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

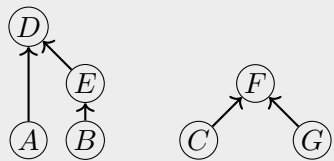
L'algorithme appelle `makeset` $|V|$ fois, `find` $2|E|$ fois, et `union` $|V| - 1$ fois.

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.

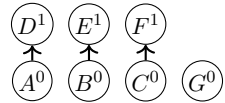
Fusionner deux ensembles efficacement

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

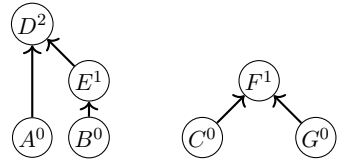
Une illustration

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

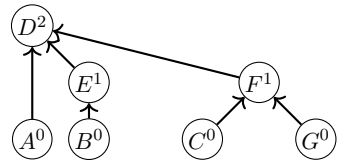
$\text{union}(A, D), \text{union}(B, E), \text{union}(C, F)$:



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



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



La fonction union

Fonction union

$r_x \leftarrow \text{find}(x)$
$r_y \leftarrow \text{find}(y)$
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 nœuds.
- A a donc 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.
- 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.

Relation entre arbres couvrants de poids minimum et coupes

Rappel

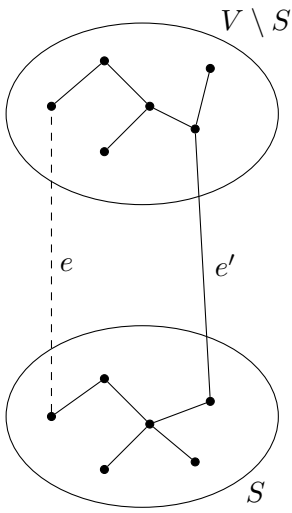
Soit $G = (V, E)$ un graphe et soit $U \subseteq V$. Alors, $\delta(U)$ est l'ensemble des arêtes avec une (et une seule) extrémité dans U. On dit que $\delta(U)$ est une *coupe*.

- Il y a une relation importante entre les arbres couvrants de poids minimum et les coupes :

Lemme

Soit X un sous-ensemble des arêtes d'un arbre couvrant de poids minimum de $G = (V, E)$. Soit $S \subseteq V$ t.q. aucune arête de X n'est présente dans la coupe $\delta(S)$. Soit e l'arête la plus légère dans $\delta(S)$. Alors $X \cup \{e\}$ est un sous-ensemble des arêtes d'un arbre couvrant de poids minumum.

Démonstration



- Soit $T = (V, Y)$ un arbre couvrant de poids minimum de G t.q. $X \subseteq Y$.
- Supposons que $e \notin Y$ (sinon $X \cup \{e\} \subseteq Y$ et on a fini).
- Le graphe $(V, Y \cup \{e\})$ contient un cycle C (voir la characterisation des arbres).
- Il doit y avoir (au moins) une arête $e' \in Y \cap \delta(S)$.
- Par l'hypothèse, $w_e \leq w_{e'}$.
- Donc, $T' = (V, (Y \setminus \{e'\}) \cup \{e\})$ est un arbre couvrant de G de poids minimum qui contient $X \cup \{e\}$.

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

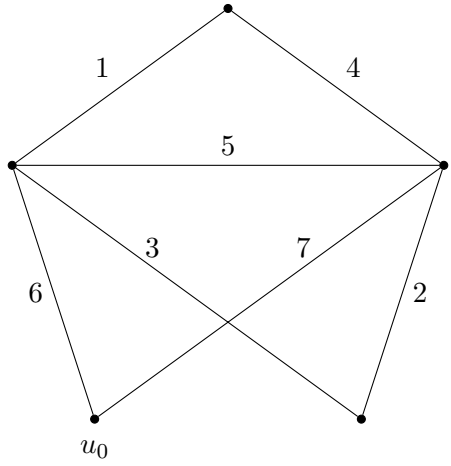
$X \leftarrow \emptyset$

Choisir arbitrairement un sommet s et le marquer

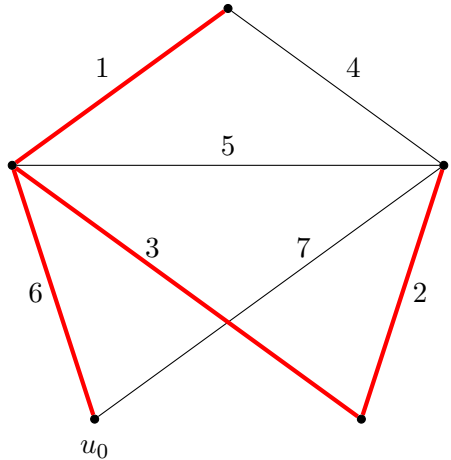
tant que $\exists$ *un sommet non marqué adjacent à un sommet marqué* **faire**

- 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

Exemple de l'exécution de l'algorithme de Prim



Exemple de l'exécution de l'algorithme de Prim



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

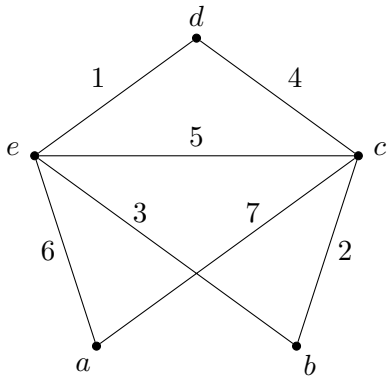
pour tous les $u \in V$ **faire**
    coût$[u] \leftarrow \infty$
    `prev`$[u] \leftarrow \emptyset$
Choisir une source u_0
coût$[u_0] \leftarrow 0$
$H \leftarrow$ makequeue(V) `// file de priorité avec coûts comme clés`
tant que $H \neq \emptyset$ **faire**
    $v \leftarrow$ deletemin(H)
    pour 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 :

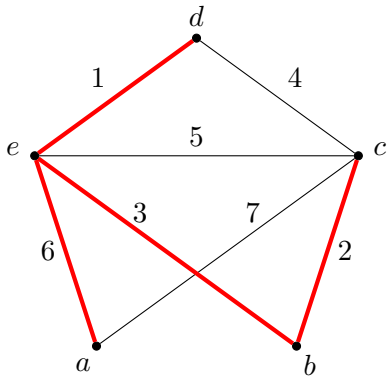
liste	$O(n^2)$
tas binaire	$O((n+m)\log n) = O(m \log n)$
tas de Fibonacci	$O(m + n \log n)$

Exemple de l'exécution de l'algorithme de Prim (version avec files de priorité)



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

Exemple de l'exécution de l'algorithme de Prim (version avec files de priorité)



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