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. □

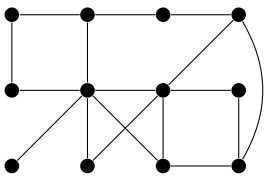
1.2 Arbres couvrants

Définition 1.8

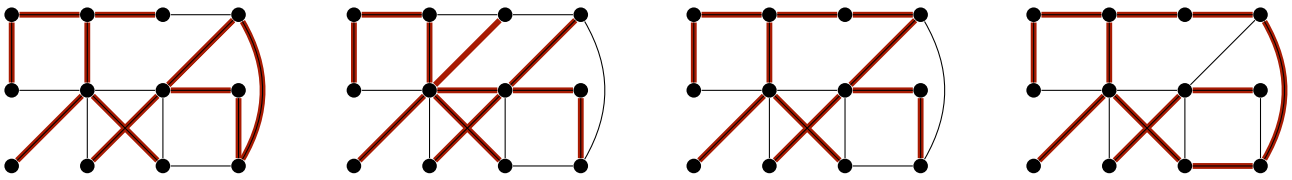
*On dit d'un graphe $G' = (S', A')$ que c'est un **arbre couvrant** de G dès lors que : $S' = S$, $A' \subseteq A$ et G' est un arbre.*

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

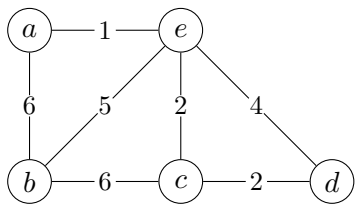
Soit un graphe connexe G à n sommets et m arêtes. Justifier que le nombre d'arbres couvrants de G est fini en donnant, en fonction de n et m, une borne sur le nombre d'arbres couvrants.

1.3 Arbre couvrant et pondération

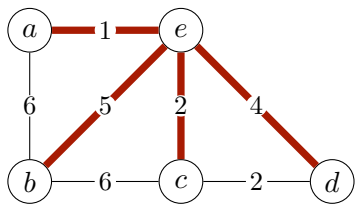
Définition 1.14

*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.

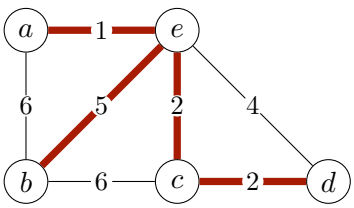
Exemple 1.15



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.

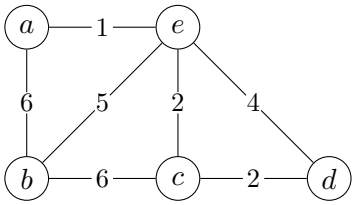
$$\text{ECPM} : \begin{cases} \textbf{Entrée :} \text{ Un graphe connexe non orienté pondéré } G = (S, A, c) \\ \textbf{Sortie :} \text{ Un ensemble d'arêtes } A' \subseteq A \text{ tel que } \sim_A = \sim_{A'} \text{ minimisant } c \end{cases}.$$

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.

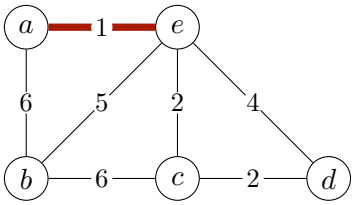
1.4 Algorithme de Kruskal

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).

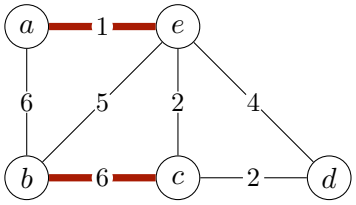
Exemple 1.18
Reprenons l'exemple ci-dessus, et mettons en regard des choix d'arêtes et les partitions en composantes connexes associées.



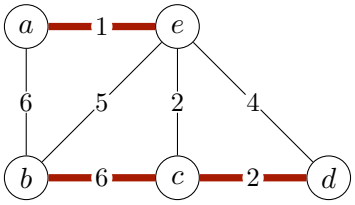
$\{\{a\}, \{e\}, \{c\}, \{b\}, \{d\}\}$



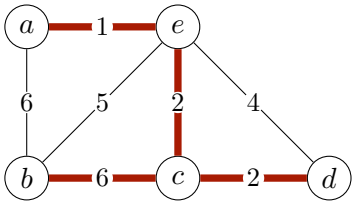
$\{\{a, e\}, \{c\}, \{b\}, \{d\}\}$



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



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



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

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.

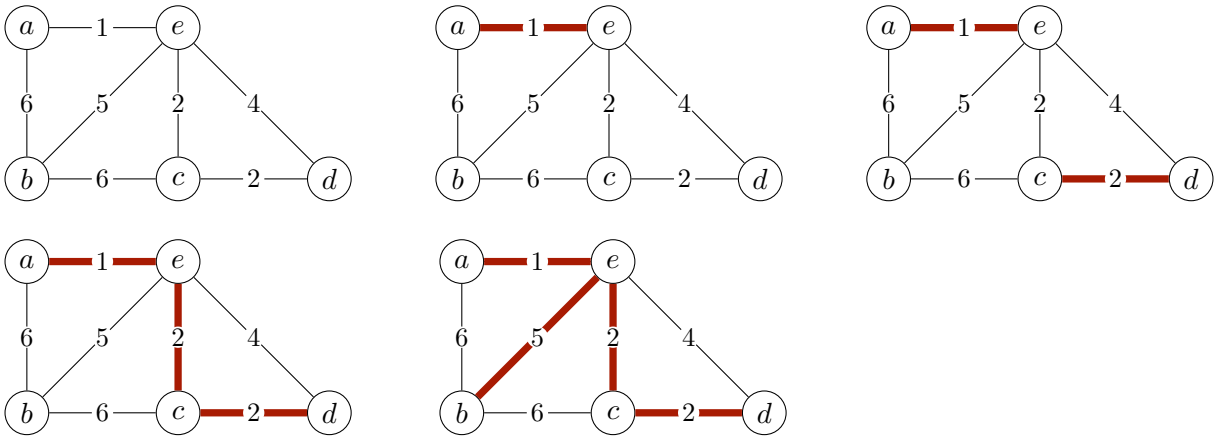
Algorithme 1 : Algorithme de Kruskal (version 0)

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) ;

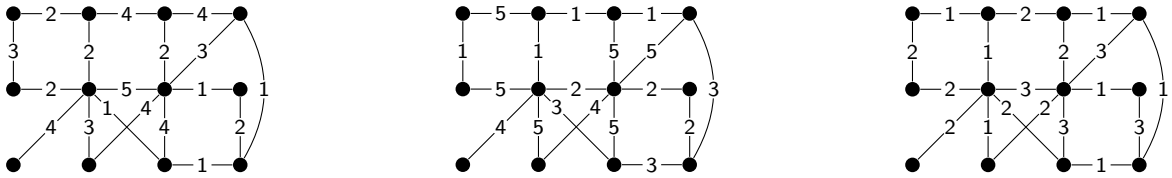
Exemple 1.19

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.

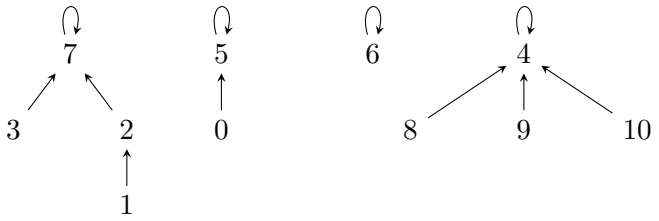
$$\forall \mathcal{P} \in \mathsf{t}, \forall (x,y) \in \mathsf{elt} \times \mathsf{elt}, \mathsf{equiv}(\mathcal{P}, x, y) \overset{\text{déf}}{=} \mathsf{find}(\mathcal{P}, x) \overset{?}{=} \mathsf{find}(\mathcal{P}, y)$$

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|]`

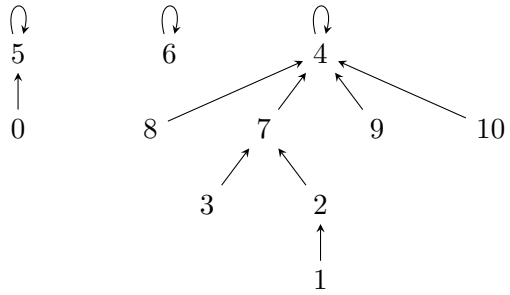
Exercice de cours 1.26

> 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.

