Chapitre 6 : Trois algorithmes sur les graphes

1 Arbres couvrants de poids minimum

Dans cette section, on travaille sur un graphe non orienté pondéré $G = (S, A, c)$ où $c \in \mathcal{F}(A, \mathbb{N})$. On notera $n = \text{card}(S)$ et $m = \text{card}(A)$.

Vocabulaire 1.1

Dans les problèmes que l'on considère dans cette section, l'ensemble des sommets ne change pas, et l'on cherche plutôt un ensemble d'arêtes $A' \subseteq A$ qui définit un arbre couvrant. On s'autorisera donc à dire qu'un ensemble d'arêtes $A' \subseteq A$ est connexe, acyclique, un arbre, un arbre couvrant ... pour dire que le graphe (S, A') l'est.

Notation 1.2

Suivant la même idée, si $A' \subseteq A$, on notera $\sim_{A'}$ la relation de connexité du graphe (S, A'). Ainsi $\forall (u, v) \in S^2, u \sim_{A'} v$ si et seulement si il existe une chaîne d'arêtes de A' entre u et v.

1.1 Arbres

Rappel 1.3

*Le graphe G est **acyclique** si G n'admet aucun cycle élémentaire de longueur supérieure ou égale à 3.*
*Le graphe G est **connexe** si pour tout couple de sommets il existe une chaîne ayant ces deux sommets comme extrémités.*
*Le graphe G est un **arbre** si et seulement si G est connexe et acyclique.*

Exercice de cours 1.4

Que dire d'un sous-graphe d'un graphe acyclique ?
Que dire d'un sur-graphe d'un graphe connexe ?

Lemme 1.5

Soit $B \subseteq A$ un sous-ensemble d'arêtes.
Si (S, B) admet un cycle élémentaire γ, et si e est une arête de γ,
alors $B' \overset{\text{déf}}{=} B \setminus \{e\}$ vérifie $\sim_B = \sim_{B'}$.
Autrement dit enlever une arête sur un cycle ne change pas la connexité.

Démonstration : La preuve est un exercice de TD. □

Lemme 1.6

Soit $B \subseteq A$ un sous-ensemble d'arêtes.
Si (S, B) est acyclique et si x et y sont deux sommets de S tels que $x \not\sim_B y$,
alors $(S, B \cup \{x, y\})$ est acyclique.
Autrement dit ajouter une arête entre deux sommets non reliés ne crée pas de cycle.

Démonstration : La preuve est un exercice de TD. □

Proposition 1.7

Les 5 propositions ci-dessous sont équivalentes.
- *G est connexe et acyclique*
- *G est connexe et $|A| = |S| - 1$*
- *G est acyclique et $|A| = |S| - 1$*
- *G est minimal parmi les sous-graphes connexes de K_n* ♣
- *G est maximal parmi les sous-graphes acycliques de K_n*

Démonstration : La preuve est un exercice de TD. □

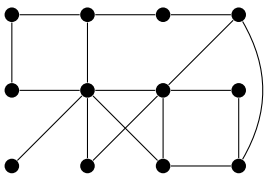
1.2 Arbres couvrants

Définition 1.8

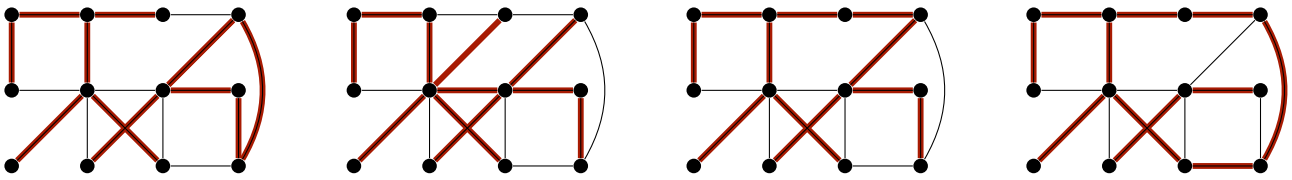
*On dit d'un graphe $G' = (S', A')$ que c'est un **arbre couvrant** de G dès lors que : $S' = S$, $A' \subseteq A$ et G' est un arbre.*

Exercice de cours 1.9

On considère le graphe G ci-dessous.



Parmi les graphes ci-dessous (représentés en —), lesquels sont des arbres couvrants de G ? Justifier.



Proposition 1.10

Un graphe admet un arbre couvrant si et seulement s'il est connexe.

Démonstration : Si G admet un arbre couvrant A', alors par définition d'un arbre, (S, A') est connexe. Deux sommets quelconques de G sont reliés par une chaîne d'arêtes de A', et donc a fortiori par une chaîne d'arêtes de A, et sont donc reliés dans G. Ainsi G est connexe. Réciproquement si G est connexe, il suffit

♣. K_n est le graphe complet à n sommets.

d'enlever des arêtes à A sur des cycles tant qu'il y en a . En effet, en posant $B = A$, on a $\sim_B = \sim_G$ par définition. Tant que B admet un cycle, on choisit e une arête de ce cycle, et on la supprime de B. D'après le lemme 1.5, on maintient ainsi $\sim_B = \sim_G$. Le nombre d'arêtes de B est un variant qui assure que cette procédure termine, et en sortie on obtient bien B un ensemble d'arête acyclique, autant connexe que G, soit un arbre couvrant de G. □

Définition 1.11

*On dit d'un graphe $G' = (S', A')$ que c'est une **forêt couvrante** de G dès lors que : $S' = S$, $A' \subseteq A$ et G' est acyclique et $\sim_G = \sim_{G'}$. Autrement dit, une forêt couvrante d'un graphe G est un sous-graphe de G acyclique qui a exactement les mêmes composantes connexes que G.*

Exercice de cours 1.12

Démontrer que tout graphe admet une forêt couvrante.

