Recherche opérationnelle

F. Olive

Table des matières

1 Rappels sur les graphes **5**
1.1 Graphes orientés et non orientés 5
1.2 Chemins, circuits et cycles 7
1.3 Arbres 9
1.4 Arborescences 11
1.5 Représentation des graphes 13

2 Algorithmes Gloutons **15**
2.1 Emploi du temps 15
2.2 Arbre couvrant minimal 16
2.2.1 Algorithme de Kruskal 17
2.2.2 Algorithme de Prim 19
2.3 Matroïdes 22

3 Programmation Dynamique **25**
3.1 Plus courts chemins dans un DAG 25
3.2 Sous suite commune maximale 26
3.3 Sac-à-dos 28
3.4 Sous-suite croissante maximale 31

4 Flot Maximal **33**

Chapitre 1

Rappels sur les graphes

Les graphes sont des structures combinatoires permettant de modéliser de nombreuses situations faisant intervenir l'algorithmique : conception de circuits intégrés, analyse de réseaux de transports ou de réseaux d'ordinateurs, ordonnancement d'un ensemble de tâches, représentation de cartes géographiques, etc. Informellement, un graphe est un ensemble de points dont certains sont mis en relation. Il existe en fait deux types de graphes : les graphes orientés et les graphes non orientés. Les premiers correspondent aux cas où la relation évoquée plus haut est quelconque ; les deuxièmes sont réservés à la représentation des ensembles de points reliés par une relation symétrique.

1.1 Graphes orientés et non orientés

Avant de définir la notion de graphe orienté, on rappelle un peu de vocabulaire relatif aux relations binaires.

Une **relation binaire** sur un ensemble X est une partie R de $X \times X$. Lorsqu'un couple $(x,y) \in X^2$ appartient à R, on note indifféremment xRy ou $R(x,y)$ ou $(x,y) \in R$.

Une relation binaire $R \subseteq X^2$ est dite :
- **réflexive** si $\forall x \in X : xRx$;
- **transitive** si $\forall x,y,z \in X : xRy$ et $yRz \Rightarrow xRz$;
- **symétrique** si $\forall x,y \in X : xRy \Rightarrow yRx$;
- **antisymétrique** si $\forall x,y \in X : xRy$ et $yRx \Rightarrow x = y$;
- **irréflexive** si $\forall x \in X : \text{non}(xRx)$;

Les relations binaires réflexives, symétriques et transitives sont appelées **relations d'équivalence**. Si R est une relation d'équivalence sur l'ensemble X, on appelle **classe d'équivalence** de l'élément $x \in X$ l'ensemble $\overline{x} = \{y \in X : xRy\}$. L'ensemble des classes d'équivalence selon R partitionne l'ensemble X. Autrement dit, il existe $x_1, \ldots, x_k \in X$ tels que $X = \overline{x_1} \sqcup \cdots \sqcup \overline{x_k}$ où $\sqcup$ dénote l'union disjointe.

Un **graphe orienté** est un couple $G = (V(G), E(G))$, où $V(G)$ est un ensemble et $E(G)$ une relation binaire sur $V(G)$. Dans la suite, on considèrera uniquement les graphes *finis* et *sans*

boucle, c'est-à-dire, les graphes G pour lesquels $V(G)$ est fini et $E(G)$ est irréflexive.

Les éléments de $V(G)$ sont les **sommets** de G. Ceux de $E(G)$ sont les **arcs** de G. Si $a = (x,y)$ est un arc du graphe orienté G, on utilise les terminologies suivantes :
- a est l'arc d'**origine** x et de **but** y. On note $ori(a) = x$ et $but(a) = y$.
- a est un arc **entrant** de y et un arc **sortant** de x.
- x est un **prédécesseur** de y et y est un **successeur** de x.

On note $Sortants(x)$ (resp. $Entrants(x)$) l'ensemble des arcs sortants (resp. entrants) du sommet x. L'union de ces deux ensembles est notée $Incid(x)$. Les arcs de $Incid(x)$ sont dits **incidents** à x.

Le **degré intérieur** (ou **degré entrant**) d'un sommet x est le nombre d'arcs entrants de x. On le note $\delta^-(x)$. Le **degré extérieur** (ou **degré sortant**) de x est le nombre d'arcs sortants de x. On le note $\delta^+(x)$. Le **degré** de x est la somme $\delta(x) = \delta^-(x) + \delta^+(x)$. En d'autres termes :

$$\delta^-(x) = |Entrants(x)| \ ; \ \delta^+(x) = |Sortants(x)| \ ; \ \delta(x) = |Incid(x)|.$$

Un **graphe non orienté** est un couple $G = (V(G), E(G))$, où $V(G)$ est un ensemble (fini) et $E(G)$ un ensemble de paires (non ordonnées) d'éléments de $V(G)$.

Les éléments de $V(G)$ sont les **sommets** de G. Ceux de $E(G)$ sont les **arêtes** de G. Si $a = \{x,y\}$ est une arête du graphe non orienté G, on utilise les terminologies suivantes :
- a est l'arête d'**extrémités** x et y ;
- x et y sont deux sommets **adjacents** ;

Si x est l'une des extrémités de l'arête a, a est dite **incidente** à x. L'ensemble des arêtes incidentes au sommet x est à nouveau noté $Incid(x)$ et l'on note encore $\delta(x)$ le **degré** de x, *i.e.* le cardinal de $Incid(x)$.

Si G est un graphe orienté, le graphe non orienté **sous-jacent** à G est le graphe non orienté obtenu en remplaçant chaque arc de G par une arête. Si G est un graphe non orienté, une **orientation** de G est un graphe orienté obtenu en choisissant une orientation pour chaque arête de G.

Remarque 1 *Dans un graphe non orienté, l'arête d'extrémités x et y est une paire (non ordonnée) de sommets et doit, en toute rigueur, être notée $\{x,y\}$ (par opposition à (x,y), qui dénote un couple ordonné). Pour faciliter la présentation de résultats communs au graphes orientés et non orientés, il nous arrivera néanmoins de noter (x,y) une arête d'extrémités x et y. Il nous faudra alors garder à l'esprit l'abus de notation que constitue ce choix. En particulier, si x et y sont deux sommets d'un graphe non orienté, (x,y) et (y,x) représentent le même objet (ce qui n'est pas le cas dans un graphe orienté).*

Les notions abordées en conclusion de cette section sont justement valables pour les deux types de graphes. C'est pourquoi on ne précise pas le statut (orienté ou non) des graphes qui entre en jeu dans leurs définitions.

Un **isomorphisme** du graphe G sur le graphe H est une application $h : V(G) \to V(H)$ bijective et telle que pour tous $x, y \in V(G)$: $(x,y) \in E(G) \Leftrightarrow (h(x),h(y)) \in E(H)$. Lorsqu'une telle application existe, on dit que G et H sont des **graphes isomorphes**.

Soit G un graphe. Un **sous-graphe** de G est un graphe H tel que $V(H) \subseteq V(G)$ et $E(H) \subseteq E(G)$. On note alors $H \preccurlyeq G$. Un sous-graphe de G est dit **couvrant** si son ensemble de sommets est $V(G)$.

Si $X \subseteq V(G)$, le sous-graphe de G **induit** par X est le sous-graphe H de G défini par : $V(H) = X$ et $E(H) = \{(u,v) \in E(G) : u,v \in X\}$. C'est le plus grand sous-graphe de G dont l'ensemble de sommets est X.

Si $Y \subseteq E(G)$, le sous-graphe de G induit par Y est le sous-graphe H de G défini par : $E(H) = Y$ et $V(H)$ est l'ensemble des extrémités des arcs (resp. arêtes) de Y. C'est le plus petit sous-graphe de G dont l'ensemble d'arcs (resp. arêtes) est Y.

À de nombreuses reprises, nous considèrerons des graphes construits à partir d'un graphe initial G par adjonction ou suppression de sommets ou d'arcs (resp. d'arêtes). Précisons tout de suite les modalités de ce type de constructions :

- Si X est une partie de $V(G)$, on note $G-X$ le graphe obtenu en supprimant de G tous les sommets apparaissant dans X et tous les arcs (resp. arêtes) incidents à ces sommets.
- Si X est un ensemble disjoint de $V(G)$, on note $G+X$ le graphe obtenu en ajoutant à G l'ensemble de sommets isolés X.
- Si Y est une partie de $E(G)$, on note $G-Y$ le graphe obtenu en supprimant de G tous les arcs (resp. arêtes) apparaissant dans Y (sans modifier $V(G)$).
- Si Y est un ensemble de couple (resp. de paires) de sommets de G disjoint de $E(G)$, on note $G+Y$ le graphe obtenu en ajoutant à G tous les arcs (resp. arêtes) apparaissant dans Y (sans modifier $V(G)$).

Autrement dit, les quatres graphes précités sont caractérisés par les égalités :

$$\begin{array}{llll} V(G-X) = V(G) - X & \text{et} & E(G-X) = E(G) - \bigcup_{x \in X} Incid(x) \\ V(G+X) = V(G) \cup X & \text{et} & E(G+X) = E(G) \\ V(G-Y) = V(G) & \text{et} & E(G-Y) = E(G) - Y \\ V(G+Y) = V(G) & \text{et} & E(G+Y) = E(G) \cup Y. \end{array}$$

Notons que $G-X \preccurlyeq G$ et $G-Y \preccurlyeq G$ alors que $G \preccurlyeq G+X$ et $G \preccurlyeq G+Y$. Enfin, si X (resp. Y) est réduit à un singleton $\{\alpha\}$, on notera $G+\alpha$ et $G-\alpha$ au lieu de $G+\{\alpha\}$ et $G-\{\alpha\}$.

1.2 Chemins, circuits et cycles

Un **chemin** dans un graphe orienté (resp. non orienté) G est une suite finie $p = x_0 a_1 x_1 a_2 \ldots x_{k-1} a_k x_k$ telle que pour tout i : $x_i \in V$, $a_i \in E$ et $a_i = (x_i, x_{i+1})$. Les sommets x_0 et x_k sont les **extrémités** de p. On dit aussi que p est un chemin **entre** x_0 et x_k. Le nombre d'arcs (resp. d'arêtes) par lesquelles passe le chemin est appelé sa **longueur**. Si p est un chemin entre u et v, on note $u \overset{p}{\rightsquigarrow} v$ ou $p : u \rightsquigarrow v$. Si $u = v$ ou s'il existe un chemin entre u et v, on dit que v est **accessible** à partir de

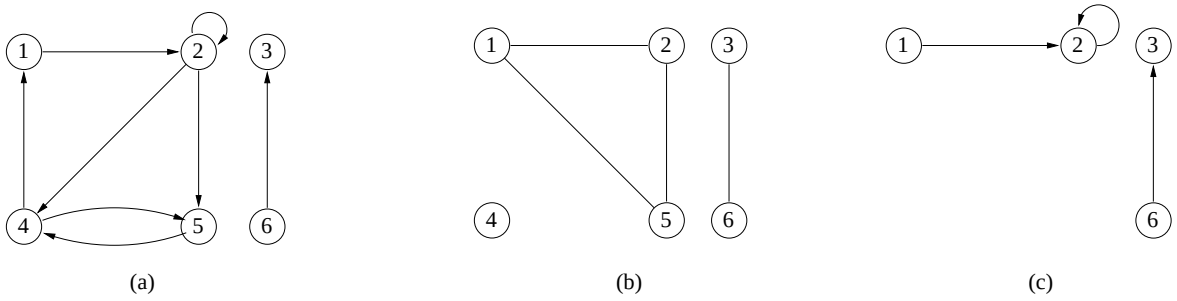


FIGURE 1.1 – Exemples de graphe orienté (a), non-orienté (b) et de sous-graphe induit, (c) est le sous-graphe de (a) induit par le sous-ensemble de sommets $\{1,2,3,6\}$

u. Si u est accessible à partir de v et v est accessible à partir de u, on dit que les deux sommets sont **mutuellement accessibles**, ou encore, **connectés**. Dans les graphes non orientés, les notions d'accessibilité et de mutuelle accessibilité coïncident.

