Graphes: modélisation et algorithmes

Notes de cours

21 février 2016

Table des matières

1 Notions élémentaires sur les graphes

La théorie des graphes est un outil privilégié de modélisation et de résolution de problèmes dans un grand nombre de domaines allant de la science fondamentale aux applications technologiques concrètes. Par exemple, les graphes déterministes et aléatoires sont utilisés en chimie (modélisation de structure), en sciences sociales (pour représenter des relations entre groupes d'individus), en mathématiques combinatoires, en informatique (structures de données et algorithmique). Concernant les applications, elles sont très nombreuses : réseaux électriques et transport d'énergie, routage du trafic dans les réseaux de télécommunications et les réseaux d'ordinateurs, routage de véhicules et organisation de tournées, problèmes de localisation (localisation d'entrepôts dans les réseaux de distribution de marchandises, d'antennes ...), problèmes d'ordonnancements de tâches et d'affectation de ressources (problèmes de rotation d'équipages dans les compagnies aériennes)...

1.1 Quelques problèmes formalisables par des graphes

1.1.1 Choix d'un itinéraire

Comment faire pour aller le plus rapidement possible de Bordeaux à Grenoble sachant que :

Bordeaux $\rightarrow$ Nantes	4h
Bordeaux $\rightarrow$ Marseille	9h
Bordeaux $\rightarrow$ Lyon	12h
Nantes $\rightarrow$ Paris-Montparnasse	2h
Nantes $\rightarrow$ Lyon	7h
Paris-Montparnasse $\rightarrow$ Paris-Lyon	1h
Paris-Lyon $\rightarrow$ Grenoble	4h30
Marseille $\rightarrow$ Lyon	2h30
Marseille $\rightarrow$ Grenoble	4h30
Lyon $\rightarrow$ Grenoble	1h15

Ce problème est facile à représenter par un graphe dont les arcs sont valuées par les durées des trajets. Il s'agit alors de déterminer un plus court chemin entre Bordeaux et Grenoble.

1.1.2 Organisation d'une session d'examen

8 groupes d'étudiants (numérotés de G1 à G8) doivent passer des examens dans différentes disciplines, chaque examen occupant une demi-journée :

chimie	G1 et G2
Electronique	G3 et G4
Informatique	G3, G5, G6 et G7
Mathématiques	G1, G5, G6 et G8
Physique	G2, G6, G7 et G8

On cherche à organiser la session d'examen la plus courte possible.

On peut représenter chacune des disciplines par un sommet et relier par des arêtes les sommets correspondant aux examens ne pouvant se dérouler simultanément. Le problème est alors de colorier tous les sommets du graphe en utilisant le moins de couleurs possible sachant que deux sommets reliés par une arête doivent être de couleurs distinctes.

1.1.3 Problème d'ordonnancement de tâches

La mise en exploitation d'un nouveau gisement minier demande l'exécution d'un certain nombre de tâches :

Codes	Tâches	Durées	Tâches antérieures
1	Obtention d'un permis d'exploitation	120	-
2	Etablissement d'une piste de 6 km	180	1
3	Transport et installation de 2 sondeuses	3	2
4	Création de bâtiments provisoires pour le bureau des plans et le logement des ouvriers	30	2
5	Goudronnage de la piste	60	2
6	Adduction d'eau	90	4
7	Campagne de sondage	240	3, 4
8	Forage et équipement de 3 puits	180	5, 6, 7
9	Transport et installation du matériel d'exploitation	30	10, 8
10	Construction de bureaux et de logements définitifs ouvriers et ingénieurs	240	6, 5, 7
11	Traçage et aménagement du fond	360	10, 8

Résoudre un problème d'ordonnancement consiste à déterminer le temps minimum de réalisation de l'ensemble, ainsi que l'intervalle de temps dans lequel on doit débuter chacune des tâches pour réaliser l'ensemble dans le temps minimum. Pour cela, on doit rechercher un plus long chemin dans un graphe, appelé graphe potentiels-tâches, dont les sommets représentent les tâches à réaliser et les arcs les contraintes d'antériorité entre les tâches : un arc (i, j) représente la necéssité de réaliser la tâche i avant la tâche j et l'arc (i, j) est valué par la durée d'exécution de i.

1.1.4 Routage dans les réseaux de télécommunications

Le problème de savoir s'il est possible d'acheminer un ensemble d'informations entre deux centres reliés par un réseau de télécommunications revient à chercher un flot de valeur maximale dans un graphe représentant ce réseau (les sommets représentent les centres et les arêtes les liaisons entre ces centres).

1.2 Définitions et concepts de base

1.2.1 Graphe orienté

Un **graphe orienté** (fini) $G = (X, U)$ est défini par :

- Un ensemble (fini) X dont les éléments sont appelés des **sommets** ou des **noeuds**. L'ordre de G est le nombre n de sommets de G. Par la suite, les sommets seront souvent numérotés de 1 à n.
- un ensemble U dont les éléments $u = (i, j)$ sont des couples ordonnés de sommets appelés des **arcs**. On dit que i est l'**extrémité initiale** et j l'**extrémité terminale**, j est **successeur** de i et i est **prédécesseur** de j. Par la suite, on notera $|U| = m$.

Les arcs représentent une relation binaire définie sur $X \times X$.

Une **boucle** est un arc ayant le même sommet comme extrémité initiale et terminale.

La densité d'un graphe est donnée par le quotient $\frac{m}{n^2}$, rapport du nombre effectif d'arcs sur le nombre maximal théorique.

L'ensemble des successeurs de i se note $N_G^+(i)$ ou $\mathbb{N}^+(i)$ quand il n'y a pas d'ambiguïté sur le graphe. L'ensemble de ses prédécesseurs se note $N_G^-(i)$ ou $N^-(i)$. On appelle l'ensemble de ses voisins l'ensemble : $N_G(i) = N_G^+(i) \cup N_G^-(i)$.

Un graphe est parfaitement défini par l'ensemble X et l'application N^+ (ou par N^-). On peut alors le noter $G = (X, N^+)$.

Le **demi-degré extérieur** (resp. **intérieur**) du sommet i, noté $d^+(i)$ (resp. $d^-(i)$) désigne le nombre d'arcs ayant i comme extrémité initiale (resp. terminale). Autrement dit, $d^+(i) = |N^+(i)|$ et $d^-(i) = |N^-(i)|$
Le **degré** du sommet i, noté $d(i)$, est le nombre d'arcs ayant i comme extrémité (les boucles étant comptées deux fois). Ainsi : $d(i) = d^+(i) + d^-(i)$.
Nous avons les relations suivantes :

$$\begin{aligned} \sum_{i \in X} d^+(i) = \sum_{i \in X} d^-(i) &= m \\ \sum_{i \in X} d(i) &= 2m \end{aligned}$$

1.2.2 Graphe non orienté

Un **graphe non orienté** $G = (X, E)$ est défini par :
- L'ensemble X de sommets.
- L'ensemble E dont les éléments $e = ij$ sont des paires (couples non ordonnés) de sommets appelés des **arêtes**. Les sommets reliés à i par une arête sont appelés ses voisins.

Les arêtes représentent une relation binaire symétrique définie sur $X \times X$.

Un graphe est dit **simple** s'il est non orienté et sans boucle.

Comme précédemment, le **degré** du sommet i, noté $d(i)$, est le nombre d'arêtes ayant i comme extrémité (les boucles étant comptées deux fois). Nous avons donc également :

$$\sum_{i \in X} d(i) \quad = \quad 2m$$

1.2.3 Définitions complémentaires

Deux arcs (arêtes) sont dits **adjacents** s'ils ont au moins une extrémité commune. Deux sommets reliés par un arc (une arête) sont dits **voisins** ou **adjacents**. L'arc (i, j) (l'arête ij) est dite **incidente** aux sommets i et j.

Soit $G = (X, E)$ un graphe non orienté et soit $X' \subseteq X$ un sous-ensemble de sommets. Le **sous-graphe engendré (ou induit)** par X' est le graphe $G[X']$ dont les sommets sont les éléments de X' et les arêtes les éléments de E ayant leur deux extrémités dans X'.

Un **graphe partiel** de G est graphe graphe $H = (X, E')$ avec $E' \subseteq E$.

Un **sous-graphe** de G est un graphe $H = (X', E')$ avec $X' \subseteq X$ et $E' \subseteq E$.

Un graphe (orienté ou non) est dit **valué** quand ses arcs/arêtes et/ou ses sommets sont dotés d'un poids (ou longueur).

Un **multigraphe** orienté (non orienté) est une généralisation du concept de graphes au cas où il peut exister plusieurs arcs (arêtes) entre deux sommets i et j donnés. Un **p-graphe** est un multigraphe dans lequel il n'existe jamais plus de p arcs (arêtes) de la forme (i, j) entre i et j. Ainsi, un graphe est un 1-graphe.

Soiut $G = (X, U)$ un graphe orienté et soit $A \subseteq X$ un sous-ensemble de sommets.
- $\omega^+(A)$ est l'ensemble des arcs ayant leur extrémité initiale dans A et leur extrémité terminale dans $X \setminus A$;
- $\omega^-(A)$ est l'ensemble des arcs ayant leur extrémité terminale dans A et leur extrémité

initiale dans $X \setminus A$.
Un **cocycle** $\omega(A)$ du graphe est : $\omega(A) = \omega^+(A) \cup \omega^-(A)$.

1.3 Connexité

1.3.1 Chemin et circuit (graphes oriéntés)

Un **chemin** allant d'un sommet i_0 à un sommet i_q est une suite de sommets $(i_0, i_1, i_2, \cdots, i_q)$ telle que pour tout $k = 1, 2, \cdots, q$, $(i_{k-1}, i_k) \in U$. Le nombre d'arcs q du chemin est la longueur du chemin. On note aussi parfois la succession des arcs traversés : $((i_0, i_1), (i_1, i_2), \cdots, (i_{q-1}, i_q))$. Parfois, on considère qu'un sommet est un chemin (dit trivial) de longueur 0.

Un **circuit** est un chemin (non trivial) dont les extrémités coïncident, i.e. $i_0 = i_q$, et qui ne contient pas deux fois le même arc.