Exercice de cours 1.13

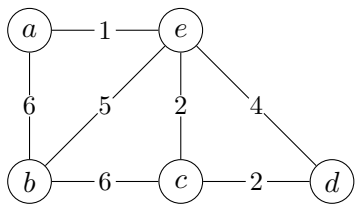
Soit un graphe connexe G à n sommets et m arêtes. Justifier que le nombre d'arbres couvrants de G est fini en donnant, en fonction de n et m, une borne sur le nombre d'arbres couvrants.

1.3 Arbre couvrant et pondération

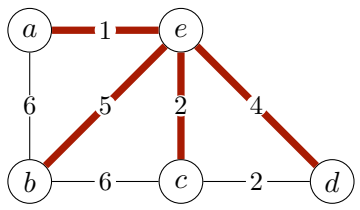
Définition 1.14

*Si $A' \subseteq A$, le **poids** de A' est $\sum_{\{x,y\} \in A'} c(x, y)$, et parfois noté $c(A')$.*
*Si $T = (S', A')$ est un arbre, son **poids**, parfois noté $c(T)$ est $c(A')$.*
Autrement dit le poids d'un arbre est la somme des poids de ses arêtes.

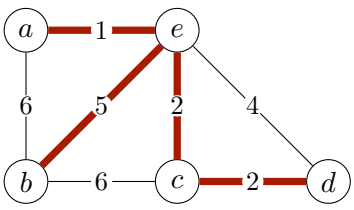
Exemple 1.15



Un graphe non orienté pondéré



Un arbre couvrant de poids 12



Un arbre couvrant de poids minimum (10)

Définition 1.16

Étant donné qu'il y a un nombre fini non nul d'arbres couvrants d'un graphe connexe, on peut alors considérer le problème d'optimisation suivant.

$$\textit{ACPM} : \begin{cases} \textbf{\textit{Entrée :}}\ \textit{Un graphe connexe non orienté pondéré } G = (S, A, c) \\ \textbf{\textit{Sortie :}}\ \textit{Un arbre couvrant de poids minimum} \end{cases}.$$

Exercice de cours 1.17

Considérons le problème d'optimisation suivant.

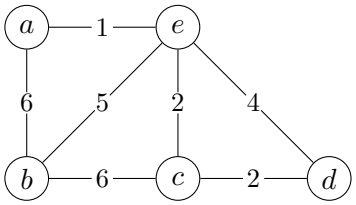
$$\text{ECPM} : \begin{cases} \textbf{Entrée :} \text{ Un graphe connexe non orienté pondéré } G = (S, A, c) \\ \textbf{Sortie :} \text{ Un ensemble d'arêtes } A' \subseteq A \text{ tel que } \sim_A = \sim_{A'} \text{ minimisant } c \end{cases}.$$

Montrer que l'ensemble des arbres est dominant pour ECPM, *i.e.* qu'il existe toujours au moins une solution optimale qui est un arbre.
Montrer de plus que dans le cas où c est à valeurs strictement positive cette dominance est stricte, c'est-à-dire que toutes les solutions optimales sont des arbres.

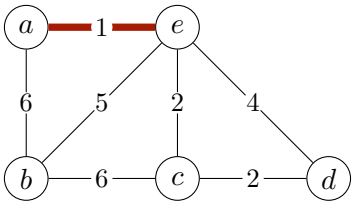
1.4 Algorithme de Kruskal

Dans cette section on suppose que G est connexe et on cherche un arbre couvrant de poids minimum de G. Une idée pour construire un tel arbre, est de partir d'un ensemble d'arêtes vide, qui a le mérite d'être acyclique, et de l'enrichir en ajoutant des arêtes pour qu'il gagne en connexité jusqu'à atteindre celle de G, tout en maintenant son caractère acyclique. On regarde à chaque étape la partition en composantes connexes associée à l'ensemble d'arêtes. D'une part cela permet de détecter si l'on a atteint la connexité voulue (lorsqu'il y a une seule composante), et d'autre part cela permet de détecter qu'une arête n'est pas bonne à ajouter (si ces deux extrémités sont déjà dans la même composante).

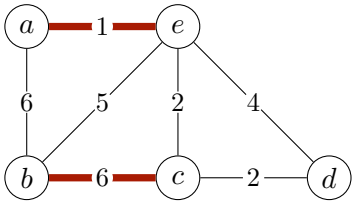
Exemple 1.18
Reprenons l'exemple ci-dessus, et mettons en regard des choix d'arêtes et les partitions en composantes connexes associées.



