makeset` $|V|$ fois, `find` $2|E|$ fois, et `union` $|V| - 1$ fois.

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 :



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.

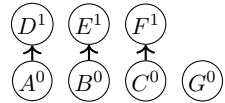
Bien fusionner deux ensembles

- 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 $\pi(r_A) = r_B$, alors la hauteur n'augmente pas.
- Avec cette stratégie, la hauteur augmente uniquement lorsque les deux arborescences ont la même hauteur (rank).

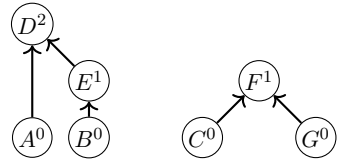
Une illustration

makeset$(A), \ldots,$ makeset(G) :

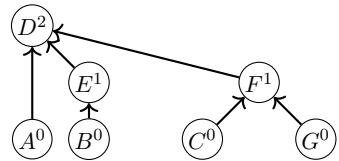
union(A, D), union(B, E), union(C, F) :



union(C, G), union(E, A) :



union(B, G) :



La fonction union

Fonction union

$r_x \leftarrow \text{find}(x)$;
$r_y \leftarrow \text{find}(y)$;
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 tel que $\texttt{rank}(x) = k + 1$.
- On a $\texttt{rank}(x) = k + 1$ parce qu'on a fusionné deux arborescences A_1, A_2 telles que le rang de la racine de chacune est k.
- Donc, par l'hypothèse de récurrence, A_1 et A_2 ont chacune au moins 2^k nœuds.
- Donc, A a au moins $2 \cdot 2^k = 2^{k+1}$ sommets.

Complexité de l'algorithme de Kruskal

- Par conséquent, rank$(x) \leq \log_2 n$, pour tout sommet x.
- Donc, find et union sont 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.