1.3.2 Chaîne et cycle

Une **chaîne** dans un graphe non orienté entre un sommet i_0 et un sommet i_q est une suite de sommets $(i_0, i_1, i_2, \cdots, i_q)$ telle que pour tout $k = 1, 2, \cdots, q$, $i_{k-1}i_k$ soit une arête du graphe. Le nombre d'arêtes q est la longueur de la chaîne. i_0 et i_q sont appelées les extrémités de la chaîne μ. On décrit aussi parfois une chaîne par la succession des arêtes traversées. Parfois on considère qu'un sommet est une chaîne (dite triviale) de longueur 0.
Dans un graphe orienté, la notion de chaîne existe aussi : on s'autorise à prendre les arcs dans n'importe quel sens ((i_{k-1}, i_k) ou (i_k, i_{k-1}) est un arc).

Un **cycle** est une chaîne (non triviale) dont les extrémités coïncident et qui ne contient pas deux fois la même arête (le même arc dans le cas orienté).
Une chaîne (un chemin) est **simple** si les arêtes (les arcs) qui la (le) composent sont tous distincts.
Une chaîne, un chemin, un cycle ou un circuit est dit :

1. **élémentaire** si les sommets qui le composent sont tous distincts (à l'exception des extrémités),
2. **hamiltonien** s'il passe une fois et une seule par chaque sommet du graphe,
3. **eulérien** s'il passe une fois et une seule par chaque arc/arête du graphe,
4. **préhamiltonien** s'il passe au moins une fois par chaque sommet du graphe,
5. **préeulérien** s'il passe au moins une fois par chaque arc/arête du graphe.

1.3.3 Graphe connexe et composantes connexes

Un graphe G est dit **connexe** si pour toute paire de sommets i, j dans G, il existe une chaîne entre i et j.

La relation R definie par

$$iRj \iff \text{il existe une chaîne (éventuellement triviale) entre } i \text{ et } j$$

est une relation d'équivalence car elle est réflexive (iRi), symétrique ($iRj \Rightarrow jRi$ car il suffit d'inverser la chaîne) et transitive (iRj et $jRk \Rightarrow iRk$ car il suffit de concaténer les 2 chaînes).

Les classes d'équivalence induites sur X par R forment une partition de X en $X_1, X_2, \ldots, X_p$. Le nombre p de classes d'équivalence est appelé le **nombre de connexité** du graphe. Un graphe est dit connexe si et seulement si son nombre de connexité est égal à 1. Les sous-graphes $G_1, G_2, \ldots, G_p$ engendrés par les sous-ensembles $X_1, X_2, \ldots, X_p$ sont appelés les **composantes connexes** de G. Chaque composante connexe est un graphe connexe.

La vérification de la connexité d'un graphe est un des premiers problèmes de la théorie des graphes.

1.3.4 Graphe fortement connexe et composantes fortement connexes

Un graphe orienté $G = (X, U)$ est dit **fortement connexe** si, étant donné deux sommets quelconques i et j, il existe à la fois un chemin d'extrémité initiale i et d'extrémité terminale j et un chemin d'extrémité initiale j et d'extrémité terminale i.
Considérons la relation R' définie comme suit :

$$iR'j \iff \text{(il existe un chemin de } i \text{ à } j \text{ et un chemin de } j \text{ à } i)$$

Cette relation est une relation d'équivalence car elle est réflexive, symétrique et transitive. Les classes d'équivalence induites sur X par R' forment une partition de X en $X_1, \ldots, X_q$. q est appelé le **nombre de connexité forte** du graphe. Les sous-graphes $G_1, \ldots, G_q$ de G engendrés par $X_1, \ldots, X_q$ sont fortement connexes et sont appelés les **composantes fortement connexes** de G. Un graphe est fortement connexe si et seulement s'il n'a qu'une seule composante fortement connexe.

On appelle le **graphe réduit**, noté G_r, le graphe défini comme suit : les sommets de G_r représentent les composantes fortement connexes et il existe un arc d'un sommet x_i à un sommet x_j ($i \neq j$) s'il existe au moins un arc d'un sommet de X_i à un sommet de X_j dans le graphe G.

Propriété 1.1 *G_r est nécessairement sans circuit.*

Preuve. La preuve est basée sur la propriété suivante : si (x_1, x_2) est un arc de G_r, alors pour tout couple de sommets $(i_1, i_2) \in X_1 \times X_2$ il existe un chemin de i_1 à i_2.
Démontrons tout d'abord cette propriété. Par définition, l'arc (x_1, x_2) signifie qu'il existe deux sommets $j_1 \in X_1$ et $j_2 \in X_2$ tels que $(j_1, j_2) \in G$. Or, il existe un chemin de i_1 à j_1 car ils sont dans la même CFC. De même, il existe un chemin de i_2 à j_2. On a donc bien un chemin de i_1 à i_2.
La preuve de la propriété 1.1 est alors immédiate. Supposons que l'on ait un circuit $(x_1, x_2, \ldots, x_k, x_1)$. Soit $j_i \in X_i$ pour $i = 1, \ldots, k$. Alors, il existe d'une part un chemin de j_k à j_1, d'autre part un chemin de j_1 à j_2, de j_2 à j_3,..., de j_{k-1} à j_k, donc un chemin de j_1 à j_k. Ces deux sommets devraient alors être dans la même CFC. □

1.4 Quelques graphes particuliers

1.4.1 Graphe complet, clique, stable

- Un graphe orienté G est dit **complet** si pour toute paire i, j de sommets (avec $i \neq j$) il existe au moins un arc (i, j) ou (j, i). Ainsi, un graphe est complet si et seulement si pour $i \neq j$: $(i, j) \notin U \Rightarrow (j, i) \in U$.
- Dans un graphe non orienté, un sous-ensemble de sommets $K \subseteq X$ tel que toute paire de sommets $i, j \in K$ ($i \neq j$) est reliée par une arête est appelé une **clique**. Un sommet isolé constitue à lui seul une clique. Une clique d'ordre n est noté K_n.
- Dans un graphe non orienté, un sous-ensemble de sommets $S \subseteq X$ tel qu'aucune paire de sommets $i, j \in S$ n'est reliée par une arête est appelé un **stable**. Un sommet isolé constitue à lui seul un stable.

1.4.2 Arbre et forêt

Un **arbre** est un graphe connexe sans cycle (en particulier sans boucle). Par exemple, une chaîne élémentaire est un arbre.

Théorème 1.1 *Soit $G = (X, E)$ un graphe d'ordre n. Alors les propriétés suivantes sont équivalentes :*

1. *G est un arbre ;*
2. *G contient $n - 1$ arêtes et G est sans cycle ;*
3. *G contient $n - 1$ arêtes et G est connexe.*

Preuve. Voir TD. □

Propriété 1.2 *Un graphe $G = (X, E)$ est un arbre si et seulement si pour toute paire de sommets distincts il existe une unique chaîne simple les reliant.*

Preuve. Si pour toute paire de sommets distincts de G il existe une unique chaîne simple les reliant, alors le graphe est connexe. Si nous avions un cycle, cela donnerait deux chaînes simples (rappelons qu'un cycle est une chaîne simple) entre deux sommets quelconques du cycle. Donc on n'a pas de cycle. Ainsi G est un arbre.
Réciproquement, supposons qu'il existe deux chaînes simples distinctes $C_1 = (i, i_1, \ldots, i_q, j)$ (suite des sommets rencontrés dans la chaîne) et $C_2 = (i, j_1, \ldots, j_l, j)$ entre deux sommets i et j ($i \neq j$). Les chaînes étant distinctes, quite à les réduire nous pouvons supposer $i_1 \neq j_1$. Soit alors z le premier sommet (relativement à l'ordre donné dans l'écriture ci-dessus) de C_1 autre que i rencontré en parcourant C_2 à partir de i (si aucun des i_k n'égale un des j_t, nous avons $z = j$). Alors les deux chaînes $(i, i_1, \ldots, z)$ et $(i, i_2, \cdots, z)$ forment un cycle (on ne rencontre pas deux fois le même sommet, donc a fortiori pas deux fois la même arête). □

Dans un arbre, une **feuille** est un sommet de degré 1.

Montrons maintenant que tout arbre (ayant au moins deux sommets) a au moins deux feuilles. Dans un graphe simple on a $\sum d(i) = 2m$ donc dans un arbre $\sum d(i) = 2n - 2$. Comme $d(i) \geq 1$ (connexité et au moins 2 sommets), alors si pour au moins $n - 1$ sommets $d(i) \geq 2$, on aurait $\sum d(i) \geq 2(n-1) + 1 > 2n - 2$.

Une **forêt** est un graphe sans cycle (non forcément connexe). En d'autres termes, c'est un graphe dont chaque composante connexe est un arbre.

1.4.3 Arborescence

Une **arborescence** de racine r est un graphe connexe orienté tel que, pour tout sommet j de l'arborescence (autre que r), il existe un chemin unique allant de r à j. Un sommet i quelconque sur le chemin allant de r à j est appelé **ancêtre** de j et j est appelé un **descendant** de i. Si i est le prédécesseur de j dans le chemin allant de r à j alors i est également appelé le **père** de j et j le **fils** de i. Un sommet j sans fils est appelé une **feuille**. La longueur du chemin (nombre d'arcs) entre r et j est la **profondeur** de j dans l'arborescence. La plus grande profondeur dans l'arborescence est appelée la **hauteur** de l'arborescence.

Dans une arborescence, tout sommet $i \neq r$ a un unique père, r n'en ayant pas. En effet, i a un père sinon il n'y aurait pas de chemin de r à i, et si i avait deux pères j_1 et j_2, on aurait deux chemins distincts de r à i (un chemin de r à j_1 plus l'arc (j_1, r), un chemin de r à j_2 plus l'arc (j_2, r)). Si r avait un père j, on aurait un circuit de r à r, donc plusieurs chemins de r aux autres sommets (en bouclant en r).

Une arborescence sur n sommets a $n - 1$ arcs. En effet, $d^-(r) = 0$ et $d^-(j) = 1$ pour tout $r \neq j$. Donc $m = \sum d^-(j) = n - 1$.