Un chemin $p = x_0 a_1 x_1 a_2 \dots x_{k-1} a_k x_k$ est dit **élémentaire** si, en dehors de ses extrémités, les sommets par lesquels il passe sont deux à deux distincts. Dans un graphe orienté, un chemin dont les extrémités coïncident est appelé **circuit**. Dans un graphe non orienté, un tel chemin est un **cycle**. Un graphe non orienté sans cycle est dit **acyclique**.

Lemme 2 *Un graphe non orienté acyclique contient nécessairement un sommet de degré inférieur ou égal à* 1.

PREUVE. Supposons le contraire, en vue d'une contradiction. Soit donc G un graphe non orienté acyclique dont tous les sommets sont de degré > 1. Considérons la suite $x_0 a_1 x_1 a_2 x_2 \dots$ construite de la manière suivante : x_0 est l'un quelconque des sommets, a_1 est une arête incidente à x_0 et pour tout $i > 0$:

- x_i est l'extrémité de a_i distincte de x_{i-1} ;
- a_{i+1} est une arête incidente à x_i et distincte de a_i.

Notons que l'existence de a_1 et de chacune des a_{i+1} $(i > 0)$ est garantie par l'hypothèse : $\forall x \in V(G)$, $\delta(x) > 1$. Par ailleurs, puisque $V(G)$ est fini, il existe nécessairement deux entiers $i < j$ tels que $x_i = x_j$. Mais alors la suite $x_i a_{i+1} x_{i+1} \dots a_j x_j$ est un chemin (par construction) dont les extrémités coïncident : c'est un cycle de G, d'où une contradiction. □

Proposition 3 *Soit G un graphe non orienté acyclique. Alors* $|E(G)| \leq |V(G)| - 1$.

PREUVE. Par récurrence sur $|V(G)|$. Le résultat est clair pour $|V(G)| = 1$ ou $|V(G)| = 2$. Supposons-le vrai pour tout graphe de cardinal strictement inférieur à un entier $n > 1$ fixé. Soit G un graphe acyclique de cardinal n. Par le Lemme 2, il existe un sommet $x \in V(G)$ de degré < 1. Alors $G - x$ est un graphe acyclique de cardinal $n-1$ et, par hypothèse de récurrence : $|E(G-x)| \leq |V(G-x)| - 1$. Or, $|E(G-x)| = |E(G)| - \delta(x)$ et $|V(G-x)| = |V(G)| - 1$. D'où finalement : $|E(G)| - \delta(x) \leq |V(G)| - 2$ et, puisque $\delta(x) \leq 1$: $|E(G)| \leq |V(G)| - 1$. □

Lemme 4 *Soient x,y deux sommets d'un graphe G. S'il existe un chemin entre x et y, alors il existe un chemin élémentaire entre x et y. En particulier, s'il existe un circuit passant par x, alors il existe un circuit élémentaire passant par x.*

PREUVE. Exercice. □

La relation de mutuelle accessibilité dans un graphe G est une relation d'équivalence sur $V(G)$. On la note $\sim$. Dans un graphe orienté (resp. non orienté), les classes d'équivalence de $V(G)$ selon $\sim$ sont appelées **composantes fortement connexes** (resp. **composantes connexes**) du graphe. Par conséquent, deux sommets u et v d'un graphe orienté G sont dans une même composante fortement connexe si et seulement si il existe $p : u \rightsquigarrow v$ et $p' : v \rightsquigarrow u$; deux sommets u et v d'un graphe non orienté G sont dans une même composante connexe si et seulement si il existe $p : u \rightsquigarrow v$. Un graphe orienté (resp. non orienté) G est dit **fortement connexe** (resp. **connexe**) si $V(G)$ ne contient qu'une classe d'équivalence relativement à $\sim$, *i.e.*, si deux sommets quelconques de G sont mutuellement accessibles.

Proposition 5 *Soit G un graphe non orienté connexe. Alors $|E(G)| \geq |V(G)| - 1$.*

PREUVE. Par récurrence sur $|V(G)|$. Le résultat est clair pour $|V(G)| = 1$ ou $|V(G)| = 2$. On considère un entier $n > 1$ et on suppose que la propriété est vraie pour tous les graphes de cardinal $< n$. Soit G un graphe connexe à n sommets et m arêtes. Pour un sommet fixé $x \in V(G)$, on note $G_1, \ldots, G_k$ les sous-graphes induits par les composantes connexes du graphe $G - x$. Par hypothèse de récurrence, chacun de ces graphes satisfait $|E(G_i)| \geq |V(G_i)| - 1$. Par ailleurs, puisque G est connexe, x est relié par au moins une arête à chacune des k composantes G_i. Par conséquent :

$$m \geq \sum_{i=1}^{k} |E(G_i)| + k \geq \sum_{i=1}^{k} (|V(G_i)| - 1) + k = \sum_{i=1}^{k} |V(G_i)| = n - 1.$$

C'est le résultat cherché. □

1.3 Arbres

Nous introduisons ici une classe de graphes très importante : celle des **arbres**, c'est-à-dire, des graphes non orientés *connexes* et *acycliques*. Nous verrons dans les chapitres suivants que cette notion est à la base d'un grand nombre d'algorithmes sur les graphes. Le résultat principal de cette section est celui du Théorème 7, qui donne plusieurs caractérisations des arbres. Etablissons, avec le Lemme 6, une première relation entre les notions de cyclicité et de connexité :

Lemme 6 *Soit $a = (x,y)$ une arête d'un graphe non orienté G.*

1. *a est sur un cycle de G ssi x et y sont connectés dans $G - a$.*
2. *En particulier, si G est connexe : a est sur un cycle de G ssi $G - a$ est connexe.*

PREUVE.

1. $\boxed{\Leftarrow}$ Soit $p_0 : x \rightsquigarrow y$ un chemin de $G-a$. Alors, $x \overset{p_0}{\rightsquigarrow} y \overset{a}{\to} x$ est un cycle de G qui contient a. $\boxed{\Rightarrow}$ Considérons un cycle *élémentaire* de G contenant a. Il s'écrit : $x \overset{p_0}{\rightsquigarrow} y \overset{a}{\to} x$, où p_0 est un chemin de G qui ne contient pas a. Par conséquent, p_0 est un chemin de $G-a$ qui connecte x et y dans $G-a$.
2. $\boxed{\Leftarrow}$ Puisque $G-a$ est connexe, x et y sont connectés dans $G-a$ et a est sur un cycle de G d'après 1. $\boxed{\Rightarrow}$ Par 1, il existe un chemin $p_0 : x \rightsquigarrow y$ dans $G-a$. Soient u, v deux sommmets de G et $p : u \rightsquigarrow v$ un chemin de G (dont l'existence est garantie par la connexité de G). Soit p' la suite obtenue en remplaçant chaque occurence de a dans p par le chemin p_0. Alors p' est clairement un chemin de G d'extrémités u, v. De plus, p' ne contient pas a : c'est un chemin de $G-a$ qui connecte u et v dans $G-a$.

□

Ce dernier lemme peut s'entendre comme suit : dans un graphe connexe, la présence d'un cycle implique l'existence d'une arête « redondante » pour la connexité : on peut la retirer sans déconnecter le graphe. A l'inverse, dans un graphe acyclique, le retrait d'une arête affaiblit le degré de connexité du graphe, c'est-à-dire, augmente le nombre de ses composantes connexes. Ainsi les arbres réalisent-ils deux exigences contradictoires : la connexité, qui demande suffisamment d'arêtes pour lier deux à deux tous les sommets ; l'acyclicité, qui au contraire appelle un nombre d'arêtes inférieur à un certain seuil, au dela duquel des cycles apparaissent nécessairement. En d'autres termes, les arbres sont les graphes acycliques qui ont suffisamment d'arêtes pour rester connexes, ou encore, les graphes connexes qui ont suffisamment peu d'arêtes pour rester acycliques. Le théorème suivant précise ces considérations :

Théorème 7 *Soit G un graphe non orienté avec n sommets et m arêtes. Les assertions suivantes sont équivalentes :*

(1) *G est un arbre ;*

(2) *G est connexe et $m = n-1$;*

(3) *G est connexe et la suppression d'une arête quelconque le déconnecte ;*

(4) *G est acyclique et $m = n-1$;*

(5) *G est acyclique et l'ajout d'une arête quelconque crée un cycle ;*

(6) *entre deux sommets quelconques de G, il existe un* unique *chemin élémentaire.*

PREUVE. On montre les trois chaînes d'implications : $(1) \Rightarrow (2) \Rightarrow (3) \Rightarrow (1)$, $(1) \Rightarrow (4) \Rightarrow (5) \Rightarrow (1)$ et $(1) \Rightarrow (6) \Rightarrow (1)$.

$(1) \Rightarrow (2)$ Evident, par définition des arbres et grâce aux Propositions 3 et 5.

$(2) \Rightarrow (3)$ Soit a une arête quelconque de G. On a $|V(G-a)| = |V(G)| = n$ et $|E(G-a)| = m-1 = n-2 < |V(G-a)|-1$. Donc $G-a$ est non connexe, par la Proposition 5.

$(3) \Rightarrow (1)$ L'hypothèse dit notamment que pour toute arête a de G, $G-a$ est non connexe. Par le Lemme 6, cela signifie qu'aucune arête de G n'est sur un cycle. En d'autres termes, le graphe G est acyclique. Comme il est connexe, c'est bien un arbre.

$(1) \Rightarrow (4)$ Evident, par définition des arbres et grâce aux Propositions 3 et 5.

(4) $\Rightarrow$ (5) Soit $a = \{x,y\}$ une paire de sommets qui n'est pas dans $E(G)$. On a $|V(G+a)| = |V(G)| = n$ et $|E(G+a)| = |E(G)|+1 = m+1$. Or $m+1 = n$, par hypothèse. Donc $|E(G+a)| > |V(G+a)|-1$ et $G+a$ contient un cycle, par la Proposition 3.

(5) $\Rightarrow$ (1) Il suffit de montrer que G est connexe, c'est-à-dire que deux sommets quelconques de G sont reliés par un chemin. Soient donc x,y deux sommets de G. Si $a = \{x,y\}$ est dans $E(G)$, alors x et y sont connectés. Sinon, $G+a$ contient un cycle (par hypothèse) qui passe nécessairement par a (sinon ce serait un cycle de G). Puisque a est sur un cycle de $G+a$, ses extrémités x et y sont connectées dans le graphe $(G+a)-a$ (par le Lemme 6). Autrement dit, x et y sont connectés dans G.

(1) $\Rightarrow$ (6) Soient x,y deux sommets de G. L'existence d'un chemin élémentaire entre ces deux sommets est assurée par la connexité de G et le Lemme 4. Prouvons son unicité. En vue d'une contradiction, supposons qu'il existe deux chemins élémentaires distincts $p : x \rightsquigarrow y$ et $q : x \rightsquigarrow y$. Puisque $p \neq q$, il existe une arête a de G qui apparaît dans q et pas dans p. Donc p est toujours un chemin dans $G-a$ et les sommets x et y sont connectés dans $G-a$. Par conséquent (Lemme 6), a est sur un cycle de G, ce qui est impossible puisque G est acyclique.

(6) $\Rightarrow$ (1) Il nous faut montrer que G est connexe et acyclique. La connexité est clair, puisqu'il existe un chemin (élémentaire) entre deux sommets quelconques de G. De plus, si $a = \{x,y\}$ est une arête quelconque de G, $x \xrightarrow{a} y$ est l'unique chemin de G entre x et y. Par conséquent, x et y ne sont pas connectés dans $G-a$ et, par le Lemme 6, a n'est pas sur un cycle de G. Donc G est acyclique.

Ceci clôt la preuve du théorème. □

Un graphe dont chaque composante connexe est un arbre est appelé **forêt**. Autrement dit, une forêt est un graphe non orienté acyclique. Un **arbre enraciné** est la donnée d'un arbre dans lequel on a désigné un sommet particulier. Autrement dit, c'est un couple (T,r), où T est un arbre et r un sommet de T. Ce sommet est alors appelé **racine** de l'arbre enraciné (T,r).