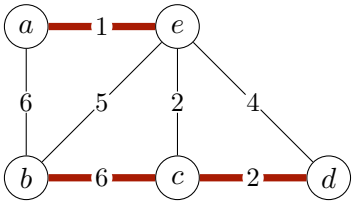
$\{\{a\}, \{e\}, \{c\}, \{b\}, \{d\}\}$



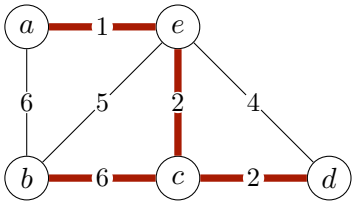
$\{\{a, e\}, \{c\}, \{b\}, \{d\}\}$



$\{\{a, e\}, \{c, b\}, \{d\}\}$



$\{\{a, e\}, \{c, b, d\}\}$



$\{\{a, e, c, b, d\}\}$

Le coût de l'arbre couvrant ainsi généré est alors la somme des coûts des arêtes sélectionnées à chaque étape, ce qui suggère de considérer les arêtes de plus petit coût d'abord.
Ce qui donne l'algorithme glouton suivant.

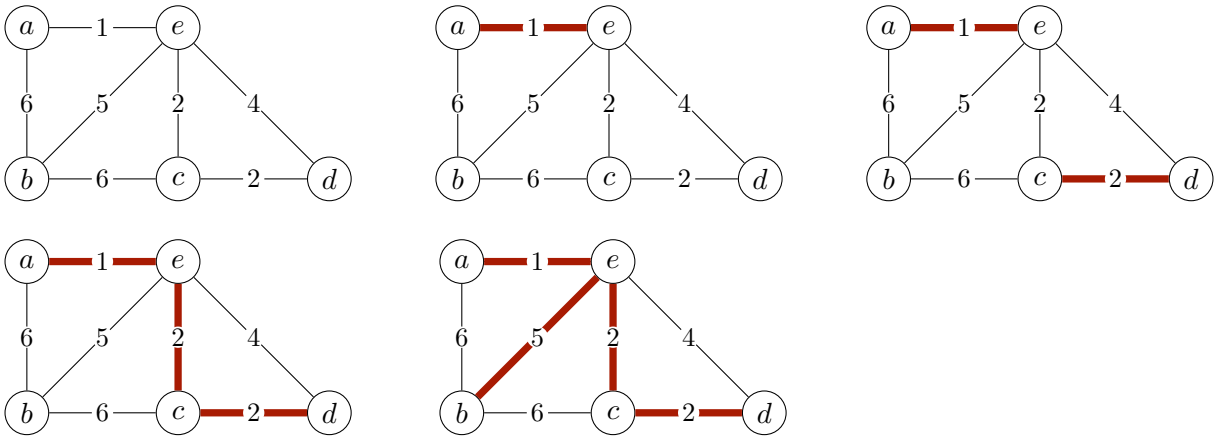
Algorithme 1 : Algorithme de Kruskal (version 0)

Entrée : Un graphe connexe non orienté pondéré $G = (S, A, c)$
Sortie : Un arbre couvrant de poids minimum

$(a_j)_{j\in[\![1,m]\!]} \leftarrow$ une indexation des arêtes par pondération croissante ;
$B \leftarrow \emptyset$;
$i \leftarrow 1$;
tant que le graphe (S, B) n'est pas connexe **faire**
  si le graphe $(S, B \cup \{a_i\})$ est acyclique **alors**
    $B \leftarrow B \cup \{a_i\}$;
  $i \leftarrow i + 1$;
retourner (S, B) ;

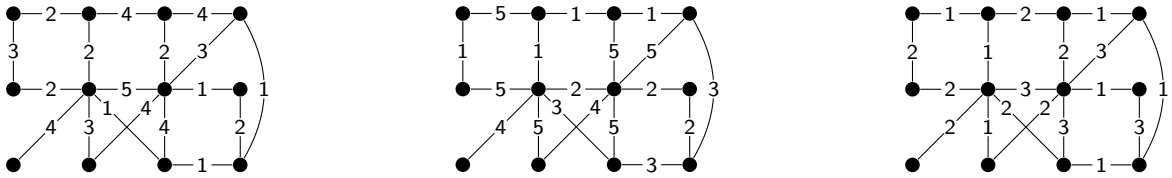
Exemple 1.19

L'exécution de l'algorithme sur l'exemple ci-dessus conduit aux choix représentés ci-dessous.



Exercice de cours 1.20

Exécuter l'algorithme de Kruskal sur les graphes ci-dessous.



Théorème 1.21

L'algorithme de Kruskal fournit un arbre couvrant de poids minimal.

Démonstration : Nous allons mener la preuve en trois temps : établir des propriétés invariantes de la boucle **tant que** de l'algorithme, démontrer la terminaison de l'algorithme, conclure en utilisant les invariants et la négation de la condition de boucle. Pour chaque $j \in [\![1, m]\!]$, notons x_j et y_j les extrémités de l'arête a_j.

Invariants. Commençons par démontrer que les propriétés suivantes sont des invariants de la boucle **tant que** l'algorithme de Kruskal.

I_1 Il existe un arbre couvrant de poids minimal de G contenant les arêtes de B.

I_2 B est acyclique.

I_3 $\forall j \in [\![1, i-1]\!]$, $x_j \underset{B}{\sim} y_j$.

I_4 $i \in [\![1, m+1]\!]$.

Initialisation. Avant les itérations, $B = \emptyset$ et $i = 1$. Ceci assure :

- la propriété I_1, en effet G étant connexe il admet un arbre couvrant de poids minimal, et donc un arbre couvrant de poids minimal contenant les arêtes de $\emptyset$;
- la propriété I_2, en effet le graphe $\emptyset$ est trivialement acyclique ;
- la propriété I_3, en effet $[\![1, i-1]\!] = \emptyset$;
- la propriété I_4, en effet $i = 1$.