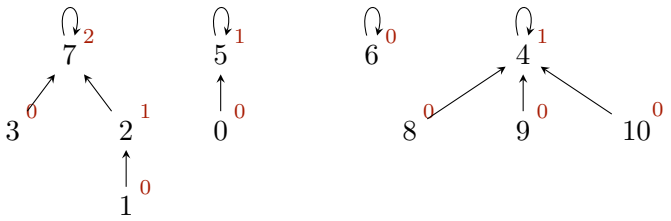


Première amélioration : union par rang. La remontée de père en père dans un arbre est d'autant plus coûteuse (dans le pire cas) que les arbres sont hauts. Aussi dans l'exemple ci-avant, il est particulièrement malheureux d'avoir changer le père de 7 en 4 plutôt que le père de 4 en 7. En effet la profondeur 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**.

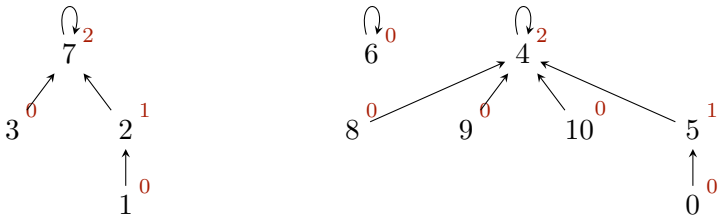
Seconde amélioration : compression de chemins. Lorsqu'on fait une opération find(x) on parcourt les nœuds de x vers la racine de l'arbre contenant x. Cette opération a un coût algorithmique qui est la longueur du chemin entre x et la racine de l'arbre, notons C ce coût. Une fois qu'on a trouvé le représentant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

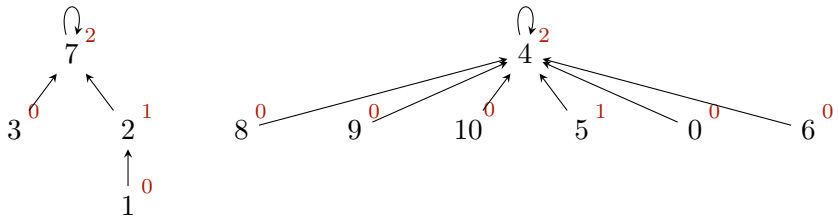
Exemple 1.28
Considérons la structure Union-Find de l'exemple ci-dessus. Les rangs sont indiqués en rouge au dessus à droite des nœuds.



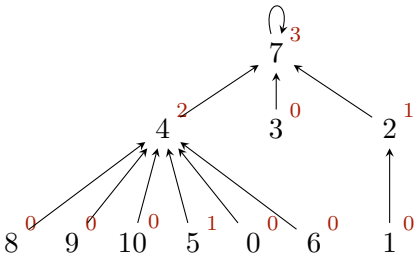
Un appel à union$(0, 8)$ conduit aux appels find$(0) = 5$ et find$(8) = 4$. Ces deux sommets sont de même rang, on choisit indifféremment lequel sera le représentant de la classe : 4. On obtient la structure suivante.



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 arbre dont la racine est de rang p contient au moins 2^p nœuds.

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 la forme finale de l'algorithme de Kruskal.

Algorithme 2 : Algorithme de Kruskal, version avec UnionFind

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

```
1  (a_i)_{i∈⟦1,m⟧} ← une indexation des arêtes par pondération croissante. ;
2  B ← ∅ ;
3  𝒫 ← initialiseUnionFinf(S) ;
4  i ← 1 ;
5  tant que card(𝒫) > 1 faire
6      {x, y} ← a_i ;
7      si find(𝒫, x) ≠ find(𝒫, y) alors
8          𝒫 ← union(𝒫, x, y) ;
9          B ← B ∪ {a_i} ;
10     i ← i + 1 ;
11 retourner (S, B) ;
```

Étude de complexité. Notons $(C^n_{\text{union}})_{n\in\mathbb{N}}$ et $(C^n_{\text{find}})_{n\in\mathbb{N}}$ des majorants de la complexité algorithmique des opérations union et find appelés sur des structures contenant n éléments.
Notons de plus $(C^n_{\text{initialise}})_{n\in\mathbb{N}}$ 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 $C^n_{\text{initialise}}$.

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 $\mathcal{P}$, contenant n éléments : $2C^n_{\text{find}}$

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

Soit un bilan à :

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

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 ?

2 Algorithme de Kosaraju

L'algorithme de Kosaraju est un algorithme permettant le calcul des composantes fortement connexes (notées CFC) d'un graphe orienté. Dans toute cette section, on travaille sur un graphe orienté $G = (S, A)$, on notera $n = \text{card}(S)$ et $m = \text{card}(A)$.

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

Notation 2.1

Pour $(u,v)\in S^2$, on note $u \xrightarrow{*} v$ (resp. $u \xrightarrow{*} v$) s'il existe un chemin (resp. un chemin de longueur k) menant de u à v dans G. On note $u \sim v$ si et seulement si $u \xrightarrow{*} v$ et $u \xrightarrow{*} v$.

2.1 Tri préfixe (rappels)

Rappel 2.2

Soit $T = (T_i)_{i\in[\![1,n]\!]}$ une permutation des sommets de G.
*On définit le **rang** d'un sommet u dans la permutation T, noté $rg_T(u)$, comme étant le plus petit indice d'un élément dans la même CFC que u.*

$$rg_T(u) \overset{\text{déf}}{=} \min\{i \in [\![1,n]\!] \mid T_i \sim_G u\}$$

*On dit que T est un **tri préfixe** de G dès lors que $\forall (u,v)\in S^2, (u,v)\in A \Rightarrow rg_T(u) \leqslant rg_T(v)$.*

Exercice de cours 2.3

Rappeler comment on calcule un tri préfixe d'un graphe, et avec quelle complexité.

Définition 2.4

Pour $(u,v)\in S^2$ deux sommets du graphe :
- *u est **descendant** (resp. **ascendant**) de v si et seulement si $v \xrightarrow{*} u$ (resp. $u \xrightarrow{*} v$) ;*
- *u est **descendant propre** (resp. **ascendant propre**) de v si et seulement si $v \xrightarrow{*} u$ et $u \not\xrightarrow{*} v$ (resp. $u \xrightarrow{*} v$ et $v \not\xrightarrow{*} u$).*

Exercice de cours 2.5

Soit $(u,v)\in S^2$. Soit T un tri préfixe de G.
- Si $u \sim v$, que peut-on dire de $\text{rg}_T(u)$ et $\text{rg}_T(v)$? Peut-on savoir qui de u et v apparaît en premier dans T ?
- Si u est descendant propre de v que peut-on dire de $\text{rg}_T(u)$ et $\text{rg}_T(v)$? Peut-on savoir qui de u et v apparaît en premier dans T ?

2.2 Graphe transposé et CFC

Définition 2.6

*On appelle **graphe transposé** de G le graphe $G^t = (S,B)$ où $B = \{(v,u) \mid (u,v)\in A\}$.*
Autrement dit c'est le graphe obtenu en retournant tous les arcs.

Remarque 2.7

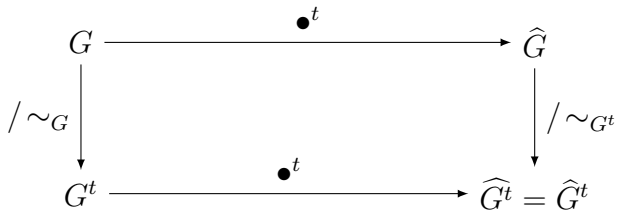
Le nom vient de la représentation matricielle : la matrice d'adjacence du graphe transposé est la transposée de la matrice d'adjacence du graphe initial.

Exercice de cours 2.8

Donner le pseudo-code d'un algorithme de calcul du graphe transposé en complexité linéaire, pour une représentation par matrice d'adjacence, puis pour une représentation par table de listes d'adjacence.

Proposition 2.9

- *La relation $\sim$ définie par la mutuelle accessibilité dans G est la même que celle définie de même dans G^t. Par conséquent les CFC de G sont les mêmes que celles de G^t.*
- *Le graphe réduit du transposé est le transposé du graphe réduit, i.e. $\widehat{G^t} = \widehat{G}^t$.*
  Autrement dit le passage au quotient selon $\sim$ et le retournement des arcs commutent.