On peut transformer un arbre en arborescence en choisissant n'importe quel sommet comme racine. Si l'on choisit le sommet i comme racine, il suffit ensuite d'orienter les arêtes "en s'éloignant de i". On parle d'**arbre enraciné**. Chaque arbre correspond donc à n arborescences, suivant le choix du sommet racine.

1.4.4 Graphe biparti

Un graphe **biparti** est un graphe pour lequel les sommets peuvent être partitionnés en deux sous-ensembles X_1 et X_2 tels que tout arc (arête) a une extrémité dans X_1 et une dans X_2. Un graphe non orienté est dit **biparti complet** s'il est biparti et si pour tout $(i, j) \in X_1 \times X_2$, $ij \in E$. On le note généralement K_{n_1,n_2} avec $n_1 = |X_1|$ et $n_2 = |X_2|$.

1.5 Représentation des graphes

1.5.1 Matrices d'adjacence et d'incidence

Soit $G = (X, U)$ un graphe orienté, la **matrice d'adjacence** $A = (a_{ij})$ $(n \times n)$ à coefficient 0 ou 1 est définie comme suit :

$$a_{ij} = 1 \iff (i, j) \in U\,;$$
$$a_{ij} = 0 \text{ sinon.}$$

Ainsi, le nombre de 1 dans la ligne i vaut $d^+(i)$, et le nombre de 1 dans la colonne i vaut $d^-(i)$.
De même, pour un graphe $G = (X, E)$ non orienté, a_{ij} vaut 1 si le graphe contient l'arête ij (et 0 sinon). Ainsi, la matrice d'adjacence d'un graphe non orienté est symétrique. Le nombre de 1 dans la ligne i égale $d(i)$, de même que le nombre de 1 dans la colonne i.

Soit $G = (X, U)$ un graphe orienté sans boucle, la **matrice d'incidence sommets-arcs** est une matrice $A = (a_{ij})$ $(n \times m)$ à coefficients entiers 0, +1, -1 telle que chaque colonne correspond à un arc et chaque ligne à un sommet. Si $u = (i, j)$ est un arc du graphe alors la colonne u a tous ses termes nuls sauf :

$$a_{iu} = +1\,;$$
$$a_{ju} = -1.$$

Le nombre de +1 (resp. de -1) dans la ligne i égale $d^+(i)$ (resp. $d^-(i)$).

Soit $G = (X, E)$ un graphe non orienté sans boucle, la **matrice d'incidence sommets-arêtes** est une matrice $A = (a_{ij})$ $(n \times m)$ à coefficients entiers 0 ou 1 telle que chaque

colonne correspond à une arête et chaque ligne à un sommet. Si $e = ij$ est une arête du graphe alors la colonne e a tous ses termes nuls sauf :

$$a_{iu} = 1\,;$$
$$a_{ju} = 1.$$

2 Parcours de graphes et problèmes de connexité

De nombreux problèmes fondamentaux en théorie des graphes concernent la connexité. Par exemple :

- Un sommet j est-il accessible par un chemin à partir d'un autre sommet i ?
- Quel est l'ensemble de tous les sommets accessibles par un chemin à partir d'un sommet i ?
- Le graphe est-il connexe, c'est-à-dire tous les sommets du graphe sont-ils accessibles par une chaîne à partir d'un sommet donné i ?
- Le graphe est-il fortement connexe, c'est-à-dire existe-il un chemin joignant toute paire ordonnée de sommet (i, j) dans le graphe ?

Pour répondre à toutes ces questions, Tarjan, en 1972, a proposé un ensemble d'algorithmes efficaces (de complexité linéaire en fonction du nombre d'arcs ou d'arêtes du graphe) qui sont fondés sur une méthode d'exploration systématique des sommets d'un graphe connue sous le nom de **parcours en profondeur**.

2.1 Parcours en profondeur d'un graphe orienté (Depth First Search)

Soit $G = (X, U)$ un graphe orienté. L'idée du parcours DFS est la suivante :

On explore G à partir d'un sommet i_0 quelconque. Le but est de parcourir un à un tous les sommets du graphe accessibles à partir de i_0 (*i.e.* les sommets j pour lesquels il existe un chemin de i_0 à j). Les arcs de G utilisés pour faire le parcours forment une arborescence de racine i_0 (sur les sommets accessibles depuis i_0).
Au cours de l'exploration, chaque sommet peut se trouver dans l'un des deux états suivants :

- *non exploré* (on n'est pas encore passé par ce sommet)
- *exploré* (on est déjà passé par ce sommet).

Un sommet i exploré peut être dans l'un des deux états suivants :

- *fermé* ; c'est le cas quand tous les successeurs j de i ont été explorés,
- *ouvert* ; c'est le cas lorsque certains successeurs j de i n'ont pas encore été explorés.

A l'itération courante, on part du sommet exploré i (exploré à partir d'un sommet s).

- Si l'on n'a pas exploré tous les successeurs de i (i.e. i est *ouvert*), on explore un successeur non encore exploré j de i ; j passe alors à l'état d'exploré et on commence une nouvelle itération à partir de j.
- Si l'on a exploré tous les successeurs de i (i.e. i est *fermé*), on en a fini avec i et il faut remonter sur le sommet s pour commencer une nouvelle itération à partir de s.

L'exploration à partir de i_0 se termine lorsqu'on est remonté en ce sommet de départ i_0 et que le sommet i_0 est fermé. On a donc pu atteindre tous les sommets pouvant être atteints par un chemin à partir de i_0 dans le graphe. En effet, le parcours a été fait en utilisant des arcs du graphe donc les sommets atteints sont bien accessibles par un

chemin à partir de i_0. Réciproquement, s'il existe un chemin de i_0 à j, si j n'a pas été exploré, soit j_1 le premier sommet du chemin non exploré. Alors (j_0, j_1) est un arc du graphe, et j_0 a été exploré, donc d'après l'algorithme j_1 aussi (boucle dans *Explore*).
Voici une présentation en pseudo-langage de cette exploration appelée DFSi_0 ; le graphe G est représenté par ses listes de successeurs et on utilise un tableau *etat* de dimension n dont chaque élément prend la valeur exploré ou inexploré :

DFSi_0(graphe G, sommet i_0)
    Pour $i = 1$ à n **faire**
        etat$[i] \leftarrow$ inexploré ;
    Fin Pour
    Explore(G, i_0)
Fin DFSi_0

Algorithm 1 Procedure Explore (graphe G, sommet i)

1. état$[i] \leftarrow$ exploré ;
2. **Pour** tout j dans $N^+(i)$ **faire**
   2.1 **Si** état$[j]$ =inexploré **alors**
      2.1.1 *Explore*(G, j) ;
      Fin Si
   Fin Pour

Quand un sommet passe à l'état exploré, il est ouvert. A la fin de la boucle d'exploration de ses successeurs, il devient fermé (on a fini l'exploration du graphe à partir de ce sommet).
DFS$i_0(G, i_0)$ permet d'explorer tous les sommets de G s'il est possible d'atteindre tous les sommets du graphe à partir de i_0. Si ce n'est pas le cas, il reste des sommets non explorés. Pour continuer l'exploration et couvrir tous les sommets de G, on choisit un autre sommet i parmi les sommets non encore explorés, et on applique DFS au graphe composé des sommets non explorés à partir de ce sommet i, et ainsi de suite tant qu'il reste des sommets non explorés. L'algorithme prend fin lorsque tous les sommets ont été explorés.

La complexité de cet algorithme est en $O(m + n)$. Pour un sommet i, la procédure *Explore* contient la boucle "pour tout j dans $N^+(i)$" qui consiste à parcourir la liste des successeurs de i afin de savoir si on peut explorer un sommet non encore exploré. Comme chaque sommet est exploré au plus une fois, la procédure *Explore* appliquée à chaque sommet engendre $O(m)$ opérations au total. Dans DFS on a $O(n)$ opérations ce qui nous donne $O(m+n)$. Or, si G connexe, $m \geq n-1$ et le terme $O(n)$ est absorbé.

Lors de l'application de l'algorithme, on obtient une ou plusieurs arborescence(s) décrivant l'exploration de G par DFS, mais également une caractérisation des arcs du graphe. On distingue 4 types d'arc :

- **Arc d'arbre** : arc ayant permis de visiter de nouveaux sommets lors de l'exploration par DFS.
- **Arc avant** : (i, j) est un arc avant si et seulement si (i, j) n'est pas un arc d'arbre et s'il existe un chemin allant de i à j dans une arborescence (en d'autres termes i est un ancêtre de j dans une arborescence décrivant l'exploration par DFS).
- **Arc arrière** : (i, j) est un arc arrière si et seulement si (i, j) n'est pas un arc d'arbre et s'il existe un chemin allant de j à i dans une arborescence décrivant l'exploration par DFS (en d'autres termes i est un descendant de j dans une arborescence décrivant l'exploration par DFS).
- **Arc croisé** : (i, j) est un arc croisé si et seulement si il n'existe pas de chemin allant de i à j et de j vers i dans une arborescence décrivant l'exploration par DFS (en d'autres termes il n'existe aucun lien de parenté entre ces deux sommets).

2.2 Parcours en profondeur d'un graphe non orienté (DFSno) et détermination des composantes connexes

2.2.1 DFS pour les graphes non orientés

Il faut changer la procédure DFS en explorant, non pas les successeurs des sommets, mais ses voisins.
L'exploration à partir d'un sommet i_0 permet d'atteindre tous les sommets j du graphe tel qu'il existe une chaîne entre i_0 et j. Le graphe induit par les arêtes ayant permis de visiter de nouveaux sommets est un arbre. Notons cependant que si l'on tient compte du sens de parcours des arêtes (lors de l'exploration), l'exploration fournit alors, comme dans le cas orienté, une arborescence de racine i_0.