Propagation de l'invariant. Soit B^{av} et i^{av} les valeurs des variables B et i au début d'une itération de boucle, et B^{ap} et i^{ap}, les valeurs à la fin de cette même itération.

Supposons que les propriétés I_1, I_2, I_3 et I_4 sont vérifiées par B^{av} et i^{av}.

Montrons qu'elles le sont alors toujours par les valeurs B^{ap} et i^{ap}.

Remarquons tout d'abord que $i^{ap} = i^{av} + 1$. De plus, d'après la condition de boucle, B^{av} n'est pas connexe.

I_4 Montrons que $i^{ap} \in [\![1, m+1]\!]$.
Par invariant I_4, $i^{av} \in [\![1, m+1]\!]$, montrons en fait que $i^{av} \neq m+1$. Par l'absurde supposons que $i^{av} = m+1$. L'invariant I_3 assure alors $\forall j \in [\![1, m]\!], x_j \underset{B^{av}}{\sim} y_j$, autrement dit les deux extrémités de n'importe quelle arête de G sont reliées dans B^{av} ($\bigstar$). Montrons alors que B^{av} est connexe.
Soit $(x, y) \in S^2$. Par connexité de G, il existe une chaîne de x à y dans G, qu'on note comme suit.

$$x = \gamma_0 \underset{G}{—} \gamma_1 \underset{G}{—} \gamma_2 \underset{G}{—} \cdots \underset{G}{—} \gamma_p = y$$

De la remarque ($\bigstar$), on déduit que $\forall i \in [\![0, p-1]\!], \gamma_i \underset{B^{av}}{\sim} \gamma_{i+1}$ et finalement par transitivité $x = \gamma_0 \underset{B^{av}}{\sim} \gamma_p = y$. Ainsi B^{av} est connexe, Absurde.
On en déduit que $i^{av} \neq m+1$ donc $i^{av} \in [\![1, m]\!]$ donc $i^{ap} \in [\![2, m+1]\!]$ et a fortiori $i^{ap} \in [\![1, m+1]\!]$.

I_3 Montrons que $\forall j \in [\![1, i^{ap}-1]\!], x_j \underset{B^{ap}}{\sim} y_j$.
Soit $j \in [\![1, i^{ap}-1]\!]$. Si $j \in [\![1, i^{av}-1]\!]$, puisque $B^{av} \subseteq B^{ap}$, la propriété I_3 en début de tour assure $x_j \underset{B^{ap}}{\sim} y_j$. Si $j = i^{ap} - 1 = i^{av}$, on distingue deux cas.
- Cas $B^{ap} = B^{av} \cup \{a_{i^{av}}\}$, soit $B^{ap} = B^{av} \cup \{a_j\}$. La chaîne réduite à l'arête a_j assure $x_j \underset{B^{ap}}{\sim} y_j$.
- Sinon $B^{ap} = B^{av}$. D'après la condition du **si**, $B^{av} \cup \{a_{i^{av}}\}$ soit $B^{av} \cup \{a_j\}$ n'est pas acyclique, or B^{av} est acyclique par invariant I_2. Ainsi, par contraposé du Lemme 1.6, $x_j \underset{B^{ap}}{\sim} y_j$.

I_2 Montrons que B^{ap} est acyclique.
Si $B^{ap} = B^{av} \cup \{a_{i^{av}}\}$, c'est parce que $B^{av} \cup \{a_{i^{av}}\}$ est acyclique (d'après la condition du **si**). Sinon $B^{ap} = B^{av}$ qui est acyclique par invariant I_2.

I_1 Montrons qu'il existe un arbre couvrant de poids minimal de G contenant les arêtes de B^{ap}.
Soit $T \subseteq A$ un arbre couvrant de poids minimal de G contenant les arêtes de B^{av} (un tel arbre existe par invariant I_1).
Si $B^{ap} \subseteq T$, on conclut en considérant le même arbre.
Sinon $B^{ap} \not\subseteq T$, autrement dit on a sélectionné dans B une arête lors du tour de boucle considéré, ainsi $B^{ap} = B^{av} \sqcup \{a_{i^{av}}\}$. De plus, d'après la condition du **si**, B^{ap} acyclique.
Notons $\{x, y\} \overset{\text{déf}}{=} a_{i^{av}}$, et posons alors $U \overset{\text{déf}}{=} T \sqcup \{a_{i^{av}}\}$. Ainsi U est connexe et contient B^{ap}, mais il n'est pas acyclique. En effet, puisque T est un arbre, on peut considérer δ l'unique chaîne élémentaire de x à y dans T, et former un cycle élémentaire γ en y ajoutant l'arête $a_{i^{av}}$.