Exercice de cours 2.10

Démontrer le second point de la proposition 2.9.

Remarque 2.11

Soit $(u, v) \in S^2$.
u est ascendant (propre) de v dans G si et seulement si u est descendant (propre) de v dans G^t.
u est descendant (propre) de v dans G si et seulement si u est ascendant (propre) de v dans G^t.

Vocabulaire 2.12

Soit T une permutation des sommets. Soit L un parcours de G^t.
*On dit que les points de régénération sont **choisis prioritairement selon** T si les points de régénération de L apparaissent dans le même ordre dans T et dans L.*
De manière équivalente, pour tout $r \in [\![1, n]\!]$ tel que L_r est point de régénération de L, $L_r = T_{i_0}$ avec $i_0 = \min \{i \in [\![1, n]\!] \mid T_i \notin \{L_j \mid j \in [\![1, r[\![\}\}$.

Lemme 2.13

Soit T une permutation des sommets de G. Soit L un parcours de G. Si les points de régénération de L sont choisis prioritairement selon T, alors pour tout L_r point de régénération de L, pour tout $i \geqslant r$, $\mathrm{rg}_T(L_r) \leqslant \mathrm{rg}_T(L_i)$.

Démonstration : Par l'absurde, on suppose qu'il existe $i \geqslant r$ tel que $\mathrm{rg}_T(L_i) < \mathrm{rg}_T(L_r)$. On peut alors considérer v le premier sommet dans T tel que $v \sim L_i$, v apparaît à l'indice $\mathrm{rg}_T(L_i)$ dans T donc avant L_r. Étant donné que L_i n'est pas visité au moment où L_r est choisi comme point de régénération, et que $v \xrightarrow{*} L_i$, v n'a pas non plus été visité à ce moment là ♣ et donc v aurait dû être choisi. □

Proposition 2.14

Soit T un tri préfixe de G. Soit L un parcours de G^t.
Si les points de régénération de L sont choisis prioritairement selon T, alors la partition associée à L est la décomposition en CFC de G^t et G.

Démonstration : Puisque G et G^t ont les mêmes CFC, on ne précisera donc pas toujours de quelles CFC on parle. On sait déjà que deux sommets de la même CFC sont nécessairement dans la même partie selon L (voir chapitre sur les parcours). Il reste à montrer que deux sommets dans la même partie selon L sont

♣. par des arguments de bordure non vide déjà développés dans le chapitre sur les parcours

nécessairement de la même CFC, autrement dit qu'ils sont mutuellement accessibles l'un depuis l'autre. On reprend les notations de la définition du partitionnement associé à un parcours :

- K le nombre de points de régénération de L ;
- $(r_k)_{k\in[\![1,K]\!]} \in [\![1,n]\!]^K$ les indices de ces points de régénération, en ordre strictement croissant ;
- $r_{K+1} = n+1$;
- $\forall k \in [\![1,K]\!]$, $C_k = \{L_j \mid j \in [\![r_k, r_{k+1}[\![\}$.

Par l'absurde supposons qu'il existe $(u,v) \in S^2$ appartenant à la même partie C_k et tels que $u \not\sim v$. Puisque L est un parcours de G^t, on sait que u et v sont tous les deux accessibles depuis L_{r_k} dans G^t (par des arguments de bordure non vide déjà développés dans le chapitre sur les parcours).

On en déduit que, dans G^t, L_{r_k} n'est pas accessible depuis u ou pas accessible depuis v (sinon en concaténant les chemins on aurait $u \sim v$). Quitte à échanger u et v, on suppose que L_{r_k} n'est pas accessible depuis u dans G^t. Ainsi u est un descendant propre de L_{r_k} dans G^t, autrement dit u est un ascendant propre de L_{r_k} dans G. Puisque T est un tri préfixe de G on en déduit $\mathsf{rg}_T(u) < \mathsf{rg}_T(L_{r_k})$.

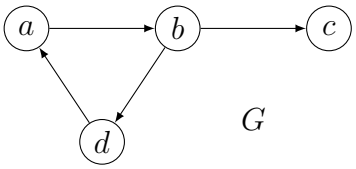
Par ailleurs, en appliquant 2.13 à T et L (respectivement permutation des sommets et parcours du graphe G^t) : $\mathsf{rg}_T(L_{r_k}) \leqslant \mathsf{rg}_T(u)$.

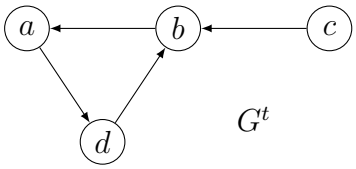
Absurde. On en déduit que deux sommets appartenant à une même partie selon L sont nécessairement dans la même CFC. □

Remarque 2.15

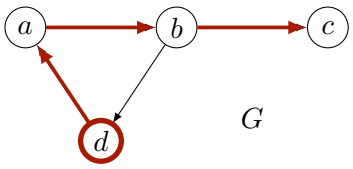
> Attention il faut bien passer au graphe transposé et le parcourir en choisissant les points de régénération selon un tri préfixe, rester sur le même graphe avec le miroir d'un tri préfixe ne suffit pas.

Démonstration : On considère le graphe G ci-dessous à gauche (on donne ci-dessous à droite G^t son graphe transposé). Notons $T = [a,b,c,d]$, ainsi T est un tri préfixe de G, et $\overleftarrow{T} = [d,c,b,a]$ son miroir.

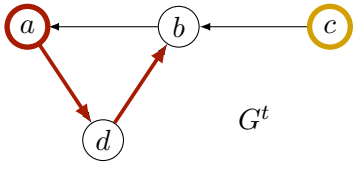




Si on fait un parcours du graphe G en choisissant les points de régénération selon $\overleftarrow{T} = [d,c,b,a]$, on obtient le parcours $[\underline{d}, a, b, c]$, et donc la partition $\{\{a,b,c,d\}\}$. Or ce n'est pas la décomposition en CFC puisque a et c par exemple sont dans la même partie, alors que $c \not\overset{*}{\to} a$. En effet, dans G on peut aller de b à c mais pas de c à b, mais on ne s'en rend pas compte ici car on visite b avant c ….



Si on fait un parcours du graphe G^t en choisissant les points de régénération selon le tri préfixe $T = [a,b,c,d]$, on obtient $\{\{\underline{a}, d, b\}, \{\underline{c}\}\}$. Cette décomposition est bien la décomposition en CFC. En effet ici, le fait qu'on ne puisse pas aller de c à b dans G, qui se traduit par l'impossibilité d'aller de b à c dans G^t, a forcé le parcours à prendre un nouveau point de régénération entre b et c.



□

2.3 Algorithme de Kosaraju

D'après la propriété précédente on peut décomposer un graphe orienté en CFC avec n'importe quel parcours pourvu qu'on choisisse les points de régénération (y compris le premier point du parcours)

selon un tri préfixe. De plus on a vu précédemment qu'on peut construire un tri préfixe par un parcours en profondeur. L'algorithme de Kosaraju propose donc d'appliquer deux fois la même routine de parcours en profondeur : la première fois sur le graphe G avec des points de régénérations arbitraires afin d'obtenir T un tri préfixe de G, la seconde fois sur le graphe G^t en choisissant les points de régénération selon l'ordre établi par T. La partition associée à ce second parcours fournit alors la décomposition en CFC.

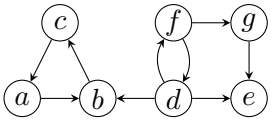
Algorithme 2 : Algorithme de Kosaraju

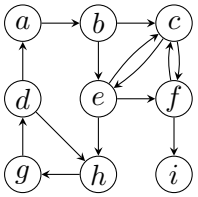
Entrée : Un graphe orienté $G = (S, A)$
Sortie : La décomposition de G en CFC

1 On calcule L un tri préfixe de G ;
2 On parcourt G^t, suivant l'ordre induit par L ;
3 On retourne le partitionnement associé à ce parcours ;

Exercice de cours 2.16

Appliquer l'algorithme de Kosaraju aux graphes ci-dessous.





Proposition 2.17

La complexité♣ de l'algorithme de Kosaraju est en $O(n + m)$.

Démonstration : On représente les graphes par tables de listes d'adjacence, ainsi les deux parcours sont en $O(n + m)$. Le calcul de G^t se fait aussi en $O(n + m)$ (*Cf.* exercice de cours 2.8). □

Exercice de cours 2.18

On décide d'améliorer l'algorithme de Kosaraju en profitant du premier parcours pour tester si le graphe est, ou non, sans circuit. Expliquer en quoi cette information est pertinente.