1.4 Arborescences

La définition des arbres est intimement liée à la nature non orienté des graphes qui en relèvent. Elle n'a pas d'équivalent direct dans le cadre orienté. Par exemple, la tentation de transcrire cette définition par : "graphes orientés *fortement connexe* et *sans circuit*" est vouée à l'échec puisque de tels graphes n'existent pas. On définit pourtant une notion d'arbre orienté – on parle plutôt d'arborescence – de la manière suivante : une **arborescence** est un couple (G,r) constitué d'un graphe orienté et d'un sommet de ce graphe, tel que pour tout $x \in V(G)$, il existe un unique chemin de r à x. Le lien entre cette définition et celle des arbres est illustré par le lemme suivant :

Lemme 8 *Soit G un graphe.*

– *Si G est une arborescence de racine r, alors le graphe non orienté sous-jacent à G est un arbre.*

– *Si G est un arbre, alors pour tout $r \in V(G)$, il existe une orientation de G qui en fait une arborescence de racine r.*

PREUVE. Exercice. □

Le théorème suivant donne deux caractérisations des arborescences :

Théorème 9 *Soient G un graphe orienté et r un sommet de G. Les assertions suivantes sont équivalentes :*

(1) *G est une arborescence de racine r ;*

(2) *G est connexe et $\delta^-(r) = 0$ et $\forall x \neq r$, $\delta^-(x) = 1$;*

(3) *G est sans circuit et $\delta^-(r) = 0$ et $\forall x \neq r$, $\delta^-(x) = 1$.*

PREUVE. (1) $\Rightarrow$ (2) et (1) $\Rightarrow$ (3) résultent de la définition des arborescences.

(2) $\Rightarrow$ (1) Considérons un chemin *non orienté* $p = ra_1x_1 \dots a_kx$ entre r et un sommet quelconque x. (Un tel chemin existe puisque G est connexe.) Comme $\delta^-(r) = 0$, a_1 est un arc tel que $a_1 = (r, x_1)$. Comme $\delta^-(x_1) = 1$, r est son unique prédécesseur, et a_2 est un arc tel que $a_2 = (x_1, x_2)$, et ainsi de suite. Autrement dit, p est un chemin orienté entre r et x. L'unicité de p provient du fait que tout $x \neq r$ admet un prédécesseur unique, puisque $\delta^-(x) = 1$.

(3) $\Rightarrow$ (1) Puisque G est sans circuit, tout chemin de longueur non nulle est élémentaire. Soient $x \in V(G)$ et $p = x_0a_1x_1 \dots a_kx$ le plus long chemin élémentaire se terminant en x. Supposons, en vue d'une contradiction, que x_0 a un prédécesseur y. Si $y \notin p$, alors $y(y, x_0)p$ est un chemin élémentaire se terminant en x, ce qui contredit l'hypothèse de maximalité de p. Si $y \in p$, alors $y(y, x_0)p[x_0, y]$ est un circuit, ce qui contredit l'hypothèse (3). Finalement, x_0 n'a pas de prédécesseur. Autrement dit : $x_0 = r$ et p est un chemin entre r et x. L'unicité de p se prouve comme précédemment.

□

Le vocabulaire relatif aux arborescences emprunte autant à la botanique qu'à la généalogie : les sommets d'une arborescence (G, r) sont appelés **nœuds** de G. Nous avons vu qu'à l'exception de la racine, qui est de degré entrant nul, tout nœud d'une arborescence est de degré entrant 1. Pour chaque $x \neq r$, l'unique prédécesseur de x dans G est appelé **père** de x. On le note $père(x)$. Un nœud peut être de degré sortant quelconque. Les nœuds de degré sortant nul sont appelés **feuilles** de l'arbre ; les autres sont dits **nœuds internes**. Les successeurs d'un nœud interne sont appelés ses **fils**. Par définition, il existe un unique chemin entre r et un nœud quelconque x. Les nœuds qui constituent ce chemin sont les **ancêtres** de x. Les ancêtres de x sont exactement les nœuds u pour lesquels il existe un chemin $p : u \rightsquigarrow x$ dans G. A l'inverse, les sommets v accessibles à partir de x, c'est-à-dire pour lesquels il existe un chemin $q : x \rightsquigarrow v$, sont appelés **descendants** de x. Pour des raisons de commodité d'écriture, nous adoptons la convention que tout nœud x est à la fois ancêtre et descendant de lui-même. Le sous-graphe de G induit par l'ensemble des descendants de x est une arborescence de racine x.

1.5 Représentation des graphes

Il existe de nombreuses façons de représenter des graphes. On en présente ici deux : les représentations par matrice d'adjacence et par listes de successeurs.

Pour la représentation par **matrice d'adjacence** d'un graphe G à n sommets, on suppose que les sommets du graphe sont numérotés arbitrairement de 1 à n : $V(G) = \{x_1, \ldots, x_n\}$. La matrice d'adjacence de G est alors la matrice M de taille $n \times n$, à valeurs dans $\{0,1\}$, définie par :

$$M_{ij} = \begin{cases} 1 \text{ si } (x_i, x_j) \in E(G), \\ 0 \text{ sinon.} \end{cases}$$

Cette matrice s'implémente facilement en *C* comme un tableau de dimension 2. L'encombrement mémoire de cette représentation est en $O(n^2)$.

La représentation de G par **listes de successeurs** consiste en un tableau T de n listes, une pour chaque sommet de G. Pour chaque $x \in V(G)$, la liste des successeurs de x est une liste chaînée des sommets y tels que $(x,y) \in E(G)$. Autrement dit, $T(x)$ est constitué de tous les successeurs de x dans G. Les sommets de chaque liste de successeurs sont généralement chaînés suivant un ordre arbitraire.

Si G est un graphe orienté, la somme de toutes les longueurs des listes de successeurs vaut $|E(G)|$. Si G est non orienté, cette somme vaut $2|E(G)|$. Dans les deux cas, l'encombrement mémoire de la représentation d'un graphe à n sommets et m arêtes est en $O(n+m)$.

Les figures 1.2 et 1.3 illustrent ces deux types de représentations pour un graphe orienté et un graphe non orienté.

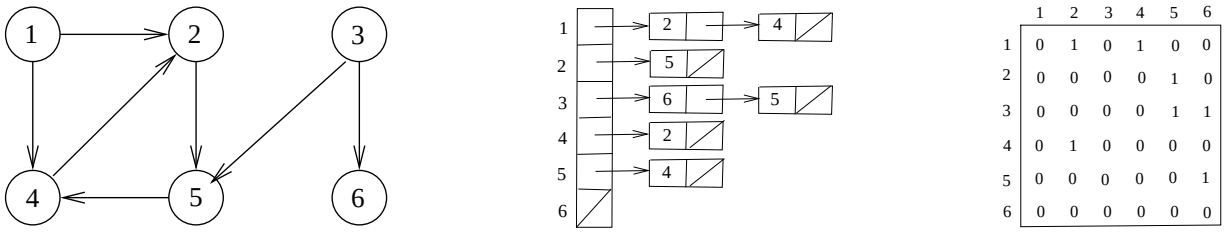


FIGURE 1.2 – Représentation d'un graphe *orienté* par listes de successeurs et matrice d'adjacence

La représentation par listes d'adjacence est souvent privilégiée parce qu'elle permet de représenter d'une façon plus compacte les graphes peu denses – ceux pour lesquels $|E(G)|$ est beaucoup plus petit que $|V(G)|^2$. La représentation par matrice d'adjacence est néanmoins intéressante si le graphe est dense ou si l'on doit souvent tester la présence d'un arc entre deux sommets. Dans ce dernier cas, la représentation par listes d'adjacence est particulièrement mal adaptée. Un intérêt supplémentaire de la représentation matricielle est que la détermination de chemins dans G revient au calcul des puissances successives de la matrice M, comme le montre la proposition suivante :

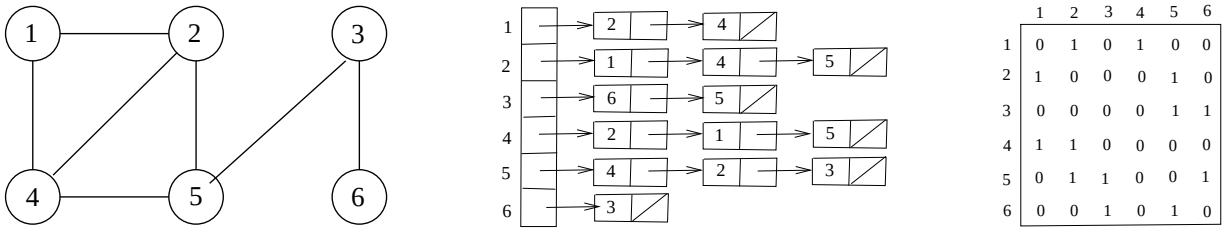


FIGURE 1.3 – Représentation d'un graphe *non orienté* par listes de successeurs et matrice d'adjacence

Proposition 10 *Soit M^k la puissance k-ème de la matrice d'adjacence d'un graphe G. Le coefficient M^k_{ij} est égal au nombre de chemins de G de longueur k entre x_i et x_j.*

PREUVE. Récurrence sur k. Le résultat est immédiat pour $k = 1$ puisqu'un chemin de longueur k est un arc du graphe. Le calcul de M^k pour $k > 1$ donne :

$$M^k_{ij} = \sum_{\ell=1}^{n} M^{k-1}_{i\ell} M_{\ell j}.$$

Or tout chemin de longueur k entre x_i et x_j de décompose en un chemin de longueur $k-1$ entre x_i et un certain x_ℓ, suivi d'un arc reliant x_ℓ et x_j. Le résultat découle alors de l'hypothèse de récurrence selon laquelle $M^{k-1}_{i\ell}$ est le nombre de chemins de longueur $k-1$ joignant x_i à x_ℓ. □

Pour conclure cette section, remarquons que la structure des arborescences donne lieu à une représentation particulièrement compacte : puisque les arcs d'une arborescence (G,r) sont entièrement décrits par la fonction *père* – les arcs de G sont tous les couples de la forme $(père(x),x)$ pour $x \neq r$ –, le graphe peut être représenté par un tableau qui rend compte de cette fonction, *e.g.* le tableau de taille $|V(G)|$ dont la case d'indice i contient l'unique j tel que $x_j = père(x_i)$. L'encombrement mémoire d'une telle représentation est en $O(n)$, où n est le nombre de sommets du graphe.

Chapitre 2

Algorithmes Gloutons

2.1 Emploi du temps

On veut programmer l'occupation d'une salle par des cours (ou plus généralement, l'utilisation d'une ressource par des événements). Les contraintes sont les suivantes :

- Chaque cours c est doté d'une heure de début $d(c)$ et d'une heure de fin $f(c)$.
- Deux cours c, c' programmés dans la salle doivent être « compatibles » :

$$]d(c), f(c)[\cap]d(c'), f(c')[=\varnothing.$$

- On cherche de plus à maximiser le nombres de cours programmés.

Plus formellement, le problème à résoudre s'écrit :

Input : un ensemble fini E et pour chaque $i \in E$, deux nombres $d(i) < f(i)$;
Output : $S \subseteq E$ de taille maximale tel que $\forall i \neq j$ dans S : $]d(i), f(i)[\cap]d(j), f(j)[=\varnothing$.

Algorithme glouton pour l'emploi du temps : On suppose les cours triés par heures de fin croissantes. À chaque étape on fait un choix "localement optimal", en prenant le cours qui termine le plus tôt. Les heures de début et de fin de cours sont stockées dans deux tableaux d et f. On note $C = \{1, \ldots, n\}$ l'ensemble des cours.