2.2.2 Détermination des composantes connexes

Pour déterminer les composantes connexes d'un graphe $G = (X, E)$ non orienté, il suffit d'appliquer l'algorithme de parcours du graphe en profondeur. L'exploration à partir d'un sommet i_0 donne tous les sommets j pour lesquels il existe une chaîne entre i_0 et j, c'est donc la composante connexe contenant i_0. Ainsi, chaque arbre de la forêt obtenue par DFSno couvre des sommets appartenant à une même composante connexe et le nombre d'arbres de la forêt est donc le nombre de connexité de G.
Pour déterminer les composantes connexes d'un graphe $G = (X, U)$ orienté, il faut transformer G en un graphe G' non orienté et appliquer DFSno(G'). Chaque arbre de la forêt obtenue couvre des sommets appartenant à une même composante connexe ; le nombre d'arbres de la forêt est donc le nombre de connexité de G.

2.3 Détermination des composantes fortement connexes

Pour déterminer les CFC d'un graphe, une première possibilité est de faire n parcours DFSi_0 à partir de chacun des sommets du graphe. Ensuite, pour savoir si i et j sont dans la même CFC, on regarde si j est dans l'arborescence de DF$i_0(i)$ et si i est dans l'arborescence de DF$i_0(j)$. La complexité de l'algorithme est en $O(mn + n^2) = O(mn)$. Nous allons maintenant voir comment déterminer les CFC en faisant non pas n parcours mais 2 seulement.

2.3.1 Numérotation préfixe et suffixe

Lors d'un parcours DFS, la **numérotation préfixe** indique l'ordre dans lequel les sommets ont été ouverts (explorés), la **numérotation suffixe** indique l'ordre dans lequel les sommets ont été fermés.
On peut modifier simplement l'algorithme DFS pour obtenir ces numérotations :

Algorithm 2 ALGORITHME DFS

Données : Un graphe orienté $G = (X, U)$.
Sortie : Un parcours DFS de G.

1. $k \leftarrow 1$;
2. $f \leftarrow 1$;
3. **Pour** $i = 1$ à n **faire**
   3.1 état$[i] \leftarrow$ inexploré ;
   Fin Pour
4. **Pour** $i = 1$ à n **faire**
   4.1 **Si** état$[i] =$ inexploré **alors**
   4.1.1 *Explore*(G, i, k, f) ;
   Fin Si
   Fin Pour

Ainsi, dans les arborescences créées par un parcours DFS :
- j est un descendant d'un sommet i si et seulement si $p(j) > p(i)$ et $s(j) < s(i)$;
- j est un ancêtre de i si et seulement si $p(j) < p(i)$ et $s(j) > s(i)$ (c'est la même chose que le point précédent !)
- si j et i ne sont pas descendants l'un de l'autre, alors soit $p(j) < p(i)$ et $s(j) < s(i)$ (j a été exploré avant), soit $p(i) < p(j)$ et $s(i) < s(j)$ (i a été exploré avant).

Algorithm 3 PROCEDURE *Explore*(GRAPHE G, SOMMET i, PRÉFIXE k, SUFFIXE f)

1. état$[i] \leftarrow$ exploré ;
3. p$[i] \leftarrow k$;
3. **Pour** tout j dans $N^+(i)$ **faire**
   3.1 **Si** état$[j]$ =inexploré **alors**
      3.1.1 *Explore*(G, j, k, f) ;
      Fin Si
   Fin Pour
4. s$[i] \leftarrow f$;
5. $f \leftarrow f + 1$.

2.3.2 Algorithme de Kosaraju (1978)

L'algorithme de Kosaraju-Sharir utilise la numérotation suffixe pour déterminer les composantes fortement connexes d'un graphe $G = (X, U)$.

Algorithm 4 ALGORITHME DE KOSARAJU-SHARIR

Données : Un graphe orienté $G = (V, U)$.
Sortie : Un ensemble d'arborescences dont les sommets sont une partition de V.

1. Faire un parcours DFS de G, et numéroter les sommets selon l'ordre suffixe ;
2. Créer le graphe inverse G^{-1} de G ;
3. Faire un parcours DFS de G^{-1} où le sommet choisi pour chaque racine d'arborescence est le sommet (non exploré) de plus grand numéro suffixe ;
4. **Retourner** les arborescences obtenues lors de ce dernier parcours.

Théorème 2.1 *Chaque arborescence construite par DFS(G^{-1}) couvre des sommets constituant une composante fortement connexe de G.*

Preuve. On va procéder de la manière suivante.

1. D'abord on montre que si x et y sont dans la même composante fortement connexe, alors ils sont dans la même arborescence renvoyée par l'algorithme.
2. Réciproquement, si x et y sont dans la même arborescence, soit r la racine de cette arborescence. On montre que x est dans la même composante fortement connexe que r.
3. Finalement, on va en déduire la validité de l'Algorithme de Kosaraju-Sharir.

Considérons donc deux sommets x et y de G.

1. Si x et y appartiennent à la même composante fortement connexe, il existe un chemin allant de x à y et un autre allant de y à x dans G. En conséquence, il existe dans G^{-1} un chemin allant de y à x et un autre allant de x à y. Ainsi, si on effectue un parcours DFS dans G^{-1} à partir d'un sommet r et que l'on atteint x (ou y) alors on atteindra nécessairement y (ou x) dans l'arborescence.
2. Il existe nécessairement un chemin allant de r à x dans G^{-1} et donc un chemin de x à r dans G. Nous allons montrer par récurrence sur la longueur de ce chemin que x est un descendant de r dans le parcours DFS de G (et donc qu'il existe un chemin de r à x dans G). Si $x = r$, le résultat est évident. Supposons que ce soit vrai si la longueur du chemin est au plus $k \geq 0$. Supposons que le chemin soit de longueur $k + 1$, et soit z le successeur de x dans ce chemin. Par hypothèse de récurrence, z est un descendant de r dans le DFS de G. Si x n'en est pas un, puisque d'après le principe de l'algorithme suffixe(r)>suffixe(x), cela signifie que x a été visité avant r, c'est-à-dire prefixe(x)<prefixe(r). Mais alors, étant donnée que l'arc (x, z) est dans G, soit z est visité avant x, ce qui est impossible car z est visité après r, soit z est un descendant de x. Dans ce dernier cas, puisque x n'est pas un descendant de r, r est un descendant de x, absurde car suffixe(r)>suffixe(x).
3. Un raisonnement analogue permet de montrer que r et y sont aussi dans la même composante fortement connexe. Donc x et y appartiennent à la même composante fortement connexe.

□

2.4 Tri topologique dans un graphe orienté sans circuit

Les graphes sans circuit sont fréquents dans les applications. Par exemple, dans un problème d'ordonnancement, le graphe potentiels-tâches $G = (X, U)$, défini de la façon suivante : les sommets représentent les tâches à exécuter et (i, j) est un arc si et seulement si l'éxecution de la tâche i doit précéder celle de la tâche j (la longueur de l'arc (i, j) étant la durée de la tâche i), doit par construction être un graphe sans circuit.

Dans un graphe sans circuit, on peut définir les notions d'**ordre topologique** et de **fonction rang**.

Etant donné un graphe $G = (X, U)$ d'ordre n, on appelle **ordre topologique** une numérotation v des sommets de 1 à n telle que $\forall (i, j) \in U : v(i) < v(j)$.

En d'autres termes, si l'on parcourt la liste des sommets dans l'ordre défini par une telle numérotation, un sommet j ne peut être rencontré que si l'on a, au préalable, rencontré tous ses prédécesseurs.

Remarque : il n'y a pas unicité (plusieurs ordres topologiques peuvent être définis sur un même graphe).

Théorème 2.2 *Soit $G = (X, U)$ un graphe orienté. Alors les 4 propriétés suivantes sont équivalentes :*

1. *G admet un ordre topologique ;*
2. *G est sans circuit ;*
3. *Il n'y a pas d'arc arrière dans un parcours DFS de G (quel que soit le parcours) ;*
4. *L'ordre suffixe inverse d'un parcours DFS de G est un ordre topologique.*

Preuve. (1)⇒ (2). Si on a un ordre topologique v et un circuit $(i_1, i_2, \cdots, i_k, i_1)$, alors $v(i_1) < v(i_2) < \cdots < v(i_k) < v(i_1)$, contradiction.
(2)⇒ (3). S'il y a un arc arrière (i, j), alors i est un descendant de j par définition, donc il existe un chemin de j à i, donc un circuit an ajoutant l'arc (i, j).
(3)⇒ (4). Le tableau ci-dessous présente pour un type d'arc (i, j) les relations entre les numérotations suffixes et préfixes des sommets i et j dans un DFS du graphe.

avant	préfixe(i) < préfixe(j)	suffixe(i) > suffixe(j)
arbre	préfixe(i) < préfixe(j)	suffixe(i) > suffixe(j)
croisé	préfixe(i) > préfixe(j)	suffixe(i) > suffixe(j)
arrière	préfixe(i) > préfixe(j)	suffixe(i) < suffixe(j)

S'il n'y a pas d'arc arrière, alors pour tout arc (i, j), on a suffixe(i)>suffixe(j) donc l'ordre suffixe inverse est un ordre topologique.
(4)⇒ (1). Evident. □

Si l'on note R l'ensemble des sommets d'un graphe sans circuit $G = (X, U)$ de demi-degré intérieur nul, on appelle **fonction rang associée à** G l'application *rang* qui a tout sommet j associe un nombre entier défini de la façon suivante :

— $rang(j) = 0 \ \forall j \in R$
— $rang(j)$ = nombre d'arcs dans un chemin de cardinal maximal joignant un sommet quelconque de l'ensemble R et le sommet j.

La fonction rang est unique et elle définit une partition de l'ensemble des sommets en niveaux, chaque niveau k étant formé par le sous-ensemble de sommets :

$$X_k = \{i \in X : rang(i) = k\}$$

2.5 Fermeture transitive d'un graphe

Considérons un graphe orienté connexe $G = (X, U)$. La **fermeture transitive** d'un graphe G est un graphe $F = (X, U_F)$ défini sur le même ensemble de sommets X et dont l'ensemble des arcs est défini par :

$$U_F = \{(i, j) : i \neq j \text{ et il existe un chemin de } i \text{ à } j\}$$

Le problème de la fermeture transitive d'un graphe a été étudié par de très nombreux auteurs. Une des premières approches suggérées pour le résoudre est la technique dite des multiplications matricielles, consistant à élever à la puissance $n-1$ la matrice d'adjacence du graphe. Cependant, une multiplication de deux matrices carrées de taille n se fait en $O(n^3)$. [1] Il existe une autre famille de méthodes, beaucoup plus efficaces, où l'on se ramène à la fermeture transitive d'un graphe sans circuit en décomposant le problème en trois étapes : dans une première étape, on détermine les composantes fortement connexe de G ; dans une deuxième étape, on détermine la fermeture transitive du graphe réduit G_r induit par les composantes fortement connexes, lequel est sans circuit ; et enfin dans une troisième étape, on déduit la fermeture transitive de G de la fermeture transitive de G_r.