2.4 Application à 2-SAT

Cette section établit un corollaire de l'existence de l'algorithme de Kosaraju, et plus généralement de l'existence d'un algorithme de complexité polynomiale permettant le calcul des composantes fortement connexes d'un graphe orienté. On rappelle que le problème de décision 2-SAT est défini comme suit.

2-SAT : $\begin{cases} \textbf{Entrée :} \text{ Une formule } H \in \mathbb{F}_p(\mathcal{Q}) \text{ donnée comme conjonction de 2-clauses} \\ \textbf{Sortie :} \ H \text{ est-elle satisfiable ?} \end{cases}$

Corollaire 2.19

2-SAT $\in$ P.

Démonstration : Soit $\mathcal{Q}$ un ensemble de variables propositionnelles. Soit $H = (l_{1,1} \vee l_{1,2}) \wedge \ldots \wedge (l_{t,1} \vee l_{t,2})$ une instance de 2-SAT. Dans la suite de cette preuve, lorsque l, est un littéral, on note l^c le littéral opposé. On remarque que lorsqu'un littéral $l_{i,1}$ est interprété à F, $l_{i,2}$ doit nécessairement être interprété à V pour que H soit satisfaite (car la clause $(l_{i,1} \vee l_{i,2})$ doit en particulier l'être).

♣. Sous réserve que l'ensemble des sommets du graphe considéré soit de la forme $[\![1, n]\!]$

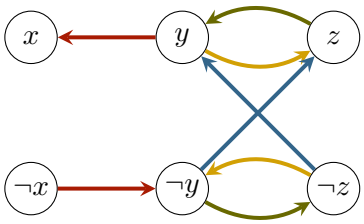
Ainsi si $l^c_{i,1}$ est interprété à V, $l_{i,2}$ doit aussi être interprété à V. De même $l^c_{i,2}$ est interprété à V, $l_{i,1}$ doit aussi être interprété à V.

On traduit cette dépendance au moyen d'un graphe orienté sur les littéraux. Plus précisément, on construit le graphe G_H de la manière suivante :

- l'ensemble de ses sommets est l'ensemble des littéraux sur $\mathcal{Q}$, *i.e.* $S = \mathcal{Q} \cup \{\neg p \mid p \in \mathcal{Q}\}$;
- l'ensemble de ses arcs est $A = \{(l^c_{i,1}, l_{i,2}) \mid i \in [\![1,t]\!]\} \cup \{(l^c_{i,2}, l_{i,1}) \mid i \in [\![1,m]\!]\}$.

Exemple 2.20

Par exemple la formule $(x \vee \neg y) \wedge (\neg y \vee z) \wedge (y \vee \neg z) \wedge (y \vee z)$, le graphe associé est le suivant.



Le graphe G_H est construit de manière à assurer le lemme suivant.

Lemme 2.21

Soit $(u,v) \in S^2$. Si $u \overset{*}{\to} v$ dans G_H et si ρ est un modèle de H ♣ tel que $[\![u]\!]^\rho = V$ alors $[\![v]\!]^\rho = V$.

Démonstration : On le montre par récurrence sur la longueur du chemin menant d'un sommet à l'autre.

- Soit $(u,v) \in S^2$ tels que $u \overset{0}{\to} v$. Dans ce cas $u = v$, donc $[\![u]\!]^\rho = [\![v]\!]^\rho = \mathsf{V}$.
  Ainsi la propriété est vraie au rang 0.
- Soit $k \in \mathbb{N}$. Supposons la propriété est vraie au rang k. Soit $(u,v) \in S^2$ tels que $u \overset{k+1}{\to} v$. Soit ρ soit un modèle de H tel que $[\![u]\!]^\rho = \mathsf{V}$. Puisque $u \overset{k+1}{\to} v$, il existe $w \in S$ tel que $u \overset{k}{\to} w$ et $w \overset{1}{\to} v$. Comme $u \overset{k}{\to} w$ et $[\![u]\!]^\rho = \mathsf{V}$, on a aussi $[\![w]\!]^\rho = \mathsf{V}$ par hypothèse de récurrence. De plus comme $(w,v) \in A$, $w^c \vee v$ ou $v \vee w^c$ est une clause de H, et donc satisfaite par ρ. Ainsi $[\![w^c]\!]^\rho + [\![v]\!]^\rho = \mathsf{V}$, or $[\![w^c]\!]^\rho = \overline{[\![w]\!]^\rho} = \overline{\mathsf{V}} = \mathsf{F}$, on en déduit que $[\![v]\!]^\rho = \mathsf{V}$. La propriété est donc vraie au rang $k+1$.

□

Proposition 2.22

H est satisfiable si et seulement si aucune CFC de G_H ne contient à la fois une variable et sa négation.

Démonstration :

⇒ Si H est satisfiable, on dispose alors d'un environnement propositionnel $\rho \in \mathbb{B}^{\mathcal{Q}}$ tel que $[\![H]\!]^\rho = \mathsf{V}$.
Supposons par l'absurde qu'il existe une CFC du graphe G_H contenant les littéraux x et $\neg x$.
Par définition d'une CFC, $x \overset{*}{\to} \neg x$ et $\neg x \overset{*}{\to} x$.
- Si $\rho(x) = \mathsf{V}$ alors $[\![x]\!]^\rho = \mathsf{V}$. Puisque $x \overset{*}{\to} \neg x$, on déduit du lemme 2.21 que $[\![\neg x]\!]^\rho = \mathsf{V}$ soit $\rho(x) = \mathsf{F}$.
- Si $\rho(x) = \mathsf{F}$ alors $[\![\neg x]\!]^\rho = \mathsf{V}$. Puisque $\neg x \overset{*}{\to} x$, on déduit du lemme 2.21 que $[\![x]\!]^\rho = \mathsf{V}$ soit $\rho(x) = \mathsf{V}$.

Les deux cas étant absurdes, aucune CFC de G_H ne contient une variable et sa négation.

⇐ Réciproquement supposons qu'aucune CFC de G_H ne contient une variable et sa négation.
Montrons que H est alors satisfiable en exhibant un modèle.
Notons $(C_1, C_2, \ldots, C_s)$ un tri topologique du graphe réduit de G_H. On peut définir l'environnement $\rho : \mathcal{Q} \to \mathbb{B}$ par $\rho(x) = \mathsf{V}$ si et seulement si $i < j$ où $(i,j) \in [\![1,s]\!]^2$ sont tels que $\neg x \in C_i$ et $x \in C_j$.
Ainsi pour toute variable $x \in \mathcal{Q}$, si $[\![x]\!]^\rho = \mathsf{V}$ alors x est dans une CFC d'indice strictement supérieur à la CFC contenant $\neg x$. Étant donné qu'aucune variable n'apparaît dans la même CFC que sa négation, si $[\![x]\!]^\rho = \mathsf{F}$ alors x est dans une CFC d'indice strictement inférieur à la CFC contenant $\neg x$. Autrement

♣. Cela signifie que ρ est un environnement propositionnel tel que $[\![H]\!]^\rho = \mathsf{V}$

dit, si $[\![\neg x]\!]^\rho = \mathsf{V}$ alors $\neg x$ est dans une CFC d'indice strictement supérieur à la CFC contenant x. Ainsi pour tout littéral l, si $[\![l]\!]^\rho = \mathsf{V}$ alors l^c est dans une CFC d'indice strictement inférieur à la CFC contenant l.
Supposons alors par l'absurde qu'il existe une clause $l_{i,1} \vee l_{i,2}$ de H telle que $[\![l_{i,1} \vee l_{i,2}]\!]^\rho = \mathsf{F}$. Notons alors :

- k_1 l'indice de la CFC de $l_{i,1}$;
- k_2 l'indice de la CFC de $l_{i,2}$;
- k'_1 l'indice de la CFC de $l^c_{i,1}$;
- k'_2 l'indice de la CFC de $l^c_{i,2}$.