Algorithme 2 : Algorithme `edt-glouton`(d, f)

Données : les tableaux $d[1..n]$ et $f[1..n]$ de dates de début et de fin des cours
Résultat : Un ensemble maximal de cours deux à deux compatibles
Sol $= \{1\}$; $j = 1$;
pour $i = 2$ *à* n **faire**
  si $d[i] \geq f[j]$ **alors**
    Sol $=$ Sol $+ i$;
    $j = i$;
  fin
fin
1 **retourner** Sol ;

La terminaison et la complexité ($O(n)$) sont immédiates.

Correction de `edt-glouton` : On note :
- $(g_1,\ldots,g_p)$ la solution retournée par `edt-glouton`(C,f,d) ;
- $(o_1,\ldots,o_n)$ une solution optimale (en particulier, $n \geq p$).

On suppose les deux suites triées par heures de fin croissantes. Soit j le premier indice pour lequel les suites diffèrent. Autrement dit : $j \in \{1,\ldots,p+1\}$, $g_j \neq o_j$ et $\forall i < j : g_i = o_i$. Montrons que $(g_1,\ldots,g_{j-1},g_j,o_{j+1},\ldots,o_n)$ est une solution optimale :

Par construction de `edt-glouton`, g_j est l'evénement de date finale minimale, parmi ceux compatibles avec $\{g_1,\ldots,g_{j-1}\}$. En particulier, $f(g_j) \leq f(o_j)$ et par conséquent, g_j est, comme o_j, compatible avec o_{j+1}. Il s'ensuit que $(g_1,\ldots,g_{j-1},g_j,o_{j+1},\ldots,o_n)$ est une solution. Comme elle est de même cardinal que $(o_1,\ldots,o_n)$, c'est une solution optimale. De proche en proche on peut ainsi construire une solution optimale du type $(g_1,\ldots,g_p,o_{p+1},\ldots,o_n)$. Reste à constater que $n = p$, sinon un cours supplémentaire aurait été considéré et intégré à Sol par `edt-glouton`.

2.2 Arbre couvrant minimal

Un graphe **pondéré** (ou **valué**) est la donnée d'un graphe $G = (V,E)$ et d'une fonction $\ell : E \to \mathbb{R}$, appelée fonction de **poids** (ou **valuation**, ou encore fonction de **coût**). Le poids d'un graphe pondéré (G,ℓ) est la somme des poids de ses arêtes (resp. de ses arcs).

Nous avons vu (Exercice 11) que tout graphe G non orienté et connexe admet un arbre couvrant. Par conséquent, si G est pondéré, l'ensemble des poids de ses arbres couvrants est une partie non vide de $\mathbb{N}$. Il admet donc un élément minimum. Les arbres couvrants de G dont le poids réalise ce minimum sont appelés **arbres couvrants minimaux** de G. La recherche d'un arbre couvrant minimal d'un graphe non orienté pondéré et connexe est un problème classique, pour lequel existe plusieurs algorithmes. Nous allons en évoquer deux : l'algorithme de `Kruskal` et l'algorithme de `Prim`.

Mais auparavant, nous prouvons un résultat simple qui facilitera la présentation des sections suivantes. Nous savons qu'un graphe acyclique T est un arbre ssi l'ajout d'une arête à T crée un cycle (Théorème 7). En particulier, si T est un sous-graphe couvrant acyclique d'un graphe connexe G, T est un arbre couvrant de G ssi pour toute *paire de sommets* de G, $a = \{x,y\}$, qui n'est pas une arête de T, $T + a$ contient un cycle. En fait cette condition peut être remplacer par une condition plus faible : T est un arbre couvrant de G ssi pour toute *arête* a de G qui n'est pas une arête de T, $T + a$ contient un cycle. De surcroît, cette nouvelle condition permet de se passer de l'hypothèse "T couvrant" qui en est une conséquence. En résumé, on a :

Lemme 11 *Soit T un sous-graphe acyclique d'un graphe connexe G. Alors T est un arbre couvrant de G ssi pour toute $a \in E(G) - E(T)$, $T + a$ contient un cycle.*

PREUVE. L'implication $\Rightarrow$ est claire. Pour $\Leftarrow$, notons d'abord que l'hypothèse implique facilement que T couvre G, puisque s'il existait un sommet $x \in V(G) - V(T)$, ce sommet serait

l'extrémité d'une arête a (puisque G est connexe) qui n'est pas dans $E(T)$ et telle que $T+a$ ne contient pas de cycle. Reste à montrer que T est connexe. Supposons le contraire, en vue d'une contradiction : soit C une composante connexe de T. Puisque T est non connexe, $V(G)-C \neq \varnothing$ et, grâce au résultat de l'Exercice 6, la connexité de G entraîne l'existence d'une arête $a \in E(G)$ qui relie C à $V(G)-C$. Notons $a=\{x,y\}$ avec $x \in C$ et $y \notin C$. Alors, par hypothèse, a est sur un cycle de $T+a$ et, par le Lemme 6-(1), x et y sont connectés dans $(T+a)-a=T$, ce qui contredit le fait que $x \in C$ et $y \notin C$. □

2.2.1 Algorithme de Kruskal

L'algorithme de **`Kruskal`** maintient une forêt couvrante F de G, initialement vide d'arête, en ajoutant une à une les arêtes de poids minimum qui laissent F acyclique.

Algorithme 3 : Algorithme de `Kruskal`

Données : Un graphe pondéré connexe (G,c)
Résultat : Un ACM de (G,c)
1 $F=\varnothing$;
2 trier les arêtes de G par ordre croissant de pondération ;
3 **pour** *chaque arête a, par ordre de pondération croissante* **faire**
4 **si** $F+a$ *est acyclique* **alors**
5 $F=F+a$;
  fin
fin
6 **retourner** F

Proposition 12 (Correction de `Kruskal`.) *L'algorithme de* `Kruskal` *retourne un arbre couvrant minimal de* (G,c).

PREUVE. Notons tout d'abord que l'algorithme termine bien sur toute entrée, puisque la boucle "pour" qui le compose consiste en un parcours d'une liste *finie* d'arêtes. Montrons par ailleurs que l'assertion

$$F \text{ est un sous-graphe acyclique d'un arbre couvrant minimal de } G \qquad (*)$$

est un invariant de la boucle **pour**. Il s'agit donc de montrer que si le sous-graphe F obtenu juste avant une exécutions de la boucle **pour** (ligne 3) satisfait $(*)$, il en va de même pour le sous-graphe obtenu après cette exécution. Soit donc a l'arête ajoutée à F lors de ce dernier passage dans la boucle **pour**. Puisque F est acyclique – comme sous-graphe d'un arbre –, il en va de même pour $F+a$ en vertu de la ligne 4. Soit T_0 un ACM qui contient F (hypothèse d'induction). Si T_0 contient a, alors $F+a$ est un sous-graphe de T_0 et satisfait bien $(*)$. Sinon, T_0+a contient un cycle (Théorème 7). Ce cycle contient nécessairement une arête b qui n'est pas dans $F+a$, puisque $F+a$ est acyclique. Alors $(T_0+a)-b$ est un arbre couvrant qui contient $F+a$. De plus $c((T_0+a)-b)=c(T_0)+c(a)-c(b)$. Or $c(b) \geq c(a)$: sinon, b aurait été considéré avant

a dans l'exécution de l'algorithme, et elle aurait été intégrée à F, puisqu'elle ne crée pas de cycle dans F. On aurait donc $b \in F + a$, ce qui n'est pas. Ainsi, $c(b) \geq c(a)$ et par conséquent, $c((T_0 + a) - b) \leq c(T_0)$: $(T_0 + a) - b$ est un arbre couvrant minimal qui contient $F + a$ et $F + a$ satisfait $(*)$.

Par ailleurs, cette assertion est vraie après l'exécution de la ligne 1 : après initialisation, F, sous-graphe sans arête de G, est sous-graphe de tout arbre couvrant de G. Elle reste vraie après l'instruction 2, qui ne modifie pas F. Finalement elle est bien vérifiée après exécution de l'intégralité de la boucle.

Il reste à remarquer que lorsque l'algorithme termine, toute les arêtes de G ont été examinées. Et toutes celles qui ne sont pas dans F ont été rejetées parce qu'elles créaient un cycle dans F. Autrement dit, pour tout $a \in E(G) - E(F)$, $F + a$ contient un cycle. Par le Lemme 11, F est un arbre couvrant de G. Il s'ensuit que le sous-graphe F construit par l'algorithme de `Kruskal` est un arbre couvrant inclus dans un arbre couvrant minimal de G : c'est un arbre couvrant minimal de G. □

Afin d'analyser la complexité de l'algorithme de `Kruskal`, nous allons en détailler l'implémentation. Pour mettre en œuvre l'instruction de la ligne 4, il nous faut résoudre le problème suivant : comment déterminer si une arête $\{x, y\}$ choisie au $i$ème passage de l'instruction 4 forme un cycle avec l'ensemble F_{i-1} des arêtes choisies antérieurement ? Pour que $\{x, y\}$ forme un cycle, il faut que x et y soient connectés dans F_{i-1}, et donc qu'ils soient dans une même composante connexe de F_{i-1}. Il nous faut donc gérer l'évolution des composantes connexes au fur et à mesure du choix des arêtes. Pour ce faire, on inititialise, au début de l'algorithme, n composantes connexes $V_u = \{u\}$, $u \in V$. Chaque fois qu'une arête $\{x, y\}$ est candidate, on compare les ensembles V_x et V_y. S'ils sont égaux, l'arête $\{x, y\}$ créerait un cycle et on la rejette. Sinon, on la garde et on fusionne les ensembles V_x et V_y (*i.e.*, on affecte à chacun d'eux la nouvelle valeur $V_x \cup V_y$). La création, la comparaison et l'union des ensembles V_u peut se faire au moyens de la structure de données "ensembles" et des fonctions suivantes :

- `Créer-Ensemble`(u), qui crée un ensemble réduit au singleton $\{u\}$ et lui affecte un indice identificateur ;
- `Trouver-Ensemble`(u), qui renvoie l'indice identificateur de l'ensemble qui contient u ;
- `Union`(u, v), qui remplace l'indice identificateur de l'ensemble qui contient v par celui de l'ensemble qui contient u.

L'algorithme de `Kruskal` peut alors s'écrire comme suit :

Algorithme 4 : Une implémentation de `Kruskal`

Données : Un graphe pondéré connexe (G,c)
Résultat : Un ACM de (G,c)
1 $F \leftarrow \varnothing$;
2 **pour** *chaque sommet* $u \in V(G)$ **faire**
3 `Créer-Ensemble`(u) ;
fin
4 trier les arêtes de G par ordre croissant de pondération ;
5 **pour** *chaque arête* (x,y)*, par ordre de pondération croissante* **faire**
6 **si** `Trouver-Ensemble`$(x) \neq$ `Trouver-Ensemble`(y) **alors**
7 $F \leftarrow F \cup \{x,y\}$;
8 `Union`(x,y)
fin
fin
9 **retourner** F

Lemme 13 (Complexité de l'algorithme de `Kruskal`.) *Sur un graphe à n sommets et m arêtes, l'algorithme de* `Kruskal` *termine en temps* $O(m \log m)$.

PREUVE. L'initialisation des composantes connexes (lignes 2 et 3) requiert un temps $O(n)$ (l'opération `Créer-Ensemble`(u) prend un temps constant). Le tri des arêtes (ligne 4) se fait en temps $O(m \log m)$. Dans la seconde boucle **pour** (ligne 5), les opérations de mise à jour de F et d'union prennent un temps constant. Le test de la ligne 6, lui, peut être effectué en temps $O(\log m)$ pour une bonne implémentation de la structure de données "ensembles". La seconde boucle **pour** comportant m tests **si**, on obtient au total une complexité en $O(n) + O(m \log m) + O(m \log m) = O(n + m \log m)$. Comme le graphe pris en entrée est supposé connexe, on a $m \geq n-1$ (Proposition 5) et la complexité de l'algorithme peut finalement s'écrire $O(m \log m)$. □

2.2.2 Algorithme de Prim

L'algorithme de `Prim` prend en entrée un graphe pondéré (G,c) et un sommet r de G. Il enrichit pas à pas un arbre T, initialisé au sommet isolé r, en ajoutant à T une arête de G qui relie $V(T)$ à $V(G) - V(T)$ et qui est minimale pour la fonction de pondération c. Au bout de $|V(G)| - 1$ étapes de calcul, T couvre tous les sommets de G et est un arbre couvrant minimal de G.