L'algorithme de Kosaraju-Sharir nous permet de mener à bien l'étape 1 de l'algorithme. Les étapes 2 et 3 peuvent se dérouler comme suit :

Etape 2. Recherche de la fermeture transitive du graphe $G_r = (X_r, U_r)$ (sans circuit) d'ordre k.

Pour cela, il faut en premier lieu définir un ordre topologique sur les sommets de G_r. Les sommets sont alors numérotés de 1 à k selon cet ordre topologique. Pour tout sommet i de X_r, L^i désigne la liste des sommets pouvant être atteints par un chemin partant de i (liste des descendants). On initialise $L^k = \emptyset$, puis, pour chaque sommet s allant de k à 1 (ordre topologique inverse), on établit sa liste des descendants de la façon suivante :

$$L^s \leftarrow N^+(s) \bigcup_{j \in N^+(s)} L^j$$

Comme les sommets sont examinés dans l'ordre topologique inverse, lorsque le sommet s est examiné, les listes définitives L^j de tous ses successeurs directs j ont déjà été constituées.
La fermeture transitive $F_r = (X_r, U_{F_r})$ de G_r est constituée des arcs suivants : $\forall i \in X_r \; \forall j \in L^i, (i,j) \in U_{F_r}$.

Etape 3. Déduire la fermeture transitive de G de la fermeture transitive de G_r.

- (a) Pour chaque composante fortement connexe X_i de cardinal n_i, établir la liste des $n_i(n_i - 1)$ arcs du sous-graphe plein sans boucle défini sur X_i.
- (b) Si k et l sont deux sommets de G_r (correspondant aux composantes fortement connexes X_k et X_l de G) tels que l'arc (k,l) appartienne à la fermeture transitive de G_r, alors établir la liste des $n_k \times n_l$ arcs de la forme (i,j) avec $i \in X_k$ et $j \in X_l$.

1. Il existe en réalité des algorithmes plus rapides, mais de complexité toujours supérieure à la méthode basée sur les parcours de graphe.

3 Cheminement dans les graphes

Les problèmes de cheminement dans les graphes (en particulier la recherche d'un plus court chemin) comptent parmi les plus classiques de la théorie des graphes et les plus importants dans leurs applications. Le problème du **plus court chemin** (pcch) peut être posé de la façon suivante : étant donné un graphe $G = (X, U)$, on associe à chaque arc $u = (i, j)$ un nombre réel, noté $\ell(u)$ ou ℓ_{ij}, appelé la longueur de l'arc. Le problème du pcch entre deux sommets i_0 et j_0 du graphe consiste à déterminer, parmi tous les chemins allant de i_0 à j_0 celui, noté μ^* dont la longueur totale :

$$\ell(\mu^*) = \sum_{u \in \mu^*} \ell(u)$$

soit minimale.

CNS d'existence
Soit i et j deux sommets tels qu'il existe un chemin de i à j dans G. Le problème du pcch de i à j a une solution si et seulement s'il n'existe pas dans le graphe de circuit de longueur strictement négative (tel qu'il existe un chemin de i à j contenant un des sommets du circuit).

En effet, s'il existe un circuit de longueur négative avec un sommet appartenant à un chemin de i à j, alors en empruntant le circuit on fait baisser la longueur du chemin. En passant autant de fois que nécessaire, on peut donc trouver un chemin de longueur aussi petite que l'on souhaite.
Supposons qu'il n'existe pas de circuit de longueur strictement négative. La longueur d'un chemin de i à j contenant un circuit peut être réduite en supprimant certains arcs de ce circuit. Ainsi, on peut se limiter aux chemins élémentaires de i à j. Il en existe un nombre fini, donc un plus court. En particulier, il existe donc un plus court chemin qui est élémentaire.

Lorsqu'on cherche un pcch entre deux sommets i_0 et j_0, on doit déterminer d'autres pcch entre i_0 et d'autres sommets du graphe. Aussi, les algorithmes existant se divisent en deux catégories : ceux dans lesquels on recherche un pcch d'un sommet spécifié i_0 à tous les autres sommets du graphe ; ceux qui procèdent directement à la recherche de tous les pcch entre i et j pour tous les couples (i, j) du graphe.

Il existe un grand nombre d'algorithmes permettant de déterminer un pcch d'un sommet particulier i_0 à tous les autres sommets du graphe. Les plus efficaces d'entre eux réalisent un marquage des sommets c'est-à-dire qu'à chaque sommet i est associé une marque $\lambda(i)$ représentant à la fin de l'algorithme la longueur d'un pcch allant de i_0 à i.

3.1 Plus courts chemins d'origine fixée dans un graphe sans circuit avec longueurs quelconques : algorithme de Bellman

Soit $G = (X, U)$ un graphe sans circuit dont les arcs sont munis de longueurs réelles quelconques. On cherche les pcch allant de i_0 à tous les autres sommets du graphe.

Lorsque le graphe ne présente pas de circuit, on peut déterminer une numérotation des sommets allant de 1 à n qui constitue un ordre topologique (le sommet de départ i_0 ayant le numéro 1 - il n'existe pas de chemin de i_0 à j si j est avant i_0 dans l'ordre topologique) et marquer les sommets dans cet ordre.

Algorithm 5 ALGORITHME DE BELLMAN

Données : Un graphe orienté $G = (V, U)$ sans circuit et une pondération réelle des arcs.
Sortie : Les plus courts chemins de i_0 à tous les autres sommets.

1. $\lambda(1) \leftarrow 0$;
2. $p(1) \leftarrow 1$;
3. **Pour** $j \leftarrow 2$ à n **faire**
   3.1 $\lambda(j) = \min_{i \in N^-(j)} \lambda(i) + \ell_{ij}$;
   3.2 $p(j) = \text{argmin}_{i \in N^-(j)} \lambda(i) + \ell_{ij}$;
   Fin Pour

$p(j)$ désigne le dernier sommet rencontré sur un plus court chemin de 1 à j avant d'arriver en j.

La validité de l'algorithme est évidente. Notons $\lambda^*(i)$ la longueur d'un plus court chemin de 1 à i. Supposons que pour les sommets j de 1 à $i-1$, $\lambda(j) = \lambda^*(j)$. Alors $\lambda(i) = \lambda(p(i)) + \ell_{p(i)i}$. Or $p(i) < i$, donc $\lambda(p(i)) = \lambda^*(p(i))$ est la longueur d'un chemin de 1 à $p(i)$, et ainsi $\lambda(i)$ est la longueur d'un chemin de 1 à i : $\lambda^*(i) \leq \lambda(i)$.
De plus, tout chemin μ de 1 à i passe uniquement par des sommets $\leq i-1$ (on rappelle que G est sans circuit et que $1, 2, \ldots, n$ est un ordre topologique. Ainsi, $l(\mu) \geq \lambda(j) + \ell_{ji} \geq \lambda(i)$, pour un certain $j \in N^-(i)$. Ainsi, $\lambda^*(i) \geq \lambda(i)$.

Complexité : l'ordre suffixe permet d'avoir un ordre topologique en $O(m+n)$. L'algorithme de Bellman lui-même est en $O(m+n)$: en effet, une fois le tri topologique effectué, pour chaque sommet la complexité est $O(d^-(i)+1)$ (+1 car $d^-(i) = 0$ est possible).

3.2 Plus courts chemins d'origine fixée dans un graphe avec longueurs non négatives : algorithme de Dijkstra

Soit $G = (X, U)$ un graphe dont les arcs sont munis de longueur réelles positives ou nulles. On cherche les pcch de i_0 à tous les autres sommets du graphe.

L'algorithme de Dijkstra procède en $n - 1$ itérations. A l'initialisation, $\lambda(i_0) \leftarrow 0$ et $\lambda(i) \leftarrow \infty$ pour tout $i \neq i_0$. A une itération quelconque de l'algorithme, l'ensemble des sommets est partagé en deux sous-ensembles S et $X \setminus S$. Le sous-ensemble S contient l'ensemble des sommets définitivement marqués, c'est-à-dire les sommets i pour lesquels la marque $\lambda(i)$ représente effectivement la longueur d'un pcch allant de i_0 à i (à l'initialisation $S \leftarrow \{i_0\}$), i.e. $\lambda(i) = \lambda^*(i)$. $X \setminus S$ contient les sommets i ayant une marque provisoire vérifiant :

$$\lambda(i) = \begin{cases} \min_{k \in S \cap N^-(i)} \{\lambda(k) + \ell_{ki}\} \text{ si } S \cap N^-(i) \neq \emptyset \\ \infty \text{ si } S \cap N^-(i) = \emptyset \end{cases}$$

En d'autres termes, $\lambda(i)$ est la longueur minimale pour aller de i_0 à i en passant uniquement par des sommets de S.
L'algorithme est basé sur le lemme suivant.