Puisque $[\![l_{i,1} \vee l_{i,2}]\!]^\rho = \mathsf{F}$, $[\![l_{i,1}]\!]^\rho = \mathsf{F}$ et $[\![l_{i,2}]\!]^\rho = \mathsf{F}$, soit $[\![l^c_{i,1}]\!]^\rho = \mathsf{V}$ et $[\![l^c_{i,2}]\!]^\rho = \mathsf{V}$, donc par construction de ρ :

- $l_{i,1} = (l^c_{i,1})^c$ est dans une CFC d'indice strictement inférieur à la CFC contenant $l^c_{i,1}$, soit $k_1 < k'_1$;
- $l_{i,2} = (l^c_{i,2})^c$ est dans une CFC d'indice strictement inférieur à la CFC contenant $l^c_{i,2}$, soit $k_2 < k'_2$.

Pourtant, puisque $l_{i,1} \vee l_{i,2}$ est une clause de H, le graphe G_H contient l'arc $(l^c_{i,1}, l_{i,2})$ et l'arc $(l^c_{i,2}, l_{i,1})$, qui impliquent respectivement que $k'_1 \leqslant k_2$ et $k'_2 \leqslant k_1$. Mises bout à bout ces quatre inégalités donnent : $k_1 < k'_1 \leqslant k_2 < k'_2 \leqslant a$. Absurde Finalement toute clause $l_{i,1} \vee l_{i,2}$ de H est satisfaite par ρ, donc ρ est bien modèle de H et H satisfiable.

□

L'algorithme suivant résout donc 2-Sat et il est de complexité polynomiale.

Algorithme 3 : Satisfiabilité d'une 2-CNF

Entrée : Une 2-CNF H
Sortie : H est-elle satisfiable ?

1 Construire le graphe G_H associé à la formule H ;
2 Calculer les CFC de G_H ;
3 Tester si aucune variable et sa négation ne sont dans la même CFC ;

□

Exercice de cours 2.23

Justifier plus précisément pourquoi cet algorithme est de complexité polynomiale.

Remarque 2.24

On remarque que la preuve précédente fournit en fait un algorithme permettant non seulement de tester si une 2-CNF est satisfiable, mais aussi d'en construire un modèle.

Exercice de cours 2.25

Construire un modèle de la formule de l'exemple 2.4 en suivant la construction de la preuve précédente (*i.e.* donner d'abord les CFC du graphe, puis donner un tri topologique du graphe réduit avant de donner fixer la valeur de vérité pour chaque variable).

Exercice de cours 2.26

Donner un modèle de $(\neg y \vee x) \wedge (y \vee \neg z) \wedge (y \vee \neg z) \wedge (y \vee z) \wedge (\neg y \vee z)$.

Remarque 2.27

On remarque que la résolution d'une instance H de 2-Sat dépend seulement de l'ensemble des paires de littéraux apparaissant dans la formule. En effet si on change l'ordre des clauses, leur multiplicité ou l'ordre de deux littéraux au sein d'une clause, on e change pas le graphe G_H associé à la formule H, ainsi on ne change pas la manière de décider si H est satisfiable (ni celle, le cas échéant, de trouver un modèle de H).

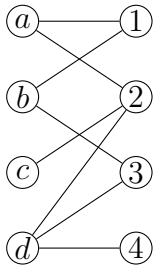
Exercice de cours 2.28

Appliquer l'algorithme précédent aux deux instances 2-SAT suivantes (représentées ici par l'ensemble de leurs clauses) $\Gamma_1 = \{\neg p \vee q, \neg q \vee r, \neg q \vee s, \neg s \vee \neg r\}$ et $\Gamma_2 = \Gamma_1 \cup \{p \vee q\}$.

3 Couplage dans un graphe biparti

3.1 Introduction

Considérons le problème suivant. On a des plantes et des pots, et on sait quelle plante rentre dans quel pot. On cherche à placer un maximum de plantes (ou à utiliser un maximum de pot, ce qui revient dans les deux cas à former un maximum d'appariement plante-pot) sachant qu'une plante ne pourra être placée que dans un seul pot, et chaque pot ne pourra contenir qu'une plante, bouture et permaculture étant remises à plus tard. On peut alors modéliser les données du problème par un graphe non orienté biparti comme le graphe ci-contre.



Graphe plante-pot

Rappel 3.1

*Un graphe non orienté $G = (S, A)$ est dit **biparti** lorsqu'il existe une partition $\{S_1, S_2\}$ de S telle que pour toute arête $\{x, y\} \in A$, ($x \in S_1$ et $y \in S_2$) ou ($x \in S_2$ et $y \in S_1$).*

Exercice de cours 3.2

Démontrer que dans un graphe biparti les cycles ne peuvent être de longueur impaire.

Exemple 3.3
Le graphe ci-dessus est biparti avec $S_1 = \{a, b, c, d\}$ et $S_2 = \{1, 2, 3, 4\}$.

Remarque 3.4

Il peut exister plusieurs partitions justifiant qu'un graphe G est biparti.

Notation 3.5

*Lorsqu'on écrit que $G = (S_1 \sqcup S_2, A)$ est biparti, on sous-entend qu'il s'agit du graphe $G = (S, A)$ avec $S = S_1 \cup S_2$ **et** qu'il est biparti pour $\{S_1, S_2\}$, autrement dit que les arêtes de A ont une extrémité dans S_1 et l'autre dans S_2.*

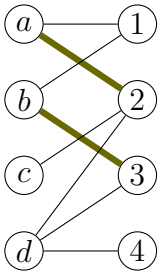
Définition 3.6

Soit $G = (S, A)$ un graphe non orienté. Soit $C \subseteq A$ un sous-ensemble d'arêtes.
*On dit que C est un **couplage** de G si et seulement si $\forall (e, e') \in C^2, e \cap e' \neq \emptyset \Rightarrow e = e'$.*
Autrement dit C est un couplage si deux arêtes distinctes de C n'ont aucune extrémité en commun.

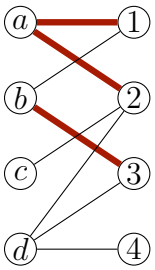
Exemple 3.7

Exercice de cours 3.8

Si C est un couplage et $C' \subseteq C$, que peut-on dire de C' ?



Couplage du graphe plante-pot



Non-couplage du graphe plante-pot

Exercice de cours 3.9

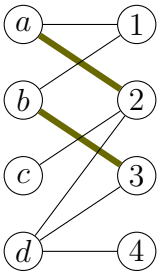
Donner un majorant du nombre d'arêtes dans un couplage d'un graphe à n sommets.
Donner un majorant du nombre d'arêtes dans un couplage d'un graphe biparti $G = (S_1 \sqcup S_2, A)$ en fonction de $n_1 = |S_1|$ et $n_2 = |S_2|$.

Représentation machine Les arêtes d'un couplage définissant un graphe dont les sommets sont de degré 0 ou 1, on peut le représenter en machine par un tableau indexé par les sommets qui associe à chaque sommet son voisin s'il en a un, et une valeur par défaut sinon.

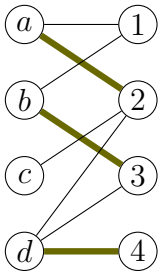
Définition 3.10

Soit $G = (S, A)$ un graphe non orienté. Soit $C \subseteq A$ un couplage de G.
*Le couplage C est dit **maximal** s'il n'existe pas de couplage de G strictement plus grand pour $\subseteq$.*
Autrement dit C est maximal si et seulement si $\forall C' \subseteq A, C'$ couplage de G et $C \subseteq C' \Rightarrow C = C'$.
*Le couplage C est dit **maximum** s'il est de cardinal maximal parmi les couplages de G. Autrement dit C est maximum si et seulement si $\forall C' \subseteq A, C'$ couplage de $G \ \Rightarrow |C| \geqslant |C'|$.*

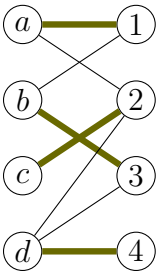
Exemple 3.11



Couplage □
ni maximum ni maximal



Couplage ★
maximal non maximum



Couplage ♢
maximal et maximum

Remarque 3.12

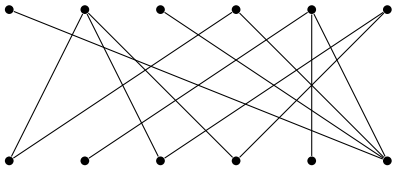
Un couplage maximum est nécessairement maximal. En effet, s'il existait un couplage C' strictement plus grand pour $\subseteq$ qu'un couplage maximum C, on aurait $\mathrm{card}(C') > \mathrm{card}(C)$, Absurde.