$$\gamma : \underbrace{x — \ldots — y}_{\delta} \underset{a_{i^{av}}}{—} x$$

B^{ap} étant acyclique, il existe une arête de γ qui n'est pas dans B^{ap}. Notons-la f et remarquons que $f \neq a_{i^{av}}$ car $a_{i^{av}} \in B^{ap}$. Posons alors $T' \overset{\text{déf}}{=} U \setminus \{f\}$.
Du lemme 1.5, T' est encore connexe et contient toujours B^{ap}.
Montrons que T' est de plus acyclique. En effet, $T' = (T \setminus \{f\}) \cup \{a_{i^{av}}\}$ soit $T' = (T \setminus \{f\}) \cup \{x, y\}$. $T \setminus \{f\}$ est acyclique en tant que sous-graphe acyclique de T (lui-même un arbre). De plus f est

une arête de l'unique chaîne élémentaire dans T de x à y, donc $x \not\sim_{(T\setminus\{f\})} y$. D'après le lemme 1.6, on en déduit que T' est acyclique. Ainsi T' est un arbre contenant B^{ap}.
Montrons qu'il est aussi de poids minimal en montrant qu'il est de poids moindre que T.
Pour cela montrons finalement que le coût de l'arête f est moindre que celui de l'arête $a_{i^{av}}$. Considérons les deux sommets u et v tels que $f = \{u, v\}$ et l'entier k tel que $f = a_k$ et montrons que $k \geqslant i^{av}$.
Par l'absurde supposons que $k < i^{av}$. Alors l'invariant I_3 assure que $u \underset{B^{av}}{\sim} v$, ainsi il existe une chaîne élémentaire γ dans B^{av} reliant u à v. Puisque $B^{av} \subseteq T$, β est une chaîne de T. La chaîne β n'emprunte pas l'arête $f = \{u, v\}$ puisque celle-ci ne se trouve pas dans B^{av} (car on a choisi f hors de B^{ap}). Ainsi on a deux chaînes élémentaires distinctes reliant u et v dans T : l'arête f et la chaîne β. Absurde puisque T est un arbre.
Ainsi $k \geqslant i^{av}$, et puisque les arêtes sont triées par poids croissant, $c(a_k) \geqslant c(a_{i^{av}})$, assurant ainsi que $c(T') \leqslant c(T)$ et donc $c(T') = c(T)$ par minimalité.
Finalement, $\mathcal{T}'$ est donc bien un arbre couvrant de poids minimal contenant B^{ap}.

Variants. Montrons que la boucle **tant que** de l'algorithme de Kruskal termine. Considérons pour cela l'expression numérique suivante des variables de la boucle **tant que** B et i.

$$V(B, i) \overset{\text{déf}}{=} m + 1 - i$$

- Par I_4, $i \in [\![1, m+1]\!]$ ainsi $V(B, i) \in \mathbb{N}$.
- Avec les notations introduites ci-avant, $i^{ap} = i^{av} + 1$, ainsi $V(B^{ap}, i^{ap}) < V(B^{av}, i^{av})$.

Ainsi $V(B, i)$ est bien un variant de boucle à valeurs dans l'espace bien fondé $(\mathbb{N}, \leqslant)$, ce qui nous assure la terminaison de la boucle **Tant que**.

Conclusion. Finalement, les valeurs des variables B et i en sortie de boucle sont telles que :

I_1 il existe un arbre couvrant de poids minimal de G contenant les arêtes de B ;

I_2 B est acyclique ;

I_3 $\forall j \in [\![1, i-1]\!], x_j \underset{B}{\sim} y_j$;

I_4 $i \in [\![1, m+1]\!]$;

Négation de la condition de boucle : B est connexe.

On en déduit donc que B est un arbre couvrant et qu'il est contenu dans un arbre couvrant de poids minimal, il est donc lui aussi de poids minimal.

□

Maintenant que nous nous sommes convaincus que le choix glouton consistant à prendre à chaque étape l'arête de plus petit poids conduit bien à un arbre de poids minimal, il nous faut nous demander comment nous allons implémenter les opérations "le graphe (S, B) est-il connexe ? " ou encore "le graphe $(S, B \cup \{a_i\})$ est-il acyclique ?". La donnée, à chaque instant de l'algorithme, de l'ensemble des composantes connexes du graphe $G = (S, B)$ nous permet de répondre "aisément" à ces deux questions. De plus nous remarquons que l'évolution des composantes connexes du graphe (S, B), à mesure que l'algorithme se déroule, peut être exprimée à l'aide de fusions de partitions, depuis le partitionnement trivial (dans lequel chaque élément est seul dans sa partie). En effet initialement $B = \emptyset$, aussi la décomposition en composantes connexes du graphes (S, B) est la décomposition en des parties singletons. L'ajout d'une arête à B a pour effet la fusion des composantes connexes des sommets se trouvant aux deux extrémités de l'arête en question. On se pose alors la question de l'implémentation d'une structure de données qui permette la représentation de partitionnements, sur lesquels il est possible de faire des opérations de fusion.

1.5 Union Find

Définition 1.22

On définit le type de données abstrait UnionFind comme contenant :

- *un type elt des éléments manipulés ;*
- *un type t représentant la structure ;*
- *une fonction union de signature $t \times elt \times elt \to t$;*
- *une fonction trouve de signature $t \times elt \to elt$;*
- *une fonction initialise de signature $\mathcal{P}_f(elem) \to t$.*

La fonction union est telle que $\forall \mathcal{P} \in t, \forall (x,y) \in elt^2$, $union(\mathcal{P}, x, y)$ calcule le partitionnement obtenu à partir de $\mathcal{P}$ en fusionnant les classes de x et y.

La fonction trouve de signature $t \times elt \to elt$ est telle que $\forall \mathcal{P} \in t, \forall x \in elt$, $find(\mathcal{P}, x)$ calcule un représentant de la classe de x dans la partie $\mathcal{P}$, ainsi $\forall \mathcal{P} \in t, \forall (x,y) \in elt^2$, x est équivalent à y dans $elt^2 \Leftrightarrow find(\mathcal{P}, x) = find(\mathcal{P}, y)$.