Algorithme 5 : Algorithme de `Prim`

Données : Un graphe pondéré connexe (G,c) et un sommet r de G
Résultat : Un ACM de (G,c)
1 $T = (\{s\}, \varnothing)$;
2 **tant que** $V(T) \neq V(G)$ **faire**
3 sélectionner une arête $\{u,v\}$ de plus faible poids parmi celles qui respectent : $u \in V(T)$ et $v \notin V(T)$;
4 $E(T) = E(T) \cup \{u,v\}$;
5 $V(T) = V(T) \cup \{v\}$;
fin
6 **retourner** T

Proposition 14 (Correction de l'algorithme de `Prim`.) *L'algorithme de* `Prim` *renvoie un arbre couvrant minimal de* (G,c).

PREUVE. Notons d'abord que l'algorithme termine sur toute entrée, puisque le test de sa boucle principale (ligne 2) devient nécessairement faux après $|V(G)|-1$ étapes, en vertu de la ligne 5. Montrons par ailleurs que l'assertion suivante est un invariant de la boucle **tant que** :

$$T \text{ est un sous-graphe connexe d'un arbre couvrant minimal de } G. \qquad (*)$$

Supposons que le sous-graphe T obtenu avant un passage dans la boucle vérifie l'assertion $(*)$, et prouvons que tel est encore le cas après ce passage. Soit donc $a = \{u,v\}$ l'arête selectionnée ligne 3. Il est facile de voir que le sous-graphe $T+a$ obtenu après les lignes 4 et 5 est encore connexe (puisque $u \in V(T)$). Montrons que $T+a$ est contenu dans un arbre couvrant minimal de G. Soit T_0 un ACM qui contient T (hypothèse d'induction). Si T_0 contient a, $T+a$ est encore un sous-graphe de T_0 et le résultat cherché est établi. Sinon, T_0+a contient un unique cycle p passant par a (voir Théorème 7). Mais puisque p contient l'arête a qui joint $V(T)$ à $V(G)-V(T)$, il contient nécessairement une arête $b \neq a$ joignant $V(G)-V(T)$ à $V(T)$. Considérons alors le nouveau sous-graphe T_1 obtenu à partir de T_0 en retirant b et en ajoutant a (*i.e.* $T_1 = (T_0+a)-b$). Alors : T_1 est acyclique et connexe, puisqu'on l'a obtenu en retirant une arête (l'arête b) à l'unique cycle du graphe T_0+a. T_1 est couvrant, puisqu'il a même ensemble de sommets que T_0, lui-même couvrant. C'est donc un arbre couvrant de G. T_1 contient $T+a$, puisque T_0 contient T et $b \notin E(T+a)$. Enfin, T_1 est minimal. En effet, d'une part : $c(T_1) = c(T_0)+c(b)-c(a)$ et d'autre part : $c(b) \geq c(a)$ puisque, par la ligne 3 de l'algorithme, a est de poids minimal parmi les arêtes qui joignent $V(T)$ à $V(G)-V(T)$. Ainsi, $c(T_1) \leq c(T_0)$. Autrement dit, puisque T_0 est minimal : $c(T_1) = c(T_0)$, et T_1 est un arbre couvrant minimal qui contient $T+a$. L'assertion $(*)$ est bien un invariant de la boucle **tant que**. Comme de plus cette assertion est vraie après l'initialisation – tout sommet isolé est un sous-graphe connexe de tout arbre couvrant – elle est vérifiée au sortir de la boucle. Il reste, pour conclure, à constater que l'algorithme termine lorsque le test de la boucle est devenu faux, c'est-à-dire lorsque $V(T) = V(G)$. À cet instant, T est donc un arbre *couvrant* inclus dans un ACM de G : c'est un ACM de G. □ Nous allons maintenant détailler l'implémentation de l'algorithme de `Prim`. La gestion dynamique de l'arbre T tire parti de sa

structure d'arbre *enraciné* (en r). Cette structure permet de représenter T par un tableau à une dimension –appelons-le *père*–, comme indiqué Section 1.5. Ainsi, l'ajout d'une arête $\{u,v\}$ à T (avec $u \in V(T)$ et $v \notin V(T)$) se fait *via* l'affectation $père(v) = u$. Lorsque l'algorithme termine, l'arbre calculé est entièrement décrit par le tableau *père*, qui est retourné comme résultat. Pour effectuer le test $V(T) \neq V(G)$, on utilise une variable X représentant l'ensemble $V(G) - V(T)$. Le test précédent devient alors $X \neq \varnothing$. La variable X est initialisée à $V(G)$ et chaque passage dans la boucle **tant que** comporte la suppression de l'un de ses sommets. Notons qu'avec ce choix, l'ensemble $V(T)$ à contruire est l'ensemble complémentaire de X, noté X^c. De plus, chaque arête sélectionnée dans la ligne 3 de l'algorithme est une arête de plus faible poids parmi celles qui joignent X à X^c. L'efficacité de l'algorithme dépend pour l'essentiel de la manière dont cette sélection est faite. Dans l'implémentation proposée plus bas, on maintient un tableau *dist* qui rend compte, pour chaque sommet u, du poids minimal d'une arête joignant u à un sommet de l'ensemble X^c courant. Si u n'est pas connecté à X^c, $dist(u)$ est posé égal à ∞. Le choix d'une arête minimale joignant X à X^c revient alors à sélectionner celui des sommets de X qui minimise $dist(u)$. Pour cette raison, on implémente X comme une file de priorité dépendante du champ *dist*. On dispose alors d'une procédure classique d'extraction du minimum, `Extraire-Min`(X), qui retourne un élément de X sur lequel *dist* prend la valeur minimale, et qui supprime cet élément de la file. On obtient finalement l'implémentation suivante :

Algorithme 6 : Une implémentation de `Prim`

Données : Un graphe pondéré connexe (G,c) et un sommet r de G
Résultat : Un ACM de (G,c)

```
1  X ← V(G) ;
2  pour chaque sommet u ∈ X faire
3      dist(u) = ∞ ;
   fin
4  dist(r) = 0 ;
5  père(r) = nil ;
6  tant que X ≠ ∅ faire
7      u = Extraire-Min(X) ;
8      pour chaque v ∈ adj(u) faire
9          si v ∈ X et c(u,v) < dist(v) alors
10             père(v) = u ;
11             dist(v) = c(u,v) ;
           fin
       fin
   fin
12 retourner père
```

Lemme 15 (Complexité de l'algorithme de `Prim`.) *Sur un graphe à n sommets et m arêtes représenté par listes de successeurs, l'algorithme de* `Prim` *termine en temps $O(m \log n)$.*

PREUVE. Si la file de priorité X est implémentée comme un tas binaire, l'initialisation des lignes 1 à 4 se fait en $O(n)$. Les instructions de la boucle **tant que** sont exécutées pour chaque

sommet $x \in V(G)$. Chaque opération `extraire-min` requiert un temps $O(\log n)$. Par conséquent, le temps total requis pour tous les appels à `extraire-min` est $O(n \log n)$. La boucle **pour** de la ligne 8 est exécutée $\delta(x)$ fois. À l'intérieur de cette boucle, le test de la ligne 9 peut être implémenté en temps constant, en affectant à chaque sommet un bit indiquant s'il appartient ou non à X, et en mettant ce bit à jour lorsque le sommet est supprimé de X. L'affectation de la ligne 10 se fait en temps constant ; celle de la ligne 11 cache une mise à jour du tas qui peut se faire en $O(\log n)$. Ainsi, pour chaque sommet u considéré dans la boucle **tant que**, la boucle **pour** nécessite un temps $O(\delta(u) \log n)$. En sommant ces valeurs sur tous les sommets u examinés, on obtient : $O((\sum_{x \in V(G))} \delta(x)) \log n) = O(2m \log n) = O(m \log n)$. Par conséquent, le temps total utilisé par la boucle **tant que** est en $O(n \log n) + O(m \log n)$. La complexité de l'algorithme de `Prim` est finalement, en tenant compte que $m \geq n-1$ (puisque G est connexe) : $O(n) + O(n \log n) + O(m \log n)] = O(m \log n)$. □

2.3 Matroïdes

Définition 16 (Matroïde). *Un **matroïde** est un couple $M = (X, \mathscr{I})$ constitué d'un ensemble fini non vide X et d'un ensemble de parties de X, $\mathscr{I} \subset \mathscr{P}(X)$, satisfaisant :*

1. $\varnothing \in \mathscr{I}$;
2. *si $I \in \mathscr{I}$ et $I' \subset I$, alors $I' \in \mathscr{I}$ ($\mathscr{I}$ est héréditaire) ;*
3. *si $I, I' \in \mathscr{I}$ et $|I| < |I'|$, alors il existe $x \in I' \setminus I$ t.q. $I \cup \{x\} \in \mathscr{I}$ (propriété d'échange).*

*Les éléments de $\mathscr{I}$ sont appelés **indépendants** de M.*

EXEMPLES.
- *Matroïde linéaire :* Soit A une matrice carrée de rang n. Notons X l'ensemble des vecteurs colonnes de A et $\mathscr{I}$ l'ensemble des parties de X linéairement indépendantes. Alors le couple $(X, \mathscr{I})$ est un matroïde.
- *Matroïde graphique :* Sur un graphe non orienté $G = (V, E)$, on considère l'ensemble $\mathscr{I}$ des ensembles acycliques d'arêtes. Alors $(E, \mathscr{I})$ est un matroïde.

Définition 17 (Bases d'un matroïde). *Étant donnés un matroïde $M = (X, \mathscr{I})$ et un indépendant $F \in \mathscr{I}$, un élément $x \notin F$ est une **extension** de F si $F \cup \{x\} \in \mathscr{I}$. Un indépendant qui n'admet aucune extension (i.e. qui est maximal pour l'inclusion) est appelé **base** du matroïde.*

EXEMPLES.
- les bases du matroïde linéaire associé à la matrice A sont les bases, au sens vectoriel, de l'espace engendré par les colonnes de A.
- les bases du matroïde graphique associé à $G = (V, E)$ sont les sous-arbres couvrants de G.

Lemme 18 *Les bases d'un matroïde ont toutes même cardinal.*

PREUVE. Dans le cas contraire, soient A, B deux bases telles que $|A| < |B|$. Alors, par propriété d'échange, il existe $x \in B \setminus A$ tel que $A \cup \{x\} \in \mathscr{I}$, ce qui contredit la maximalité de A. □

Définition 19 (Pondération d'un matroïde). *Un matroïde $M = (X, \mathscr{I})$ est dit **pondéré** s'il est doté d'une fonction de pondération $w : X \to \mathbb{R}^+$. Le poids d'une partie A de X est alors définie par sommation : $w(A) = \sum_{x \in A} w(x)$.*

Un grand nombre de problèmes d'optimisation pour lesquels l'approche gloutonne fonctionne peuvent être formulés en terme de recherche d'un ensemble indépendant de poids maximal dans un matroïde. Notons que la pondération d'un matroïde étant positive et additive, un indépendant de poids maximal est toujours une base. En effet, si $I \in \mathscr{I}$ est non maximal, il admet une extension x, et $I \cup \{x\}$ est alors un indépendant de poids $w(I) + w(x)$ supérieur à $w(I)$. Nous allons montrer maintenant que le calcul d'un indépendant de poids maximal dans un matroïde peut être effectué de manière gloutonne.