Lemme 3.1 *Si S et λ vérifient les conditions précédentes, et si i est un sommet de $X \setminus S$ de marque provisoire $\lambda(i)$ minimale, i.e. $\lambda(i) = \min_{j \in X \setminus S} \lambda(j)$, alors $\lambda(i)$ est la longueur d'un pcch allant de i_0 à i.*

Preuve. Montrons d'abord que $\lambda(i)$ est supérieur ou égal à la valeur d'un pcch de i_0 à i. C'est évident car $\lambda(i) = \lambda(k) + \ell_{ki}$ pour un certain $k \in S$ (ou alors $\lambda(i) = \infty$). Puisque $\lambda(k)$ est la valeur d'un chemin de i_0 à k, $\lambda(i)$ est la valeur d'un chemin de i_0 à i.
Montrons maintenant que $\lambda(i)$ est inférieur ou égal à la longueur de tout chemin de i_0 à i. Soit un chemin μ de i_0 à i. Soit j le premier sommet de μ n'appartenant pas à S, et k le prédécesseur de j dans μ. Par hypothèse, la portion de μ de i_0 à k est de longueur au moins $\lambda(k)$. Les longueurs des arcs étant positives ou nulles, nous obtenons $\ell(\mu) \geq \lambda(k) + \ell_{kj}$. Or, par hypothèse $\lambda(k) + \ell_{kj} \geq \lambda(j) \geq \lambda(i)$. Ainsi, $\ell(\mu) \geq \lambda(i)$. □

Ainsi à chaque itération, on sélectionne le sommet i de plus petite marque provisoire, on l'inclut dans l'ensemble S des sommets définitivement marqués et on met à jour les marques de ses successeurs non définitivement marqués de la façon suivante :

$$\forall j \in N^+(i) \setminus S,\ \lambda(j) = \min\{\lambda(j); \lambda(i) + \ell_{ij}\}$$

Lorsque tous les sommets du graphe sont dans l'ensemble S, les marques représentent les longueurs des pcch allant de i_0 à tous les autres sommets du graphe.

Comme à chaque itération, on attribue une marque définitive à un nouveau sommet, on détermine $n-1$ pcch entre i_0 et les autres sommets du graphe en au plus $n-1$ itérations. Dans l'algorithme, $p(i)$ correspond au sommet j tel que $\lambda(i) = \lambda(j) + l_{ji}$. Cela permet de trouver un chemin optimal (et non pas seulement sa valeur).

Algorithm 6 ALGORITHME DE DIJKSTRA

Données : Un graphe orienté $G = (V, U)$, un sommet i_0 et une pondération réelle positive des arcs.
Sortie : Les plus courts chemins de i_0 à tous les autres sommets.

1. $\lambda(i_0) \leftarrow 0$;
2. $\lambda(i) \leftarrow +\infty \; \forall i \in X \setminus \{i_0\}$;
3. $p(i) \leftarrow i \; \forall i \in X$;
4. $S \leftarrow \{i_0\}$;
5. $i \leftarrow i_0$;
6. **Tant que** $S \neq X$ **faire**
   6.1 **Pour** tout $j \in N^+(i) \setminus S$ **faire**
      6.1.1 **Si** $\lambda(j) > \lambda(i) + \ell_{ij}$ **alors**
         6.1.1.1 $\lambda(j) \leftarrow \lambda(i) + \ell_{ij}$;
         6.1.1.2 $p(j) \leftarrow i$;
         Fin Si
      Fin Pour
7. Sélectionner $i \in X \setminus S$ tel que $\lambda(i) = \min_{j \in X \setminus S} \lambda(j)$;
8. $S \leftarrow S \cup \{i\}$;

   Fin Tant que

L'exactitude de l'algorithme provient du lemme 2. En effet, la mise à jour des $\lambda(j)$ permet d'assurer qu'à tout instant, pour $j \in X \setminus S$, $\lambda(j) = \min_{k \in S \cap N^-(j)} \{\lambda(k) + \ell_{kj}\}$. On sélectionne alors à chaque étape un sommet i tel que $\lambda(i)$ est la valeur du pcch de i_0 à i.

Complexité : A chaque itération, on sélectionne le sommet i de plus petite marque en $O(n)$ opérations dans le pire cas, et on met à jour les marques des successeurs de i en $O(d^+(i)+1)$ opérations. En tout il y a n itérations pour marquer tous les sommets du graphe. La complexité totale est donc en $O(n^2) + O(n+m) = O(n^2)$.

3.3 Plus courts chemins d'origine fixée dans un graphe avec longueurs quelconques : algorithme de Ford

Soit $G = (X, U)$ un graphe dont les arcs sont munis de longueur réelles quelconques. On cherche les pcch de i_0 à tous les autres sommets du graphe.

Remarque préliminaire : Soit un graphe G présentant sur certains arcs une valuation négative et, envisageons de rendre les valuations toutes positives ou nulles en ajoutant à chaque valeur la valeur absolue maximale des valeurs négatives.

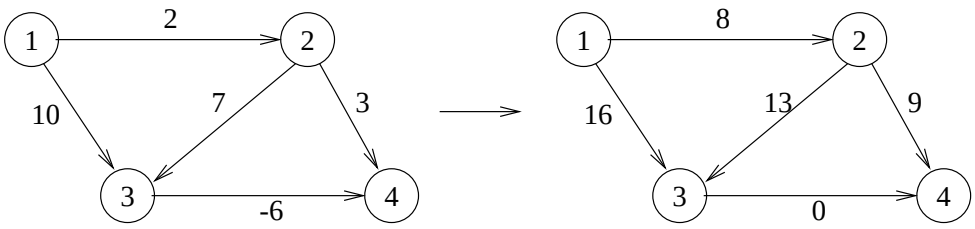

FIGURE 1 –

Ceci change la nature du problème puisque, comme on le voit dans l'exemple ci-dessus, le plus court chemin allant de 1 à 3 dans le graphe initial (de valeur 9) n'est plus optimal dans le graphe transformé.
Plus généralement, il semble délicat de se ramener au cas où les longueurs des arcs sont positives ou nulles, vu que dans le cas de longueurs négatives il peut ne pas exister de solutions.
La méthode que nous allons proposer va permettre de détecter la présence de circuits absorbants (i.e., circuits de longueur négative) quand il en existe, et de trouver un plus court chemin de i_0 à tous les autres sommets du graphe sinon. La méthode se base sur la propriété suivante, corollaire de ce que nous avons vu au début du chapitre : soit il existe un circuit absorbant, soit de i_0 à j il existe un plus court chemin empruntant au plus $n-1$ arcs (quel que soit j). Pour déterminer les pcch, on va calculer de manière récursive, pour tout $k = 0, 1, 2, \cdots, n-1$, et pour tout sommet j, la longueur $\lambda^k(j)$ d'un plus court chemin de i_0 à j empruntant au plus k arcs.
Bien sûr, au départ, $\lambda^0(i_0) = 0$ et $\lambda^0(j) = +\infty$ pour $j \neq i_0$. La relation de récurrence, à la base de l'algorithme, est la suivante :

$$\lambda^k(j) = \min(\lambda^{k-1}(j), \min_{i \in N^-(j)} (\lambda^{k-1}(i) + \ell_{ij}))$$

A chaque étape, on regarde pour chaque sommet i l'ensemble de ses prédécesseurs et on calcule $\lambda^k(i)$. En fait, à une itération k donnée, on ne va pas s'intéresser à tous les sommets du graphe mais seulement à ceux dont la marque a été modifiée au cours de l'itération précédente (il est inutile de refaire les calculs avec les sommets dont la

Algorithm 7 ALGORITHME DE FORD

Données : Un graphe orienté $G = (V, U)$, un sommet i_0 et une pondération réelle des arcs.

Sortie : Les plus courts chemins de i_0 à tous les autres sommets.

1. $k \leftarrow 0$;
2. $\lambda^0(i_0) \leftarrow 0$;
3. $\lambda^0(i) \leftarrow +\infty, \forall i \in X \setminus \{i_0\}$;
4. $p(i) \leftarrow i, \forall i \in X \setminus \{i_0\}$;
5. $M \leftarrow \{i_0\}$;
6. **Tant que** $k \leq n-1$ et $M \neq \emptyset$ **faire**
   6.1 $k \leftarrow k+1$;
   6.2 $M' \leftarrow \emptyset$;
   6.3 **Pour** tout $j \in N^+(M)$ **faire**
      6.3.1 $\lambda^k(j) \leftarrow \min\{\lambda^{k-1}(j); \lambda^{k-1}(i) + \ell_{ij},\ i \in N^-(j) \cap M\}$;
      6.3.2 **Si** $\lambda^k(j) < \lambda^{k-1}(j)$ **alors**
         6.3.2.1 $M' \leftarrow M' \cup \{j\}$;
         6.3.2.2 $p(j) \leftarrow i^*$ avec $i^* \in N^-(j) \cap M$ tel que $\lambda^k(j) = \lambda^{k-1}(i^*) + \ell_{i^*j}$;
         Fin Si
      Fin Pour
   6.4 $M \leftarrow M'$;

   Fin Tant que
7. **Si** $M \neq \emptyset$ **alors**
   7.1 $\exists$ un circuit de valeur négative ;

   Fin Si

marque n'a pas baissé). L'algorithme calcule donc à chaque itération k un ensemble de marques $\lambda^k(j)$, pour $j \in X$. On note $M = \{j \in X | \lambda^k(j) < \lambda^{k-1}(j)\}$, l'ensemble des sommets dont les marques ont été modifiées à l'iteration k, et seuls les sommets appartenant à $N^+(M)$ peuvent voir leurs marques modifiées au cours de l'itération $k+1$.

Si a un moment aucune marque n'est modifiée, l'algorithme s'arrête (les pcch ont été trouvés). Pour détecter la présence de circuit absorbant, nous itérons jusqu'à $k = n$. Si après cette itération M est non vide, alors on a un circuit absorbant (sinon nous avons les pcch).

Complexité : A chaque itération, on met à jour, dans le pire cas, les marques de tous les sommets en regardant l'ensemble des arcs : $O(m)$ opérations. Le nombre maximal d'itérations étant n, la complexité totale est en $O(nm)$.

3.4 Plus courts chemins entre toutes les paires de sommets : algorithme de Floyd

Soit la matrice $A^{(0)} = \{a_{ij}^{(0)}\}$ de taille $n \times n$ avec :

$$a_{ij}^{(0)} = \begin{cases} 0 & \text{si } i = j, \\ \ell_{ij} & \text{si } (i,j) \in U, \\ +\infty & \text{sinon.} \end{cases}$$

Pour calculer les pcch entre tout couple de sommets, nous pourrions répéter n fois l'algorithme de Ford. Cela donnerait une complexité en $O(n^2 m)$. Nous allons présenter dans ce qui suit une manière plus efficace de faire ce calcul.
L'algorithme de Floyd fonctionne comme suit. Le terme $a_{ij}^{(0)}$ de la matrice A peut s'interpréter comme le pcch de i à j ne passant par aucun autre sommet intermédiaire. Dans l'algorithme de Floyd, à la première itération, on cherche le pcch entre chaque couple (i,j) passant éventuellement par le sommet 1 ; à l'itération k (avec $k > 1$), on cherche le pcch entre chaque couple (i,j) passant par des sommets de numéro inférieur ou égal à k. Une description formelle de l'algorithme est donnée ci-dessous.