Remarque 3.13

L'ensemble vide est un toujours un couplage, de plus l'ensemble des couplages est inclus dans $\mathscr{P}(A)$ qui est fini, ainsi l'ensemble des couplages est un ensemble fini non vide ce que justifie qu'il existe nécessairement un couplage maximum, et donc a fortiori il existe un couplage maximal.

Exercice de cours 3.14

Donner un couplage maximal, non maximum et un couplage maximum du graphe biparti ci-dessous.



Exercice de cours 3.15

Soit C un couplage d'un graphe $G = (S, A)$. Démontrer que C est maximal si et seulement si on ne peut y ajouter la moindre arête, *i.e.* $\forall e \in A \setminus C$, $C \cup \{a\}$ n'est pas un couplage.

Remarque 3.16

L'algorithme glouton qui construit un couplage par ajouts successifs (*i.e.* qui sélectionne à chaque étape une arête dont aucune extrémité n'est couverte par le couplage courant) conduit à un couplage maximal, mais pas nécessairement maximum (*Cf.* exercice de cours 3.14). Ainsi cet algorithme peut être vu comme un algorithme de recherche locale qui reste parfois bloqué sur un maximum local (un couplage maximal est maximum dans son voisinage défini par la relation $\subseteq$). Pour sortir d'un tel maximum local, il faut alors envisager une autre opération que l'ajout d'arête. Il faut en fait accepter d'enlever certaines arêtes pour en remettre plus, c'est ce que fait l'opération d'inversion le long d'un chemin que l'on introduit à la section suivante.

3.2 Chemin augmentant

Définition 3.17

Soit un graphe $G = (S, A)$ un graphe non orienté. Soit C un couplage de G.

- *On dit d'un sommet $x \in S$ qu'il est* ***libre*** *pour C s'il n'est l'extrémité d'aucune arête du couplage, i.e. $\forall \{y, z\} \in C, x \neq y$ et $x \neq z$.*
- *On dit qu'une chaîne $c = (\gamma_0, \gamma_1, \ldots, \gamma_{2p+1})$ de longueur impaire qu'elle est* ***augmentante*** ♣ *pour C si :*
  - *γ_0 et γ_{2p+1} sont libres pour C ;*
  - *pour tout $i \in [\![0, p]\!]$, $\{\gamma_{2i}, \gamma_{2i+1}\} \in A \setminus C$;*
  - *pour tout $i \in [\![0, p-1]\!]$, $\{\gamma_{2i+1}, \gamma_{2i+2}\} \in C$;*
  - *c est élémentaire, i.e. $\forall (i, j) \in [\![0, 2p+1]\!]^2$, $\gamma_i = \gamma_j \Rightarrow i = j$.*

  Autrement dit une chaîne augmentante est une chaîne élémentairede longueur impaire, alternant les arêtes de C et de $A \setminus C$ dont les deux extremités sont deux sommets libres.

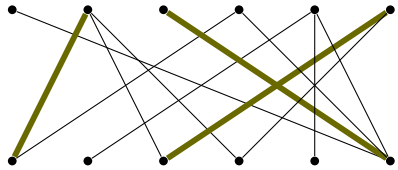
Exemple 3.18

Dans le couplage □ ci-dessus, $(d, 4)$ est un chemin augmentant.
Dans le couplage ★ ci-dessus, $(c, 2, a, 1)$ est un chemin augmentant.

♣. Par abus on parlera par la suite de **chemin augmentant** bien qu'il s'agisse d'une chaîne.

Exercice de cours 3.19

Dans le graphe biparti ci-dessous donner un chemin augmentant de longueur 1, de longueur 3 et un de longueur 5 pour le couplage —.

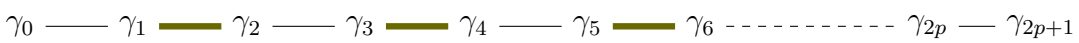


Lemme 3.20

Soit $G = (S, A)$ un graphe non orienté. Soit C un couplage de G.
Si $\gamma = (\gamma_0, \gamma_1, \ldots, \gamma_{2p+1})$ un chemin augmentant pour C,
alors l'ensemble C' défini ci-dessous est un couplage de G de cardinal $|C| + 1$.

$$C' = C \setminus \{\{\gamma_{2i+1}, \gamma_{2i+2}\} \mid i \in [\![0, p-1]\!]\} \cup \{\{\gamma_{2i}, \gamma_{2i+1}\} \mid i \in [\![0, p]\!]\}$$

Démonstration : Par construction de C', $|C'| = |C| - p + (p+1)$, soit $|C'| = |C| + 1$. Reste à justifier que C' est bien un couplage.



Les sommets hors de γ ayant les mêmes arêtes incidentes dans C et dans C' (soit une au plus puisque C est un couplage), il nous suffit d'inspecter les sommets de γ et de justifier que chacun d'eux est l'extrémité d'au plus une arête de C'.

- γ_0 est libre pour C, donc les seules arêtes de C' susceptibles de lui être incidentes sont celles qu'on a ajoutées, soit celles de la forme $\{\gamma_{2i}, \gamma_{2i+1}\}$. Les sommets de C étant deux à deux distincts, la seule arête de cette forme incidente à γ_0 est $\{\gamma_0, \gamma_1\}$.
- γ_{2p+1} est libre pour C, et de même la seule arête de C' qui lui est incidente est $\{\gamma_{2p}, \gamma_{2p+1}\}$.
- Pour $k \in [\![1, 2p]\!]$, γ_i est l'extrémité d'exactement une arête de C, (à savoir $\{\gamma_{k-1}, \gamma_k\}$ si k est pair et $\{\gamma_k, \gamma_{k+1}\}$ sinon), or celle-ci est supprimée dans C' et une seule autre arête d'extrémité γ_k est ajoutée dans C' (à savoir $\{\gamma_k, \gamma_{k+1}\}$ si k est pair et $\{\gamma_{k-1}, \gamma_k\}$ sinon), donc γ_k est l'extrémité d'exactement une arête de C'.

□

Vocabulaire 3.21

Dans le cadre de ce cours, on dira que C' est le couplage obtenu à partir de C ***par inversion le long de*** *γ.*

Proposition 3.22

Soit $G = (S, A)$ un graphe non orienté. Soit C un couplage de G.
Si C est maximum alors il n'admet pas de chemin augmentant.

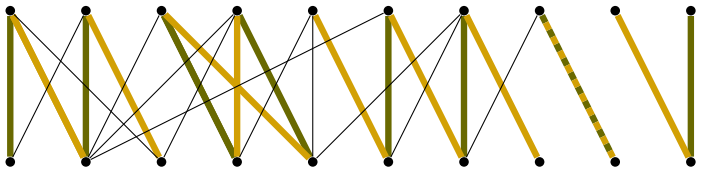
Démonstration : C'est un corollaire du lemme précédent. Si C admet un chemin augmentant γ, alors le couplage C' obtenu en inversant C le long de γ est de cardinal $|C| + 1$, et donc C n'est pas de cardinal maximum. On conclut par contraposée. □

Proposition 3.23

Soit $G = (S, A)$ un graphe non orienté. Soit C un couplage de G.
Si C n'admet pas de chemin augmentant, alors C est un couplage maximum.

Démonstration : On le montre par contraposée. Soit M un couplage de G non maximum. Montrons qu'il admet un chemin augmentant. Considérons pour cela $M^\star$ un couplage maximum.

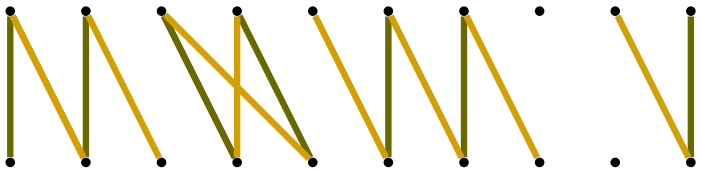
Nous illustrons la preuve sur le graphe biparti ci-dessous où le couplage M est l'ensemble — et le couplage $M^\star$ est l'ensemble —.



Posons alors :

- $D_1 = M \setminus M^\star$, ainsi $D_1 \subseteq M$;
- $D_2 = M^\star \setminus M$, ainsi $D_2 \subseteq A \setminus M$;
- $D = D_1 \sqcup D_2$, ainsi $D = M \Delta M^\star$;
- $G_D = (S, D)$.