Algorithme 7 : Algorithme `glouton`

Données : Un matroïde pondéré $M = (X, \mathscr{I}, w)$
Résultat : Un indépendant de poids maximal de M

```
1 F = ∅ ;
2 trier X par poids décroissants ;
3 pour chaque x ∈ X pris dans cet ordre faire
4     si F + x ∈ I alors
5         F = F + x ;
      fin
  fin
6 retourner F
```

Proposition 20 *Sur chaque matroïde pondéré $M = (X, \mathscr{I}, w)$, `glouton`$(X, \mathscr{I}, w)$ retourne un indépendant de M de poids maximal.*

PREUVE. Soit $G = \{g_1, \ldots, g_k\}$ la solution retournée par `glouton`$(X, \mathscr{I}, w)$. Il est clair que $G \in \mathscr{I}$. De plus, G est maximal pour l'inclusion, car s'il admettait une extension x, `glouton` l'aurait considérée et ajoutée à G. Donc G est une base. Supposons, en vue d'une contradiction, que G n'est pas de poids maximal. Soit alors H un indépendant de poids maximal. D'après la remarque qui suit la Définition 19, H est nécessairement une base et donc, grâce au Lemme 18, de même taille que G. Notons $H = \{h_1, \ldots, h_k\}$ et supposons, sans perte de généralité, que les indices des éléments, dans G et H, correpondent à un tri de ces ensembles par poids décroissants. Soit i le premier indice pour lequel $w(g_i) < w(h_i)$ (l'existence de cet indice est garantie par l'hypothèse que $w(G) < w(H)$). Notons alors $G_{i-1} = \{g_1, \ldots, g_{i-1}\}$ et $H_i = \{h_1, \ldots, h_{i-1}, h_i\}$. Par la propriété d'échange, il existe $h_j \in H_i \setminus G_{i-1}$ tel que $G_{i-1} \cup \{h_j\}$ est indépendant. On a $w(h_j) \geq w(h_i) > w(g_i)$. Mais alors, h_j est un élément de poids supérieur à celui de g_i et tel que $\{g_1, \ldots, g_{i-1}, h_j\}$ est indépendant. Donc cet élément aurait dû être examiné et intégré à G, avant l'examen de g_i, ce qui contredit l'hypothèse $h_j \notin G_{i-1}$: c'est la contradiction cherchée. □

Définition 21 (Système de parties). *Un **système de parties** $M = (X, \mathscr{I})$ est la donnée d'un ensemble fini X et d'une famille non vide $\mathscr{I} \subseteq \mathscr{P}(X)$ qui vérifie la propriété d'hérédité :*

$$(A \subseteq B \text{ et } B \in \mathscr{I}) \Rightarrow A \in \mathscr{I}.$$

Proposition 22 *Si un système de parties $M = (X, \mathscr{I})$ n'est **pas** un matroïde, alors il existe une pondération des élément de X pour laquelle `glouton`$(X, \mathscr{I}, w)$ ne retourne pas un élément de $\mathscr{I}$ de poids maximal.*

PREUVE. Si $M = (X, \mathscr{I})$ n'est pas un matroïde, il contient deux parties $A, B \in \mathscr{I}$ qui contredisent la propriété d'échange, *i.e.* qui satisfont $|A| > |B|$ et $B \cup \{x\} \notin \mathscr{I}$ pour tout $x \in A \setminus B$. Notons $m = |B|$ et définissons une pondération w sur M par :

- $w(x) = m+2$ pour $x \in B$;
- $w(x) = m+1$ pour $x \in A \setminus B$;
- $w(x) = 0$ pour $x \notin A \cup B$.

Il est facile de voir que sur le système pondéré $(X, \mathscr{I}, w)$ ainsi défini, `glouton` examine et intègre chaque $y \in B$ avant de rejeter tous les $x \in A$ (puisque pour chaque tel x, $B \cup \{x\} \notin \mathscr{I}$). Ainsi, l'ensemble retourné par `glouton`$(X, \mathscr{I}, w)$ est de poids $m(m+2)$. Or :

$$w(A) = |A|(m+1) \geq (m+1)(m+1) \text{ (car } |A| > |B| = m)$$

et donc :

$$w(A) \geq m^2 + 2m + 1 > m(m+2).$$

Ce qui prouve que `glouton`$(X, \mathscr{I}, w)$ n'est pas de poids maximal parmi les ensembles de $\mathscr{I}$. □

Chapitre 3

Programmation Dynamique

Le but de ce chapitre est de donner, à travers une série d'exemples, la description d'un paradigme algorithmique très puissant : la **programmation dynamique**. Ce terme ne relève pas d'une définition formelle. Il recouvre une famille d'algorithmes partageant un certain nombre de caractéristiques que nous détaillerons au long de ce chapitre. À l'instar de la programmation linéaire, la programmation dynamique a un champ d'application très large.

3.1 Plus courts chemins dans un DAG

On appelle DAG un graphe orienté sans circuit (*Directed Acyclic Graph*, en anglais). Dans un tel graphe, muni d'une valuation des arêtes, il est facile de calculer les plus courts chemins à partir d'un sommet fixé, en s'appuyant sur un tri topologique du graphe. Considérons par exemple le graphe de la Figure 3.1. Supposons que l'on cherche à calculer les plus courts chemins d'origine s. Pour chaque sommet y, la valeur $\mathrm{d}_s(y)$ d'un plus court chemin de s à y s'obtient en prenant le min des valeurs $dist(x,y) + \mathrm{d}_s(x)$ pour tous les prédécesseurs x de y dans le graphe. Ainsi,

$$\mathrm{d}_s(d) = \min\{\mathrm{d}_s(b)+1, \mathrm{d}_s(c)+3\}.$$

Le calcul de $\mathrm{d}_s(x)$ pour un sommet donné suppose donc que cette fonction a préalablement été évaluée pour tous les prédécessueur de x. C'est précisément l'existence d'un tri topologique du graphe qui nous garantit que les calculs successifs vont pouvoir se faire sans conflit, *i.e.* que l'on ne se retrouvera pas dans une situation où le calcul de $\mathrm{d}_s(x)$ suppose la connaissance de $\mathrm{d}_s(y)$ qui à son tour, exige la donnée de $\mathrm{d}_s(x)$. On peut calculer toutes les distances $\mathrm{d}_s(x)$ en une passe,

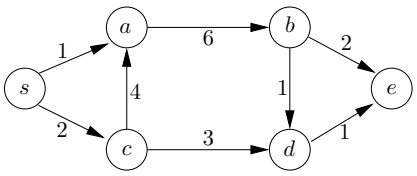


FIGURE 3.1 – Un DAG

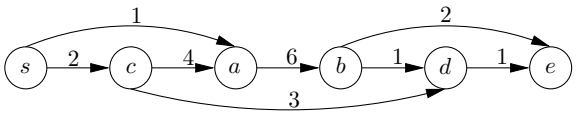


FIGURE 3.2 – Un DAG trié topologiquement

à condition d'effectuer ces calculs dans l'ordre topologique croissant des sommets, représenté Figure 3.2 :

Algorithme 8 : $\texttt{pccDag}^{\texttt{iter}}(G,s)$

Données : Un DAG pondéré (V,E,ℓ) et un sommet $s \in V$
Résultat : Les plus courts chemins d'origine s

1 $\mathrm{d}_s(s) = 0$; **pour** $v \in V \setminus \{s\}$ **faire** $\mathrm{d}_s(v) = \infty$;
2 Trier topologiquement V ;
3 **pour** $v \in V$ *pris dans l'ordre topologique* **faire**
4 $\quad \mathrm{d}_s(v) = \inf_{uv \in E} \{\mathrm{d}(u) + \ell(u,v)\}$;

fin

3.2 Sous suite commune maximale

Définition 23 *Une* sous-suite *d'une suite $X = x_1,\ldots,x_n$ est une suite $Z = x_{i_1},\ldots,x_{i_p}$ extraite de X (i.e. $i_1 < \cdots < i_p$).*

Problème : Problème de la plus longue sous-suite commune
Input : deux suites X et Y (sur un ensemble quelconque)
Output : la longueur maximale des sous-suites communes à X et Y.

L'ensemble des sous-suites communes de X et Y de longueur maximale est noté $\mathrm{PLSC}(X,Y)$.

EXEMPLE. Pour $X = abcbdab$ et $Y = bdcaba$, $\mathrm{PLSC}(X,Y) = \{bcba, bcab\}$.

Sous-structure optimale des PLSC. Soit $X = x_1,\ldots,x_m$. Pour chaque $i \leq m$, on note $X_i = x_1,\ldots,x_i$.

Théorème 24 *Soient $X = x_1,\ldots,x_m$, $Y = y_1,\ldots,y_n$ et $Z = z_1,\ldots,z_k \in \mathrm{PLSC}(X,Y)$.*
- *Si $x_m = y_n$ alors $z_k = x_m$ et $Z_{k-1} \in \mathrm{PLSC}(X_{m-1}, Y_{n-1})$;*
- *Si $x_m \neq y_n$ et $z_k \neq x_m$ alors $Z \in \mathrm{PLSC}(X_{m-1}, Y)$;*
- *Si $x_m \neq y_n$ et $z_k \neq y_n$ alors $Z \in \mathrm{PLSC}(X, Y_{n-1})$;*

Si $\mathrm{lg}(i,j)$ dénote la longueur d'une PLSC de X_i et Y_j, on a donc :

$$\mathrm{lg}(i,j) = \begin{cases} 0 & \text{si } i = 0 \text{ ou } j = 0 \\ 1 + \mathrm{lg}(i-1, j-1) & \text{si } i,j > 0 \text{ et } x_i = y_j \\ \max\{\mathrm{lg}(i, j-1), \mathrm{lg}(i-1, j)\} & \text{si } i,j > 0 \text{ et } x_i \neq y_j \end{cases}$$

Un algo récursif pour PLSC.

Algorithme 9 : $\texttt{plsc}^{\texttt{rec}}(X,Y,i,j)$

si $i = 0$ *ou* $j = 0$ **alors**
  retourner 0
sinon
  si $X[i] = Y[j]$ **alors**
    retourner $1 + \texttt{plsc}^{\texttt{rec}}(X,Y,i-1,j-1)$
  sinon
    retourner $\max\{\texttt{plsc}^{\texttt{rec}}(X,Y,i-1,j), \texttt{plsc}^{\texttt{rec}}(X,Y,i,j-1)\}$
  fin
fin

Complexité : $T(n) = 2T(n-1) + c$, d'où $T(n) = O(2^n)$.

Un algo itératif pour PLSC.

Algorithme 10 : $\texttt{plsc}^{\texttt{iter}}(X,Y)$

$m = \lg(X)$; $n = \lg(Y)$;
pour $i = 0$ *à* m **faire** $\lg[i,0] = 0$;
pour $j = 0$ *à* n **faire** $\lg[0,j] = 0$;
pour $i = 1$ *à* m **faire**
  pour $j = 1$ *à* n **faire**
    si $x_i = y_j$ **alors**
      $\lg[i,j] = 1 + \lg[i-1,j-1]$
    sinon
      $\lg[i,j] = \max\{\lg[i-1,j], \lg[i,j-1]\}$
    fin
  fin
fin
retourner $\lg(m,n)$;

Complexité : $O(mn)$ (en temps et en espace).