Algorithm 8 ALGORITHME DE FLOYD

Données : Un graphe orienté $G = (V,U)$ et une pondération réelle des arcs.
Sortie : Les plus courts chemins entre chaque paire de sommets.

1. **Pour** $k = 1$ à n **faire**
   1.1 **Pour** $i = 1$ à n **faire**
      1.1.1 **Pour** $j = 1$ à n **faire**
         1.1.1.1 $a_{ij}^{(k)} = \min\{a_{ij}^{(k-1)}, a_{ik}^{(k-1)} + a_{kj}^{(k-1)}\}$;
         Fin Pour
      Fin Pour
   Fin Pour

La complexité totale est en $O(n^3)$.
Remarque : on peut vérifier s'il n'existe pas de circuit absorbant. En effet, il en existe un si et seulement si (au moins) un terme de la diagonale est strictement négatif.

4 Arbre couvrant de poids minimum

4.1 Formulation du problème

Soit $G = (X, E)$ un graphe simple connexe d'ordre n et soit $w : E \to \mathbb{R}_+$ une pondération des arêtes de G. Un **arbre couvrant** $T = (X, E_T)$ est un graphe partiel de G connexe et sans cycle. En conséquence, $|E_T| = n - 1$. Le **poids d'un arbre couvrant** est la somme des poids des arêtes lui appartenant :

$$w(T) = \sum_{ij \in E_T} w(ij)$$

Le problème de l'arbre couvrant de poids minimum consiste à déterminer pour un graphe simple connexe $G = (X, E)$ dont chaque arête $ij \in E$ a un poids $w(ij)$ un arbre couvrant dont le poids est minimum.
Avant de présenter deux algorithmes permettant de résoudre ce problème de manière optimale, démontrons le lemme suivant qui nous permettre de justifier la validité des algorithmes.

Lemme 4.1 *Soient $T = (X, E_T)$ et $T' = (X, E_{T'})$ deux arbres couvrants d'un même graphe $G = (X, E)$ avec $|X| = n$. Soit $e \in E_T \setminus E_{T'}$. Alors il existe $e' \in E_{T'} \setminus E_T$ tel que $T^* = (X, (E_{T'} \setminus \{e'\}) \cup \{e\})$ est un arbre couvrant.*

Preuve. Comme $e \notin E_{T'}$, $E_{T'} \cup \{e\}$ contient nécessairement un cycle C (qui en plus est unique). Comme $e \in E_T$, il existe dans C une arête $e' \notin E_T$ sinon T contiendrait un cycle. Mais alors $T^* = (X, (E_{T'} \setminus \{e'\}) \cup \{e\})$ est un arbre couvrant car (i) $T^* = (X, (E_{T'} \setminus \{e'\}) \cup \{e\})$ possède $n-1$ arêtes et (ii) $T^* = (X, (E_{T'} \setminus \{e'\}) \cup \{e\})$ est sans cycle (car C était unique) et connexe. □

4.2 Algorithme de Kruskal

Le premier algorithme que nous allons présenter pour résoudre le problème de l'ARBRE COUVRANT MINIMAL est l'*Algorithme de Kruskal*. Il s'agit d'un algorithme glouton dont l'idée consiste à choisir à chaque étape l'arête avec le plus petit poids qui, avec les arêtes déjà choisies, ne crée pas de cycle. A chaque étape, on a donc une forêt dont les sommets sont tous les sommets du graphe G.

Algorithm 9 ALGORITHME DE KRUSKAL

Données : Un graphe simple connexe $G = (X, E)$ et une fonction poids $w : E \to \mathbb{R}$.
Sortie : Un arbre couvrant T de poids minimum.

1. Trier les arêtes telles que $w(e_1) \leq w(e_2) \leq \ldots \leq w(e_m)$;
2. Poser $T := (X, E_T = \emptyset)$;
3. **Pour** $i := 1$ **à** m **faire**
   3.1 **Si** $E_T \cup \{e_i\}$ ne contient pas de cycle **alors**
   3.1.1 Poser $E_T := E_T \cup \{e_i\}$;
   Fin Si
   Fin Pour
4. **Retourner** T.

Exemple

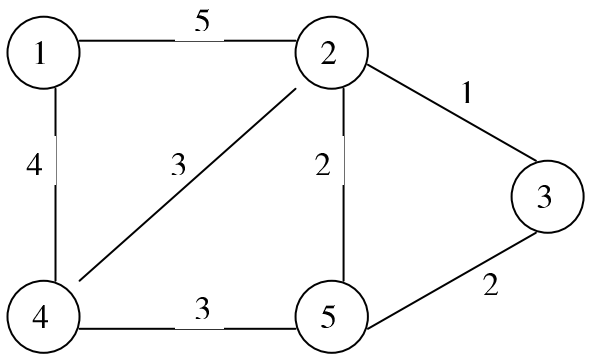

FIGURE 2 –

Théorème 4.1 *L'Algorithme de Kruskal produit un arbre couvrant de poids minimum en temps polynomial.*

Preuve. Prouvons d'abord que l'algorithme donne bien un arbre couvrant. Soit T le graphe retourné par l'algorithme. Clairement, T ne contient pas de cycle d'après la description de l'algorithme. De plus, T est un graphe connexe car sinon la première arête (selon l'ordre) dont les deux extrémités se trouvent dans des composantes connexes différentes aurait été ajoutée à T. Finalement, T est couvrant car toutes les arêtes sont considérées lors de l'éxécution de l'algorithme.

Soit $T' = (X, E_{T'})$ un arbre couvrant de poids minimum. Si $T = T'$, alors T est de poids minimum. On peut donc supposer que $T \neq T'$. Notons par $e_1, \ldots, e_{n-1}$ les arêtes ajoutées dans l'ordre par l'algorithme. Comme $T \neq T'$, il existe k tel que $e_1, \ldots, e_k \in E_{T'}$ mais $e_{k+1} \notin E_{T'}$. D'après le Lemme 4.1, il existe alors $e' \in E_{T'} \setminus E_T$ tel que $T^* = (X, (E_{T'} \setminus \{e'\}) \cup \{e_{k+1}\})$ est un arbre couvrant. Ceci implique que $w(e_{k+1}) \geq w(e')$ car sinon T^* serait un arbre couvrant avec $w(T^*) < w(T')$, une contradiction. De plus, on ne peut pas avoir $w(e_{k+1}) > w(e')$, sinon l'algorithme aurait choisi e' au lieu de e_{k+1} lors de la $(k+1)$-ème itération. Ainsi on conclut que $w(e_{k+1}) = w(e')$. Donc T^* est un arbre couvrant de poids minimum qui coincide avec T sur les $k+1$ premières arêtes ajoutées par l'algorithme. En itérant le procédé, on obtient finalement que T coincide avec un arbre couvrant de poids minimum.
Analysons maintenant la complexité de l'Algorithme de Kruskal. L'étape 1. de l'algorithme nécessite un temps $O(m \log(m))$ dans un graphe contenant m arêtes. Dans la boucle 3., l'algorithme doit tester si une arête $e = xy$ crée un cycle ou pas si on l'ajoute à E_T. Pour cela il suffit de tester si x et y appartiennent à la même composante connexe de T. On peut le faire en numérotant les composantes connexes de T et en stockant pour chaque sommet le numéro de la composante connexe à laquelle il appartient. Aussi il faut mettre à jour les numéros de certains sommets une fois qu'une arête a été ajoutée. Ainsi la boucle 3. se fait en $O(n^2)$ (on rappelle que T contient $n-1$ arêtes et donc la procédure consistant à mettre à jour les numéros de certains sommets se fait $O(n)$ fois) et l'algorithme est donc en $O(\max(m \log(m), n^2)$, i.e. en $O(n^2 \log(n))$.
Par contre, on peut faire le test ci-dessus de manière plus efficace en utilisant des structures de données adaptées (structures de données pour ensembles disjoints). Ainsi l'algorithme peut tourner avec une complexité globale de $O(m \log(m))$. □

4.3 Algorithme de Prim

Le deuxième algorithme qui résout optimalement le problème de l'ARBRE COUVRANT MINIMAL est l'*Algorithme de Prim*. Dans cet algorithme, qui est également glouton, on ajoute à chaque étape une arête de poids minimum qui, avec les arêtes déjà choisies, crée un arbre, i.e., une arête dont une extrémité est déjà incluse dans l'arbre actuel et l'autre ne l'est pas.

Exemple

Théorème 4.2 *L'*ALGORITHME DE PRIM *produit un arbre couvrant de poids minimum en temps polynomial.*

Preuve. Soit T le graphe retourné à la fin de l'algorithme. Alors T est un arbre couvrant. En effet, comme à chaque étape on ajoute une arête dont une extrémité est

Algorithm 10 ALGORITHME DE PRIM

Données : Un graphe simple connexe $G = (X, E)$ et une fonction poids $w : E \to \mathbb{R}$.
Sortie : Un arbre couvrant T de poids minimum.

1. Choisir $v \in X$. Soit $T := (X_T = \{v\}, E_T = \emptyset)$.
2. **Tant que** $X_T \neq X$ **faire**
   2.1 Choisir une arête $e = uv$ avec $u \in X_T$ et $v \in X \setminus X_T$ de poids minimum ;
   2.2 Poser $E_T := E_T \cup \{e\}$;
   2.3 Poser $X_T := X_T \cup \{v\}$;

   Fin Tant que
3. **Retourner** T.