Dans la suite nous ne représentons plus que les arêtes de D. Nous représentons D_1 par la couleur de M (—) et D_2 par la couleur de $M^\star$ (—), nous obtenons alors le graphe ci-dessous.



On peut alors faire plusieurs remarques.

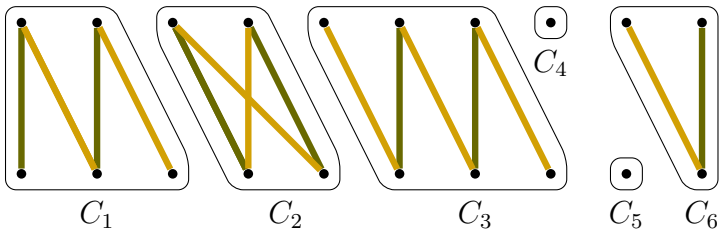
(a) D_1 (resp. D_2) est un couplage en tant que sous-ensemble du couplage M (resp. $M^\star$). Ainsi un sommet G_D est l'extrémité d'au plus une arête de D_1 et au plus une de D_2, et donc est en particulier de degré au plus 2.

(b) La remarque précédente assure aussi que les chaînes de D alternent nécessairement arête de $D_1 \subseteq M$ et de $D_2 \subseteq A \setminus M$.

(c) Puisque $M^\star$ est maximum mais pas M, $\text{card}(M) < \text{card}(M^\star)$ et par conséquent $\text{card}(D_1) < \text{card}(D_2)$. En effet $\text{card}(D_1) = \text{card}(M) - \text{card}(M \cap M^\star) < \text{card}(M^\star) - \text{card}(M \cap M^\star) = \text{card}(D_2)$.

(d) Notons $(C_k)_{k \in [\![1,K]\!]}$ les composantes connexes de G_D et $(A_k)_{k \in [\![1,K]\!]}$ leurs ensembles d'arêtes. Ainsi les A_k forment une partition de D, or D a strictement plus d'arêtes de D_2 que de D_1, donc il existe $k_0 \in [\![1, K]\!]$ tel que $\text{card}(A_{k_0} \cap D_2) > \text{card}(A_{k_0} \cap D_1)$. (En effet, si ce n'était pas le cas, on a aurait $\forall k \in [\![1, K]\!]$, $\text{card}(A_k \cap D_2) \leqslant \text{card}(A_k \cap D_1)$ et en sommant ces inégalités on aurait $\text{card}(D_2) \leqslant \text{card}(D_1)$, Absurde.)

Dans notre exemple il y a $K = 6$ composantes connexes, et pour la numérotation donnée ci-dessous c'est la composante C_3 qui a trois arêtes de D_2 et seulement deux D_1.



On s'intéresse alors à (C_{k_0}, A_{k_0}). En tant que sous-graphe de G_D, ce graphe n'a que des sommets de degré 0, 1 ou 2 (d'après (a)), et comme il est connexe, ce ne peut être qu'un cycle de longueur paire ou une chaîne,

qui de plus alterne les arêtes de D_1 et D_2 (d'après (b)). On en déduit qu'il y au plus 1 de différence entre le nombre d'arêtes de D_1 et de D_2 dans ce graphe, et sachant que $\mathrm{card}(A_{k_0} \cap D_2) > \mathrm{card}(A_{k_0} \cap D_1)$ on en déduit que $\mathrm{card}(A_{k_0} \cap D_2) = \mathrm{card}(A_{k_0} \cap D_1) + 1$. En particulier cela assure que $\mathrm{card}(A_k)$ est impaire, ce qui exclut le cas du cycle. Ainsi (C_{k_0}, A_{k_0}) est une chaîne de longueur impaire qui alterne arête de $D_1 \subseteq M$ et de $D_2 \subseteq A \setminus M$.

Posons $q = \mathrm{card}(A_{k_0})$. Le cas du cycle étant exclu, on a $\mathrm{card}(C_{k_0}) = \mathrm{card}(A_{k_0}) + 1 = q + 1$. Numérotons alors $(\gamma_i)_{i \in [\![0,q]\!]}$ les sommets de C_{k_0} de sorte que $A_{k_0} = \{\{\gamma_i, \gamma_{i+1}\} \mid i \in [\![0, q[\![\}$. (les γ_i sont deux à deux distincts par cardinalité). Ainsi pour montrer que γ est une chaîne augmentante pour M pour il ne reste qu'à vérifier que γ_0 et γ_q sont libres pour M.

- Si γ_0 est non libre pour M, il existe $x \in S$ tel que $\{\gamma_0, x\} \in M$. Puisque $\{\gamma_0, \gamma_1\} \in D_2$, d'une part $\{\gamma_0, \gamma_1\} \notin M$, et comme $\{\gamma_0, x\} \in M$ on en déduit $x \neq \gamma_1$, d'autre part $\{\gamma_0, \gamma_1\} \in M^\star$ et comme $M^\star$ est un couplage, $\{\gamma_0, x\} \notin M^\star$. Ainsi $\{\gamma_0, x\} \in M \setminus M^\star = D_1 \subseteq D$. Donc x est un voisin de γ_0 dans G_D, différent de γ_1, or γ_0 est de degré 1 dans G_D en tant qu'extremité de la chaîne qu'est (C_{k_0}, A_{k_0}). Absurde. Ainsi γ_0 est libre pour C.
- On montre de même que γ_q est libre pour M.

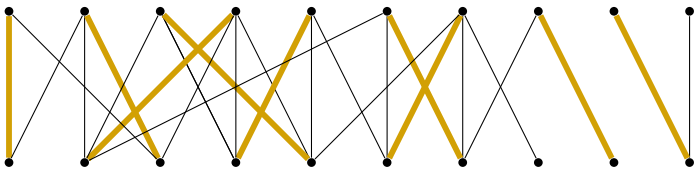
Finalement $(\gamma_0, \gamma_1, \ldots, \gamma_q)$ est un chemin augmentant pour M. □

Remarque 3.24

La proposition ci-dessus donne une condition suffisante pour qu'un couplage soit maximum, mais aussi, dans le cas où le couplage n'est pas maximum, une procédure effective pour construire un couplage de cardinal supérieur à partir d'un chemin augmentant, celle qui permet de passer de C à C' dans la propriété 3.23.

Exercice de cours 3.25

On considère le graphe biparti ci-dessous, sur lequel on a représenté un couplage maximum : —.



Donner les étapes obtenues en partant du couplage vide C, et en itérant le processus suivant :

- on découvre un chemin augmentant **en suivant la démonstration précédente** en utilisant le couplage maximum — ;
- on inverse ce chemin dans C, et on recommence.

On s'arrête lorsqu'il n'existe plus de chemin augmentant.

On déduit alors de cette proposition une méthode de recherche locale pour trouver un couplage maximum :

- partir de C l'ensemble vide qui est un couplage trivial ;
- chercher un chemin augmentant pour C, et si on trouve un tel chemin γ, inverser C le long de γ pour gagner en cardinalité ;
- si on ne trouve pas de chemin augmentant pour C, c'est que C est un couplage maximal.

La démonstration ci-dessus ne donne pas un algorithme pour trouver un chemin augmentant, en effet la construction suppose connu un couplage maximum, alors que c'est ce que l'on essaie de calculer.

Dans la section suivante on détaille comment est menée la recherche de chemin augmentant dans le cas particulier d'un graphe biparti♣.

♣. En effet tout ce qu'on a fait jusqu'ici vaut pour n'importe quel graphe non orienté

3.3 Algorithme pour les graphes bipartis

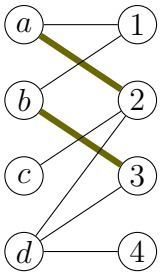
3.3.1 Trouver un chemin augmentant dans un graphe biparti

Étant donné un couplage C d'un graphe biparti non orienté $G = (S_1 \sqcup S_2, A)$, on souhaite ramener la recherche de chemin augmentant pour C à un problème d'accessibilité. Pour cela on construit un graphe orienté à partir de G, en :

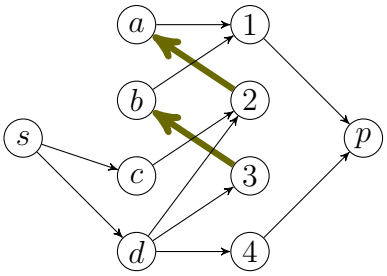
- ajoutant aux sommets une source s prédécesseur de tous les sommets de S_1 libres pour C ;
- ajoutant un puits p successeur de tous les sommets de S_2 libres pour C ;
- en orientant les arêtes hors de C de S_1 vers S_2, et celles de C de S_2 vers S_1.