La fonction initialise est telle que pour tout ensemble fini S d'éléments de elt, $initialise(S)$ retourne le partitionnement trivial dans lequel chaque élément de S est seul dans sa classe.

Remarque 1.23

On notera equiv de signature $\mathsf{t} \times \mathsf{elt} \times \mathsf{elt} \to \mathbb{B}$, la fonction permettant de tester si deux éléments sont dans la même classe d'équivalence. Cette fonction peut-être définie de la manière suivante.

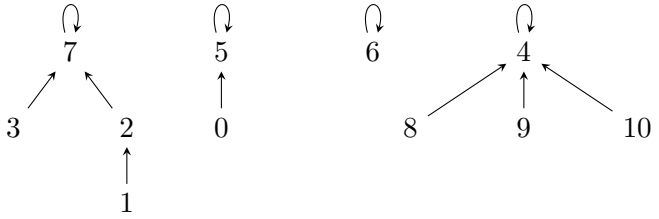
$$\forall \mathcal{P} \in \mathsf{t}, \forall (x,y) \in \mathsf{elt} \times \mathsf{elt}, \mathsf{equiv}(\mathcal{P}, x, y) \overset{\text{déf}}{=} \mathsf{find}(\mathcal{P}, x) \overset{?}{=} \mathsf{find}(\mathcal{P}, y)$$

Dans la suite on suppose que l'ensemble des éléments à représenter est un ensemble d'entiers de la forme $[\![0, n-1]\!]$.

Implémentation au moyen d'une structure arborescente. On met en place une structure de forêt. À chaque élément de S on adjoint un élément de S qui est son père. Ainsi pour chaque élément de S on peut aller visiter son père, puis le père de son père, etc. . .. Afin d'assurer qu'un tel processus termine, le père d'un élément ne peut être son descendant strict. Cependant le père d'un élément peut-être lui-même, auquel cas on dit que cet élément est une **racine**. L'ensemble S étant fini, le parcours de père en père depuis n'importe quel élément x conduit alors nécessairement à un élément racine : c'est le représentant de la classe de x. Attention, les arbres manipulés n'ont donc pas la structures inductive usuelle des arbres. Si un arbre est souvent défini comme un nœud contenant deux fils qui sont eux-mêmes des arbres, ici un nœud contient un pointeur vers son père seulement, il n'a pas accès à ses fils, qui peuvent d'ailleurs être en nombre quelconque (0, 1, 2 ou plus …). On choisit comme représentant canonique de chaque classe la racine de l'arbre représentant la classe.

Exemple 1.24

L'illustration ci-dessous représente la partition de $[\![0, 10]\!]$ en 4 classes : $\{1, 2, 3, 7\}$, $\{0, 5\}$, $\{6\}$ et $\{4, 8, 9, 10\}$.



Ainsi $\{1, 2, 3, 7\}$ admet 7 comme représentant canonique. De même $\{0, 5\}$ (resp. $\{6\}$, resp. $\{4, 8, 9, 10\}$) admet 5 (resp. 6, resp. 4) comme représentant canonique.

Une telle forêt peut aisément être représentée en machine par la donnée du père de chaque élément, stockée par exemple dans un tableau.

Exemple 1.25
On continue l'exemple précédent. La forêt ci-dessus serait représentée en OCaml par le tableau `[|5; 2; 7; 7; 4; 5; 6; 7; 4; 4; 4|]`

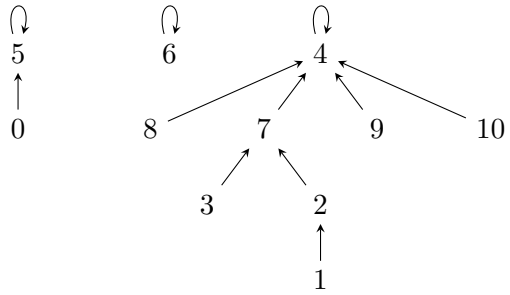
Exercice de cours 1.26

> Donner deux forêts distinctes représentant le partitionnement $\{\{0,3\},\{1\},\{2,8,9\},\{4\},\{5,6,7\}\}$.
> Pour chacune de ces forêts donner un tableau OCaml représentant la forêt en question.

Algorithmes find et union. L'algorithme find peut alors être implémenté en se déplaçant de père en père, depuis l'élément dont on souhaite connaître un représentant. Lorsqu'on atteint un élément qui est son propre père on s'arrête et on le retourne.

L'algorithme union de deux éléments x et y peut alors être implémenté en cherchant a et b les représentants respectifs de x et y (au moyen de deux appels à find), puis à changer le pointeur père de a vers b ou l'inverse.

Exemple 1.27
Ainsi dans l'exemple ci-dessus si l'on souhaite faire l'union de la classe de 1 et de la classe de 8 : on trouve le représentant de 1 (c'est 7), on trouve le représentant de 8 (c'est 4), puis on change le père de 7 pour que ce soit 4 conduisant alors à la forêt ci-dessous.