EXEMPLE. Exécution de $\texttt{plsc}^{\texttt{iter}}(abcbdab, bdcaba)$

		0 -	1 b	2 d	3 c	4 a	5 b	6 a
0	-	0	0	0	0	0	0	0
1	a	0	0	0	0	1	1	1
2	b	0	1	1	1	1	2	2
3	c	0	1	1	2	2	2	2
4	b	0	1	1	2	2	3	3
5	d	0	1	2	2	2	3	3
6	a	0	1	2	2	3	3	4
7	b	0	1	2	2	3	4	4

	0 -	1 b	2 d	3 c	4 a	5 b	6 a
0 -	0	0	0	0	0	0	0
1 a	0	$\uparrow 0$	$\uparrow 0$	$\uparrow 0$	$\nwarrow 1$	$\leftarrow 1$	$\nwarrow 1$
2 b	0	$\nwarrow 1$	$\leftarrow 1$	$\leftarrow 1$	$\uparrow 1$	$\nwarrow 2$	$\leftarrow 2$
3 b	0	$\uparrow 1$	$\uparrow 1$	$\nwarrow 2$	$\leftarrow 2$	$\uparrow 2$	$\uparrow 2$
4 b	0	$\nwarrow 1$	$\uparrow 1$	$\uparrow 2$	$\uparrow 2$	$\nwarrow 3$	$\leftarrow 3$
5 d	0	$\uparrow 1$	$\nwarrow 2$	$\uparrow 2$	$\uparrow 2$	$\uparrow 3$	$\uparrow 3$
6 a	0	$\uparrow 1$	$\uparrow 2$	$\uparrow 2$	$\nwarrow 3$	$\uparrow 3$	$\nwarrow 4$
7 b	0	$\nwarrow 1$	$\uparrow 2$	$\uparrow 2$	$\uparrow 3$	$\nwarrow 4$	$\uparrow 4$

3.3 Sac-à-dos

Le *problème du sac à dos* se définit de la manière suivante : on dispose d'objets de type $i = 1, \ldots, n$ qui ont chacun un poids p_i et une valeur v_i. On veut remplir un sac à dos de contenance maximale C en maximisant la valeur totale $V_{\max}$ du paquetage. Ce problème est NP-complet. Mais on peut trouver un algorithme "pseudo-polynomial", de complexité $O(nC)$. Le problème admet deux variantes : dans la première (sac à dos *avec répétition*), on suppose que l'on dispose d'une quantité non bornée d'objets de chaque type. Dans la seconde (sac à dos *sans répétition*), on impose au contraire que la quantité d'objets de chaque type soit préalablement fixée.

Sac à dos avec répétition. Comment définir les sous-problèmes pertinents de ce problème ? En limitant les types d'objets considérés ? En réduisant la contenance maximale ? C'est cette deuxième option qui nous sera utile ici. Notons $V_{\max}(c)$ la valeur maximale d'un sac à dos de capacité c. Considérons un chargement optimal X pour cette capacité, et un objet x participant à ce chargement (*i.e.* $x \in X$). Clairement, $p(X-x) = p(X) - p(x) \leq c - p(x)$, puisque $p(X) \leq c$. Ainsi, $X - x$, constitue un chargement compatible avec la capacité $c - p(x)$. De plus, ce chargement est optimal : dans le cas contraire, un chargement optimal Y pour $c - p(x)$ donnerait lieu au chargement $Y + x$ compatible avec c et de valeur $v(Y) + v(x) > v(X-x) + v(x) = v(X)$, contredisant ainsi l'optimalité de X pour c. Finalement, on obtient :

– $v(X) = V_{\max}(c)$, puisque X est un chargement optimal pour c ;
– $v(X-x) = V_{\max}(c-p(x))$, puisque $X-x$ est un chargement optimal pour $c-p(x)$;
– $v(X) = v(X-x)+v(x)$, par additivité de la fonction v.

Il s'ensuit :

$$V_{\max}(c) = V_{\max}(c-p(x))+v(x).$$

Ce constat, formulé en termes de types d'objets, s'écrit : pour toute capacité c et tout type i intervenant dans un chargement optimal pour c :

$$V_{\max}(c) = V_{\max}(c-p_i)+v_i. \tag{3.1}$$

Cette équation permet de d'exprimer la solution du problème SAC-À-DOS$_r(c)$ en fonction de celle de l'un de ses sous-problèmes, SAC-À-DOS$_r(c-p_i)$. La difficulté, c'est que lorsque nous cherchons à résoudre SAC-À-DOS$_r(c)$, nous ne connaissons ni ses chargement optimaux, ni les types intervenant dans ces chargements. Par conséquent, nous ne sommes pas en mesure de déterminer les entrées $c-p_i$ pour lesquelles l'égalité (3.1) est réalisée. En l'état, cette égalité ne nous est donc d'aucune aide. Comme dans les problèmes précédents, nous souhaiterions pouvoir tenir le raisonnement suivant : puisque $V_{\max}(c)$ est l'une des valeurs $V_{\max}(c-p_i)+v_i$, où $i \in \{1,\ldots,n\}$, et puisque $V_{\max}(c)$ est défini comme une valeur *maximale*, prenons pour solution la valeur maximale des $V_{\max}(c-p_i)+v_i$, quand i parcourt $\{1,\ldots,n\}$. Autrement dit, nous voudrions pouvoir écrire : $V_{\max}(c) = \max_i\{V_{\max}(c-p_i)+v_i\}$. En fait, cette écriture est illicite, puisque le terme $V_{\max}(c-p_i)$ n'a de sens que pour une capacité $c-p_i$ positive ou nulle. On contourne aisément cette difficulté en ne considérant dans le "max" que les types i pour lesquels $p_i \leq c$. Finalement, l'égalité visée s'écrit :

$$V_{\max}(c) = \max_{i:p_i\leq c}\{V_{\max}(c-p_i)+v_i\} \tag{3.2}$$

Pour établir la validité de (3.2), il nous faut répondre à une dernière questions : pour un type i intervenant dans un chargement optimal pour c, nous avons vu que $V_{\max}(c) = V_{\max}(c-p_i)+v_i$. Qu'en est-il lorsque i n'intervient dans aucun chargement optimal pour c ? Se peut-il qu'alors, $V_{\max}(c-p_i)+v_i$ soit strictement supérieure à $V_{\max}(c)$? Dans ce cas, (3.2) s'avérerait fausse. Nous montrons maintenant que cette situation ne se produit en fait jamais : pour tout type i satisfaisant $p_i \leq c$,

$$V_{\max}(c) \geq V_{\max}(c-p_i)+v_i.$$

En effet :

- Si i intervient dans un chargement optimal pour c, l'égalité, et donc *a fortiori* l'inégalité large, entre ces deux quantités a été établie.

- Sinon, considérons un objet x de type i, et un chargement X optimal pour la capacité $c-p_i$. On a, par définition d'un chargement optimal : $p(X) \leq c-p_i$ et $v(X) = V_{\max}(c-p_i)$. Par ailleurs, $X+x$ est un chargement compatible avec c, puisque $p(X+x) = p(X)+p(x) = p(X)+p_i \leq c-p_i+p_i = c$. Par conséquent, la valeur de $X+x$ est inférieure à la valeur d'un chargement optimal pour c. *I.e.* $v(X+x) \leq V_{\max}(c)$. Or : $v(X+x) = v(X)+v(x) = V_{\max}(c-p_i)+v_i$. L'inégalité attendue s'en déduit.

L'égalité (3.2) ainsi établie fournit une relation inductive entre problèmes et sous-problèmes de SAC-À-DOS$_r$. Cette relation permet à la fois de formuler l'algorithme récursif suivant et d'en garantir la correction.

Algorithme 11 : $\texttt{sad}^{\texttt{rec}}_{\texttt{rep}}(C)$

Données : Un ensemble de types $(p_i, v_i)_{i \leq n}$ et une capacité C
Résultat : La valeur $V_{\max}(C)$ d'un chargement optimal pour C
si $C = 0$ **alors**
  retourner 0
sinon
  retourner $\max_{i:p_i \leq C} \{\texttt{sad}^{\texttt{rec}}_{\texttt{rep}}(C - p_i) + v_i\}$
fin

Complexité : exponentielle.

Algo itératif pour sac à dos avec répétition
L'approche "programmation dynamique" s'incarne dans l'algorithme itératif suivant :

Algorithme 12 : $\texttt{sad}^{\texttt{iter}}_{\texttt{rep}}(C)$

Données : Un ensemble de types $(p_i, v_i)_{i \leq n}$ et une capacité C
Résultat : La valeur $V_{\max}(C)$ d'un chargement optimal pour C
$V_{\max}[0] = 0$;
pour $c = 1$ *à* C **faire**
  $V_{\max}[c] = \max\{V_{\max}[c - p_i] + v_i : p_i \leq c\}$
fin
retourner $V_{\max}(C)$

Complexité : $O(nC)$. (On remplit un tableau de $C+1$ cases. Le traitement de chaque case peut nécessiter un temps $O(n)$.)

La version "récursivité + mémorisation" est de même complexité et donnerait ici de meilleures performances.

Problème du sac à dos sans répétition On dispose de n objets $1, \ldots, n$ qui ont chacun un poids p_i et une valeur v_i. On veut remplir un sac à dos de contenance maximale C en maximisant la valeur totale $V_{\max}$ du paquetage. La relation (3.2) invoquée pour définir les sous-problèmes de "sac à dos avec répétition" n'est plus valide. On a maintenant la relation :

$$V_{\max}(c, j) = \max\{V_{\max}(c - p_j, j-1) + v_j, V_{\max}(c, j-1)\}, \tag{3.3}$$

où $V_{\max}(c, j)$ désigne *la valeur maximale accessible pour un sac de capacité c avec l'ensemble d'objets* $\{1, \ldots, j\}$. On donne directement l'interprétation de cette relation en terme d'algorithme itératif :

Algo itératif pour sac à dos sans répétition **Complexité :** $O(nC)$.

Algorithme 13 : $\texttt{sad}^{\texttt{iter}}_{\texttt{norep}}(C)$

initialiser tous les $V_{\max}(0,j)$ et tous les $V_{\max}(c,0)$ à 0 ;
pour $j = 1$ *à* n **faire**
  pour $c = 1$ *à* C **faire**
    si $p_j > c$ **alors**
      $V_{\max}(c,j) = V_{\max}(c,j-1)$
    sinon
      $V_{\max}(c,j) = \max\{V_{\max}(c,j-1), V_{\max}(c-p_j,j-1)+v_j\}$
    fin
  fin
fin
retourner $V_{\max}(C,n)$

3.4 Sous-suite croissante maximale

FIGURE 3.3 – Une sous-suite croissante

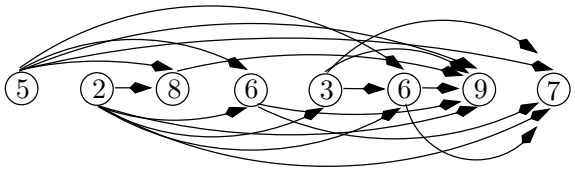


FIGURE 3.4 – Une sous-suite croissante

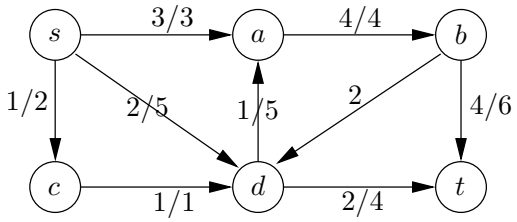
Chapitre 4

Flot Maximal

Un **réseau** est la donnée d'un graphe orienté $G = (V, E)$, d'une fonction $c : E \to \mathbb{N}$ appelée *capacité du réseau*, et de deux sommets $s, t \in V$, respectivement baptisés *source* et *puit* du graphe. Un **flot** dans un réseau (V, E, c, s, t) est une fonction $f : E \to \mathbb{N}$ qui satisfait les deux conditions suivantes :

- *Contrainte de capacité :* pour tout $(u, v) \in E$, $f(u, v) \leq c(u, v)$;
- *Conservation du flot :* pour tout $u \in V \setminus \{s, t\}$, $\sum_{x \in Pred(u)} f(x, u) = \sum_{y \in Succ(u)} f(u, y)$.

Pour $(u, v) \in E$, la quantité $f(u, v)$ est appelée *flux entre u et v*. Dans l'exemple de réseau qui suit, on convient de noter x/y le couple *flux/capacité*. Seule la valeur de la capacité apparaît si le flux correspondant est nul (c'est le cas pour l'arc (b, d) de l'exemple).



La **valeur du flot** f, notée $|f|$, est définie par : $|f| = \sum_{v \in Succ(s)} f(s, v)$. Dans l'exemple ci-dessus : $|f| = 6$.

Probléme du flot maximum Trouver un flot de valeur maximale dans un réseau donné.

Chemins améliorants Soit f un flot sur un réseau (G, c, s, t). Un *st-chemin* est un chemin joignant s à t.