Avant de formaliser cette construction, on l'illustre avec l'exemple ci-dessous.

Exemple 3.26



Graphe G et un couplage C



Le graphe augmenté G_C

Proposition 3.27

Soit $G = (S_1 \sqcup S_2, A)$ un graphe biparti. Soit C un couplage de G.
On construit le graphe ***orienté*** *$G_C = (S_1 \sqcup S_2 \sqcup \{s, p\}$[♣]$, \widetilde{A})$ où :*

$$\begin{aligned}\widetilde{A} = &\{(v,u) \mid \{u,v\} \in A \cap C \text{ et } u \in S_1 \text{ et } v \in S_2\}\\ &\cup \{(u,v) \mid \{u,v\} \in A \setminus C \text{ et } u \in S_1 \text{ et } v \in S_2\}\\ &\cup \{(s,u) \mid u \in S_1 \text{ libre pour } C\}\\ &\cup \{(v,p) \mid v \in S_2 \text{ libre pour } C\}\end{aligned}$$

Alors C admet un chemin augmentant si et seulement si G_C admet un chemin de s à p. De plus on a un algorithme permettant de construire un chemin augmentant pour C à partir d'un chemin dans G_C (et réciproquement).

Démonstration : Soit $\gamma = (\gamma_0, \gamma_1, \ldots, \gamma_{2k+1})$ un chemin augmentant dans G. Puisque γ est de longueur impaire, on a nécessairement $\gamma_0 \in S_1$ ou $\gamma_{2k+1} \in S_1$. Quitte à prendre la chaîne dans l'autre sens, on peut supposer que $\gamma_0 \in S_1$ (et donc que $\gamma_{2k+1} \in S_2$). Alors par définition d'un chemin augmentant :

- γ_0 et γ_{2k+1} sont des sommets libres pour C, donc $(s, \gamma_0) \in \widetilde{A}$ et $(\gamma_{2k+1}, p) \in \widetilde{A}$;
- pour tout $i \in [\![0, p]\!]$, $\{\gamma_{2i}, \gamma_{2i+1}\} \in A \setminus C$, et puisque $2i$ est pair, $\gamma_{2i} \in S_1$ donc $(\gamma_{2i}, \gamma_{2i+1}) \in \widetilde{A}$;
- pour tout $i \in [\![0, p-1]\!]$, $\{\gamma_{2i+1}, \gamma_{2i+2}\} \in A \cap C$, et puisque $2i+1$ est impair, $\gamma_{2i+1} \in S_2$ donc $(\gamma_{2i+1}, \gamma_{2i+2}) \in \widetilde{A}$.

Ainsi $(s, \gamma_0, \gamma_1, \ldots, \gamma_{2k+1}, p)$ est un chemin de s à p dans G_C.

Réciproquement, si $(s, \gamma_0, \gamma_1, \ldots, \gamma_{2k+1}, p)$ est un chemin de s à p dans G_C, il suffit de remarquer que $(\gamma_0, \gamma_1, \ldots, \gamma_{2k+1})$ est un chemin augmentant de G. □

♣. On suppose que s et p n'apparaissent pas dans $S_1 \cup S_2$

Finalement le problème de la découverte d'un chemin augmentant dans G se ramène au problème de la découverte d'un chemin dans un graphe orienté, ce que l'on sait déjà faire, avec une complexité en $\mathcal{O}(n+m)$ où n est le nombre de sommets du graphe et m le nombre d'arcs du graphe. On note dans la suite Trouve_chemin_augmentant un tel algorithme, prenant en arguments un graphe biparti et un couplage et retournant un chemin augmentant si celui-ci existe, None sinon.

Exercice de cours 3.28

Donner le pseudo-code de l'algorithme Trouve_chemin_augmentant.

Complexité On suppose que le graphe biparti $G = (S_1 \sqcup S_2, A)$ est décrit par une table de liste d'adjacence indexée par $S_1 = [\![0, n_1[\![$ et par n_2, et que C est décrit par un tableau de voisin indexé par S_2. Le calcul de G_C se fait alors en $O(n+m)$ (où n et m sont respectivement le nombre de sommets et d'arêtes de G). En effet, étant donné un sommet, on teste en $O(1)$ s'il est libre pour C, donc les arcs issus de s (resp. aboutissants en p) sont calculés en $O(n_1) = O(n)$ (resp. $O(n_2) = O(n)$). De plus, étant donné un sommet $u \in S_1$ on parcourt en $O(1 + \deg(u))$ ses voisins v, et pour chacun on teste en $O(1)$ si l'arête $\{u, v\}$ est dans C. Dans le cas où $\{u, v\} \in C$, on ajoute à G_C l'arc (v, u), et dans le cas contraire l'arc (u, v). Ainsi les arcs de G_C sont calculés en $O(n+m)$.
On en déduit que l'algorithme Trouve_chemin_augmentant est en $O(n+m)$ puisque qu'en plus de la construction de G_C, cet algorithme ne fait qu'un parcours de G_C, qui a $n+2$ sommets et moins de $m+2n$ arcs.

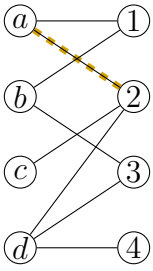
3.3.2 Résoudre le problème de couplage maximum dans un graphe biparti

Algorithme 3 : CouplageMaximum

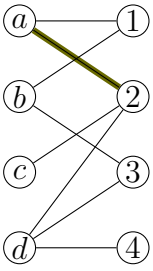
Entrée : Un graphe biparti $G = (S_1 \sqcup S_2, A)$
Sortie : Un couplage maximum de G

```
C ← ∅ ;
tant que Trouve_chemin_augmentant(G, C) ≠ None faire
    Notons γ le chemin augmentant ainsi trouvé ;
    On inverse C le long du chemin augmentant γ ;
retourner C ;
```

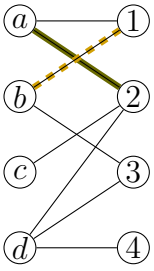
On donne ci-dessous le déroulé pas à pas de cet algorithme appliqué au graphe donné en exemple introductif à la page 19.



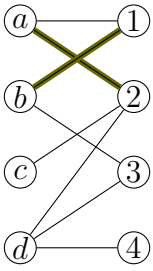
Découverte d'un chemin augmentant partant de a



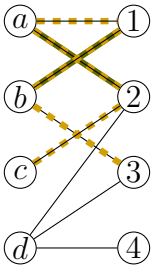
Inversion le long du chemin $(a, 2)$.



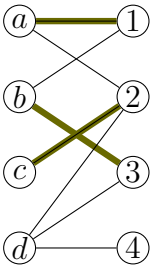
Découverte d'un chemin augmentant partant de b



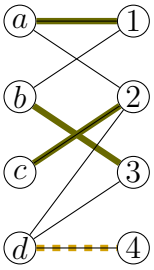
Inversion le long du chemin $(b, 1)$.



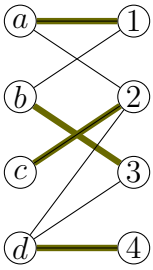
Découverte d'un chemin augmentant partant de c



Inversion le long du chemin $(c, 2, a, 1, b, 3)$.



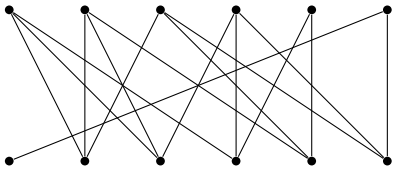
Découverte d'un chemin augmentant partant de d



Inversion le long du chemin $(d, 4)$.

Exercice de cours 3.29

Appliquer l'algorithme CouplageMaximum sur le graphe biparti ci-dessous.



Correction. La correction de l'algorithme est assurée par les propriétés précédentes, et par la négation de la condition de boucle : s'il n'existe plus de chemin augmentant, le couplage est de cardinal maximum.

Complexité L'algorithme CouplageMaximum est en $O(n(n+m))$, car le couplage maximum est de cardinal au plus $\frac{n}{2}$ (*Cf.* exercice de cours 3.9), ce qui assure qu'il y a au plus de l'ordre de n itérations, et chaque itération conduit à un appel à Trouve_chemin_augmentant dont on a montré qu'il est en $O(n+m)$.