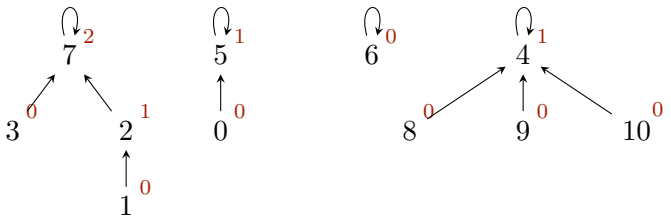


Première amélioration : union par rang. La remontée de père en père dans un arbre est d'autant plus coûteuse (dans le pire cas) que les arbres sont hauts. Aussi dans l'exemple ci-avant, il est particulièrement malheureux d'avoir changer le père de 7 en 4 plutôt que le père de 4 en 7. En effet la profondeur du nœud le plus profond n'aurait pas augmentée dans le second cas (profondeur 2 pour 1) alors que dans le cas représenté ci-dessus on atteint une profondeur de 3 pour le nœud 1. En vue de manipuler des arbres les moins hauts possibles, on souhaite connaître la hauteur des arbres impliqués lors d'une opération d'union. Pour cela, on conserve au niveau de chaque nœud une majoration de la hauteur du sous-arbre qu'il engendre. Plus précisément, on appelle **rang** d'un nœud x une majoration de la hauteur du sous-arbre enraciné en x. C'est-à-dire l'arbre constitué des élément qui admettent x comme ancêtre. Dans l'exemple ci-avant, le nœud étiqueté par la valeur 4 admet 1 comme rang (mais aussi 42). Aussi lors de l'union on utilise l'information de rang pour décider de mettre le nœud de rang inférieur "sous" le nœud de rang supérieur. En cas d'égalité on choisit indifféremment, **mais on n'oublie pas d'incrémenter le rang de la nouvelle racine**.

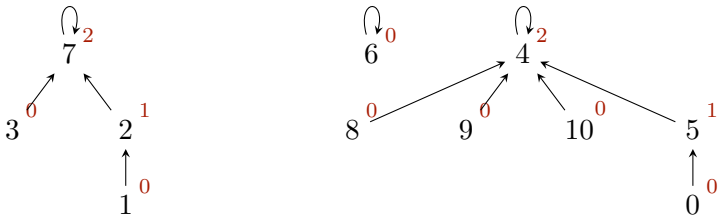
Seconde amélioration : compression de chemins. Lorsqu'on fait une opération find(x) on parcourt les nœuds de x vers la racine de l'arbre contenant x. Cette opération a un coût algorithmique qui est la longueur du chemin entre x et la racine de l'arbre, notons C ce coût. Une fois qu'on a trouvé le représentant de x (notons le r), on peut, pour un coût de l'ordre de C, reparcourir le

chemin de x à r en mettant à jour le pointeur de père vers r. C'est pour cette raison que le rang n'est pas exactement la hauteur des arbres mais bien une sur-approximation : le compression de chemin décroît la hauteur des arbres, sans changer les rangs.

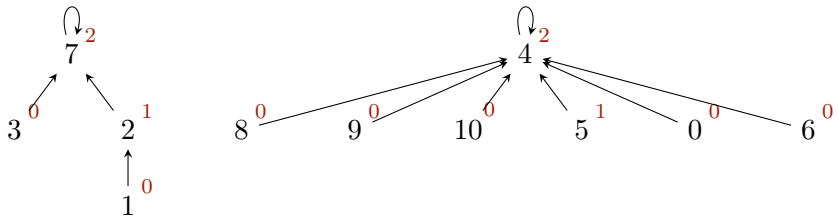
Exemple 1.28
Considérons la structure Union-Find de l'exemple ci-dessus. Les rangs sont indiqués en rouge au dessus à droite des nœuds.



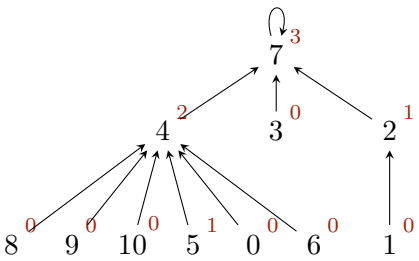
Un appel à union$(0, 8)$ conduit aux appels find$(0) = 5$ et find$(8) = 4$. Ces deux sommets sont de même rang, on choisit indifféremment lequel sera le représentant de la classe : 4. On obtient la structure suivante.



Un appel à union$(0, 6)$ conduit aux appels find$(0) = 4$ et find$(6) = 6$. Le nœud 6 est de rang inférieur, il est donc placé "sous" le nœud 4. Lors de l'appel find(0), on profite d'avoir trouver la racine (4) pour raccourcir les chemins vers la racine. On obtient la structure suivante. Remarquer que le rang de 4 est **strictement** sur-approximant de la profondeur de l'arbre.



Finalement un appel à union$(3, 4)$ conduit aux appels find$(3) = 7$ et find$(4) = 4$. Ces deux nœuds sont de même rang. On choisit indifféremment de placer 4 "sous" 7. On obtient la structure suivante.



Exercice de cours 1.29

Donner l'évolution de la forêt sous-jacente à la structure UnionFind initialisée sur l'ensemble $\{1,2,3,4,5,6,7,8,9\}$, sur laquelle on effectue la suite d'opérations ci-dessous. On pensera à appliquer les deux améliorations ci-dessus (union par rang et compression de chemins). En cas d'ambiguïté (lors d'une fusion de deux sommets de même rang) on choisira comme racine celle de plus petit numéro.