Saturation

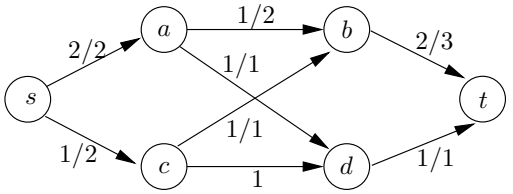
- Un arc (u, v) est *saturé* si $f(u, v) = c(u, v)$.
- Un chemin est *saturé* s'il contient un arc saturé.
- Le flot f est *saturé* si tous les *st*-chemins sont saturés pour ce flot.

Chemin améliorant Si G contient un st-chemin p non saturé, la valeur du flot f peut être améliorée en ajoutant à chaque arc de p une quantité de flot égale à

$$\min_{(u,v)\in p}\{c(u,v)-f(u,v)\}.$$

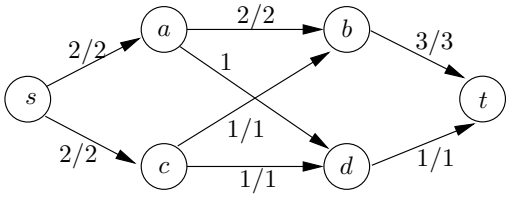
Ainsi, **un flot maximal est nécessairement saturé**. La réciproque est fausse, comme nous l'illustrons ci-aprés.

Saturation vs maximalité Le flot saturé suivant :



($|f|=3$)

peut être amélioré :



($|f|=4$)

Porter un flot à sa valeur maximale par améliorations successives peut nécessiter de réduire le flux sur certains arcs, pour "mieux l'augmenter" sur d'autres.

Réseau résiduel À un flot f dans un réseau $(G=(V,E),c,s,t)$, on associe un réseau $(G_f=(V,E_f),c_f,s,t)$, dit *réseau résiduel* et défini par :

- $(u,v)\in E_f$ ssi $\begin{cases}(u,v)\in E \text{ et } f(u,v)<c(u,v)\\ \text{ou}\\ (v,u)\in E \text{ et } f(v,u)>0\end{cases}$
- $c_f(u,v)=\begin{cases}c(u,v)-f(u,v) & \text{si } (u,v)\in E \text{ et } f(u,v)<c(u,v)\\ f(v,u) & \text{si } (v,u)\in E \text{ et } f(u,v)>0\end{cases}$

Algorithme de Ford-Fulkerson

Proposition 25 *S'il existe un st-chemin p dans G_f, alors la fonction suivante est un flot qui améliore f sur G :*

$$\forall (u,v)\in E : f'(u,v)=\begin{cases}f(u,v) & si\ (u,v)\ et\ (v,u)\notin p\\ f(u,v)+x & si\ (u,v)\in p\\ f(u,v)-x & si\ (v,u)\in p\end{cases}$$

où $x=\min\{c_f(u,v),(u,v)\in p\}$.

Cette proposition fonde l'algorithme de Ford-Fulkerson :

Algorithme 14 : `FordFulkerson`(V,E,c,s,t)

`initialiser` le flot f à 0 ;
tant que *il existe un st-chemin p dans* G_f **faire**
  améliorer f selon p
fin

La correction de `FordFulkerson` s'appuie sur la notion de coupure.

Coupures Une *coupure* dans un réseau (V,E,c,s,t) est une partition (S,T) de V telle que $s \in S$ et $t \in T$.

Soient $X,Y \subseteq V$. Pour toute fonction $h : E \to \mathbb{N}$, on note

$$h(X,Y) = \sum_{(x,y)\in X\to Y} h(x,y), \text{ où } X \to Y = E \cap (X \times Y).$$

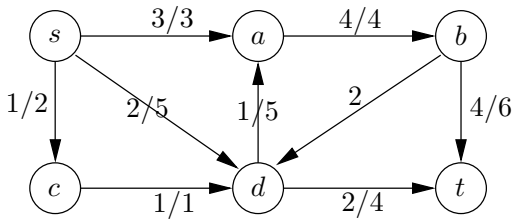
Capacité d'une coupure La *capacité* $c_{S,T}$ *de la coupure* (S,T) est la somme des capacités des arcs joignant S à T :

$$c_{S,T} = c(S,T) = \sum_{(x,y)\in S\to T} c(x,y).$$

Valeur du flot associé à une coupure La *valeur du flot associé à* (S,T) est la somme des flux entre S et T moins la somme des flux entre T et S.

$$f_{S,T} = f(S,T) - f(T,S).$$

Exemples



Si $S = \{s,a,c\}$ et $T = \{b,d,t\}$:

$$c_{S,T} = 4+5+1 = 10 \text{ et } f_{S,T} = (4+2+1)-1 = 6$$

Si $S = \{s,a,b,d\}$ et $T = \{c,t\}$:

$$c_{S,T} = 2+4+6 = 12 \text{ et } f_{S,T} = (1+2+4)-1 = 6$$

Si $S = \{s\}$ et $T = \{a,b,c,d,t\}$:

$$c_{S,T} = 2+5+3 = 10 \text{ et } f_{S,T} = 1+2+3 = 6$$

Lemme 26 *On considére un réseau (V,E,c,s,t) et un flot f sur ce réseau. Pour toute coupure (S,T) du réseau, $f_{S,T} = |f|$. En particulier, toutes les coupures ont même valeur de flot.*

PREUVE. On prouve le résultat par récurrence sur le cardinal de S. Si $|S| = 1$, alors $S = \{s\}$ et le résultat $f_{s,V-s} = |f|$ est acquis par définition de $|f|$. Considérons maintenant $f_{S+x,T-x}$. On a :

$$\begin{aligned} f(S+x,T-x) = &\quad f(S,T) + f(x,T) - f(S,x) \text{ et} \\ f(T-x,S+x) = &\quad f(T,S) - f(x,S) + f(T,x) \end{aligned}$$

D'où :

$$\begin{aligned} f_{S+x,T-x} = &\quad (f(S,T) - f(T,S)) + (f(x,T) + f(x,S)) - (f(S,x) + f(T,x)) \\ = &\quad f_{S,T} + f(x,V) - f(V,x) \\ = &\quad f_{S,T} \end{aligned}$$

□

Corollaire 27 *Si $\delta^-(s) = \delta^+(t) = 0$ alors $|f| = f(s,V) = f(V,t)$.*

Proposition 28 *Pour tout flot f et toute coupure (S,T) : $|f| \leq c_{S,T}$. En particulier, un flot dont la valeur coïncide avec la capacité d'une coupure est maximal.*

Dualité

Théorème 29 (Théoréme (Max Flow/Min Cut)). *Soit f un flot dans un réseau (V,E,c,s,t). Sont équivalents :*

1. *f est un flot maximal ;*
2. *Le réseau résiduel G_f ne comporte pas de st-chemin ;*
3. *$|f| = c_{S,T}$ pour une certaine coupure (S,T).*

Corollaire 30 *L'algorithme de Ford-Fulkerson est correct.*

Complexité de `FordFulkerson` : $O(|E||f_{opt}|)$.

Preuve du théoréme Max Flow/Min Cut $1 \Rightarrow 2$:Si G_f comportait un st-chemin, le flot f pourrait être augmenté selon ce chemin.

$2 \Rightarrow 3$: Notons $S = \{u \in V \text{ t.q. } s \overset{G_f}{\rightsquigarrow} u\}$ et $T = V - S$. Alors :

$s \in S$; $t \in T$ (c'est l'hypothése 2) ; (S,T) est une coupure de G.

De plus, pour tout $(u,v) \in E \cap (S \times T)$, $(u,v) \notin E_f$ (puisque $v \notin S$).

Ceci signifie, par définition de E_f :

– $\forall (u,v) \in E \cap (S \times T) : f(u,v) = c(u,v)$;

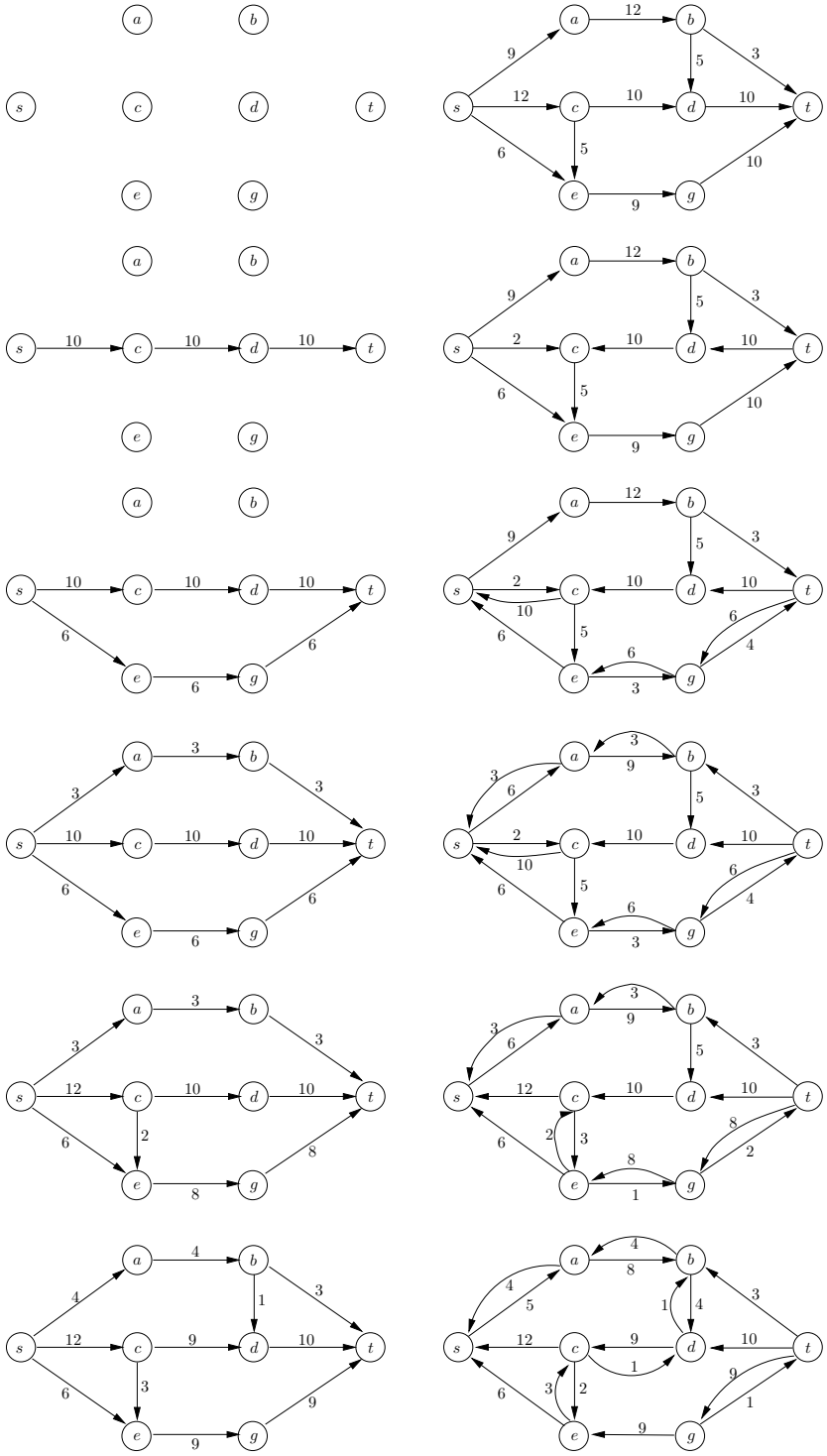
– $\forall (u,v) \in E \cap (T \times S) : f(u,v) = 0$.

Par conséquent :

$$|f| = f_{S,T} = f(S,T) - f(T,S) = c_{S,T} - 0 = c_{S,T}.$$

$3 \Rightarrow 1$: Immédiat, grâce à la proposition précédente.

Une exécution de `FordFulkerson`



Flot et coupure Le flot obtenu a pour valeur 22. Cela correspond à la capacité de la coupure suivante :