1. union$(1,3)$;
2. union$(5,7)$;
3. union$(7,5)$;
4. union$(5,1)$;
5. union$(7,5)$;
6. union$(4,6)$;
7. union$(8,9)$;
8. union$(4,8)$;
9. union$(6,3)$;

Conclusion. On se convainc que la structure UnionFind présentée ci-avant admet les invariants suivants :
- Le rang du père d'un nœud est toujours strictement supérieur au rang dudit nœud.
- Un nœud de rang p est la racine d'un arbre contenant au moins 2^p nœud.

De ces invariants, on déduit que si une opération find coûte $p \in \mathbb{N}$, c'est que le rang de la racine ainsi obtenue est au moins p et donc l'arbre de taille au moins 2^p. Ou en prenant le raisonnement dans l'autre sens : dans un arbre contenant au plus $n \in \mathbb{N}$ nœuds, il n'est pas possible de faire une opération find coûtant strictement plus de $\log_2(n)$ itérations. Finalement les opérations union et find de la structure proposée ci-avant induisent dans le pire cas des complexités logarithmiques en le nombre d'éléments stockés dans la structure.

Exercice de cours 1.30

Étant donnée une structure UnionFind initialisée à n éléments, on manipule cette structure en effectuant uniquement des opérations d'union sur des éléments **qui ne sont pas déjà dans la même classe**.
Donner des bornes sur le nombre de telles opérations qu'il est possible d'appliquer avant que la structure ne représente le partitionnement dans lequel tous les éléments sont dans la même classe.

Exercice de cours 1.31

Soit $n \in \mathbb{N}$, on considère une structure UnionFind initialisée avec les éléments de $S = \{1,2,3,\ldots,2^n\}$.
Donner une suite de $2^n - 1$ fusions conduisant à un partitionnement dans lequel une des racines est la racine d'un arbre de hauteur n.

1.6 Retour à l'algorithme de Kruskal

Donnons donc la forme finale de l'algorithme de Kruskal.

Algorithme 2 : Algorithme de Kruskal, version suivante

Entrée : Un graphe connexe non orienté pondéré $G = (S, A, c)$
Sortie : Un arbre couvrant de poids minimum

1 $(a_i)_{i\in[\![1,m]\!]} \leftarrow$ une indexation des arêtes par pondération croissante. ;
2 $B \leftarrow \emptyset$;
3 $\mathscr{P} \leftarrow \mathsf{initialise}(S)$;
4 $i \leftarrow 1$;
5 **tant que** $\mathrm{card}(\mathscr{P}) > 1$ **faire**
6 $\quad \{x, y\} \leftarrow a_i$;
7 $\quad$ **si** $\neg\mathsf{equiv}(\mathscr{P}, x, y)$ **alors**
8 $\quad\quad \mathscr{P} \leftarrow \mathsf{union}(\mathscr{P}, x, y)$;
9 $\quad\quad B \leftarrow B \cup \{a_i\}$;
10 $\quad i \leftarrow i + 1$;
11 **retourner** (S, B) ;

Étude de complexité. Notons $(C^n_{\mathsf{union}})_{n\in\mathbb{N}}$ et $(C^n_{\mathsf{find}})_{n\in\mathbb{N}}$ des majorants de la complexité algorithmique des opérations union et find appelés sur des structures contenant n éléments.
Notons de plus $(C^n_{\mathsf{initialise}})_{n\in\mathbb{N}}$ un majorant de la complexité algorithmique de l'opération initialise appelé sur un ensemble de n éléments.
La complexité algorithmique de l'algorithmique de Kruskal sur un graphe contenant n sommets et m arêtes est alors majorée par :

ligne 1 Un tri des m arêtes, induisant un coût de $\mathcal{O}(m\log(m))$.

ligne 3 Initialisation de la structure UnionFind, induisant un coût de $C^n_{\mathsf{initialise}}$.

ligne 5 Une boucle **tant que** effectuant au plus m itérations, la branche **alors** du **si** se trouvant dans le corps de la boucle est emprunté au plus $n-1$ fois.

ligne 7 Deux calculs de représentants dans $\mathscr{P}$, contenant n éléments : $2C^n_{\mathsf{find}}$

ligne 8 Un calcul d'union dans $\mathscr{P}$, contenant n éléments : C^n_{union}

Soit un bilan à :

$$\mathcal{O}(m\log(m) + C^n_{\mathsf{initialise}} + mC^n_{\mathsf{find}} + nC^n_{\mathsf{union}}).$$

En mettant en place la structure UnionFind à base de forêts présentée ci-avant, il est possible de faire descendre la complexité algorithmique en $\mathcal{O}(m\log(m) + n + m\log(n) + n\log(n)) = \mathcal{O}(m\log(n))$ ♣.

Exercice de cours 1.32

Considérons l'implémentation naïve suivante du type de données abstrait UnionFind : le partitionnement est représenté en mémoire par une liste de listes (le partitionnement $\{\{1,2\},\{3,4\},\{5\}\}$ est représenté par la liste `[[1; 2]; [3; 4]; [5]]`), les opérations initialise, union et find sont implémentés au moyen de manipulations de listes.
Quelles sont alors les complexités algorithmiques des fonctions initialise, union et find ? Quel est l'impact sur la complexité algorithmique de l'algorithme de Kruskal ?

♣. On rappelle que $n - 1 \leqslant m \leqslant n^2$ donc $\mathcal{O}(\log(m)) = \mathcal{O}(\log(n))$