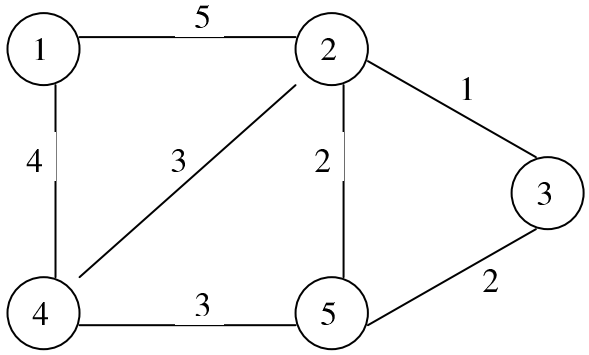

FIGURE 3 –

dans X_T et l'autre est dans $X \setminus X_T$, le graphe résultant est connexe et sans cycle, donc un arbre. De plus, $V_T = V$ à la fin de l'exécution de l'algorithme, donc T est couvrant. Soit T' un arbre couvrant de poids minimum. Si $T = T'$, alors le résultat est vrai. Donc supposons que $T' \neq T$ et soit $e = uv$ la première arête ajoutée à T qui n'est pas dans T' et ceci lors de la k-ième itération. Soit P le chemin dans T' de u vers v et soit $f \neq e$ une arête dans P dont une extrémité est dans T après la $(k-1)$-ième itération et l'autre extrémité n'est pas dans T. Si $w(f) < w(e)$, alors l'algorithme aurait choisi l'arête f lors de la k-ième étape et non pas l'arête e. Si $w(f) > w(e)$, alors $T'' = (X, (E_{T'} \setminus \{f\}) \cup \{e\})$ est un arbre couvrant (voir preuve du Lemme 4.1) de poids $w(T'') < w(T')$, une contradiction. Ainsi $w(e) = w(f)$. On pose alors $T' = (X, (E_{T'} \setminus \{f\}) \cup \{e\})$ et on cherche à nouveau la première arête ajoutée à T qui n'est pas dans T'. En répétant cet argument, on trouve finalement que $w(T) = w(T')$ et donc T est un arbre couvrant de poids minimum.
Analysons maintenant la complexité de l'Algorithme de Prim. Pour chaque sommet

$v \in X \setminus X_T$, on retient l'arête de plus petit poids dont une extrémité est v et l'autre est dans X_T. Appelons ces arêtes les *arêtes candidates* de v. L'initialisation des arêtes candidates pour chaque sommet se fait en $O(m)$. Chaque sélection de l'arête la moins chère parmi les arêtes candidates se fait en $O(n)$. La mise à jour des arêtes candidates se fait en $O(n)$. Comme on répète la boucle 2. $n-1$ fois, on obtient une complexité globale de $O(n^2)$.
Notons néanmoins que l'utilisation de structure de données adaptées permet de réduire cette complexité à $O(m \log(m))$. □

5 Flots dans les graphes

Le problème de flots dans les graphes concerne la circulation de matière sur les arcs. Parmi les nombreuses applications qui relèvent de ce problème, on trouve le transport de marchandises entre différents points, les télécommunications dans les réseaux, etc . . . .

5.1 Définitions

Un **réseau** $R = (X, U)$, avec $U = \{u_1, \ldots, u_m\}$, est un graphe orienté connexe sans boucle tel que :

- il existe une **source** notée s (il s'agit d'un sommet racine), qui représente la point d'entrée de la matière,
- il existe un **puits** (il s'agit d'un sommet antiracine) notée t, qui représente le point de sortie de la matière,
- chaque arc $u = (i, j) \in U$ est muni d'une **capacité** c_u entière positive ou nulle (et éventuellement un coût d_u entier positif ou nul).

Un **flot** ϕ dans $R = (X, U)$ est un vecteur de $\mathbb{R}^m$, dont l'élément générique ϕ_u est appelé **flux** sur l'arc $u = (i, j)$, tel que :

(i) $\forall \ell = 1, \ldots, m, \phi_{u_\ell} \geq 0$;

(ii) en tout sommet $i \in X \setminus \{s, t\}$, la loi de conservation aux noeuds est vérifiée :

$$\sum_{j \in N^+(i)} \phi_{(i,j)} = \sum_{j \in N^-(i)} \phi_{(j,i)}$$

Un **flot compatible** ϕ dans un réseau $R = (X, U)$ est un flot tel que :

$$\forall u \in U,\ 0 \leq \phi_u \leq c_u.$$

La **valeur** f_ϕ d'un flot compatible ϕ dans un réseau est égale à la somme des flux sortant de s, elle-même égale à la somme des flux entrant en t (comme nous le prouverons plus tard) :

$$f_\phi = \sum_{j \in N^+(s)} \phi_{(s,j)} = \sum_{j \in N^-(t)} \phi_{(j,t)}.$$

Le problème du **flot maximum** dans un réseau s'énonce comme suit : déterminer quelle est la valeur maximale du flot qu'il est possible de faire parvenir de la source s au puits t (étant donné que nous avons considéré une capacité maximale sur chaque arc, il ne peut existe de flot de valeur arbitrairement grande).

Le problème de flot maximum de s à t peut se formuler de la façon suivante :

$$\begin{cases} \max & f_\phi \\ s.c. & \sum_{j \in N^+(i)} \phi_{(i,j)} - \sum_{j \in N^-(i)} \phi_{(j,i)} = 0 \ \forall i \in X \setminus \{s, t\} \\ & \sum_{j \in N^+(s)} \phi_{(s,j)} = f_\phi \\ & 0 \leq \phi_u \leq c_u \ \forall u \in U \end{cases}$$

5.2 Propriétés fondamentales

Soit $R = (X, U)$ un réseau et soit $S \subset X$. Notons $\omega^+(S) = \{(i,j) |\ i \in S,\ j \notin S\}$ et $\omega^+(S) = \{(i,j) |\ i \notin S,\ j \in S\}$.

Lemme 5.1 *Considérons un réseau $R = (X, U)$. Soit $S \subset X$ ($S \neq \emptyset$, $S \neq X$) et f_u un nombre sur chaque arc u.*

$$\sum_{i \in S} \left(\sum_{j \in N^+(i)} f_{(i,j)} - \sum_{j \in N^-(i)} f_{(j,i)} \right) = \sum_{u \in \omega^+(S)} f_u - \sum_{u \in \omega^-(S)} f_u$$

Preuve. Nous allons analyser la contribution de chaque arc u à la somme (partie gauche de l'égalité) :

- Si u n'a aucune extrémité dans S, alors u n'intervient pas dans la somme ;
- Si $u = (i,j)$ avec i et j dans S : on compte f_u pour le sommet i et $-f_u$ pour le sommet j, donc au total 0 ;
- Si $u = (i,j) \in \omega^+(S)$: on compte f_u pour le sommet i ;
- Si $u = (i,j) \in \omega^-(S)$: on compte $-f_u$ pour le sommet i ;

□

Lorsque l'on a un flot dans un réseau, la quantité $\sum_{j \in N^+(i)} \phi_{(i,j)} - \sum_{j \in N^-(i)} \phi_{(j,i)}$ vaut 0 pour tout $i \notin \{s, t\}$. Ainsi, nous obtenons comme corollaire de ce lemme :

Propriété 5.1 *Soit un flot ϕ de valeur f_ϕ dans un réseau $R = (X, U)$. Alors, pour tout $S \subset X$ tel que $s \in S$ et $t \notin S$:*

$$\sum_{u \in \omega^+(S)} \phi_u - \sum_{u \in \omega^-(S)} \phi_u = \sum_{j \in N^+(s)} \phi_{(s,j)} = f_\phi$$

En particulier, si on prend $S = X \setminus \{t\}$, nous obtenons $f_\phi = \sum_{j \in N^-(t)} \phi_{(j,t)}$.

5.2.1 Flot maximum et coupe minimum

Dans un réseau $R = (X, U)$, soit S un sous-ensemble de X avec $s \in S$ et $t \notin S$. Le **cocycle engendré par** S, noté $\omega(S)$, est l'ensemble des arcs de R ayant une extrémité dans S et l'autre dans $X \setminus S$. On l'appelle **coupe séparant s de t** ou **s-t coupe**. On a également $\omega(S) = \omega^+(S) \cup \omega^-(S)$, où $\omega^+(S)$ est constitué des arcs (i,j) avec $i \in S$ et $j \notin S$ (et $\omega^-(S)$ des arcs (i,j) avec $i \notin S$ et $j \in S$.

La **capacité d'une coupe**, notée $c(S)$, est donnée par la somme des capacités des arcs "sortant" de S, c'est-à-dire ceux appartenant à $\omega^+(S)$:

$$c(S) = \sum_{u \in \omega^+(S)} c_u.$$

Théorème 5.1 *La valeur d'un flot quelconque ϕ dans un réseau $R = (X, U)$ est inférieure ou égale à la capacité de toute s-t coupe.*

Preuve. Si S est une s-t coupe, grâce à la Propriété 5.1, et en utilisant le fait que $0 \leq \phi_u \leq c_u$, nous obtenons :

$$f_\phi = \sum_{u \in \omega^+(S)} \phi_u - \sum_{u \in \omega^-(S)} \phi_u \leq \sum_{u \in \omega^+(S)} \phi_u \leq \sum_{u \in \omega^+(S)} c_u = c(S).$$

□

En conséquence, si on trouve un flot de valeur f égal à la capacité d'une coupe $\omega(S)$, ce flot est de valeur maximale et cette coupe est de capacité minimale.
Le théorème Flot Max - Coupe Min ci-dessous stipule qu'il existe nécessairement un flot dont la valeur est égale à la capacité d'une coupe.

Théorème 5.2 (Théorème de Ford-Fulkerson (max-flow min-cut)) *La valeur d'un flot maximum ϕ dans un réseau $R = (X, U)$ est égale à la plus petite des capacités des s-t coupes.*

Nous allons prouver ce théorème tout en montrant comment trouver un flot maximum et une coupe minimum dans un réseau. Nous allons présenter l'algorithme de Ford et Fulkerson.

5.2.2 Graphe d'écart et chemin augmentant

Soit ϕ un flot admissible sur R. Le **graphe d'écart** associé à ϕ sur R est le graphe $R^e(\phi) = (X, U^e(\phi))$ défini comme suit : pour tout $u = (i, j) \in U$

— Si $\phi_u < c_u$, $(i, j) \in U^e(\phi)$ avec la capacité (résiduelle) $c'_{(i,j)} = c_u - \phi_u$

— Si $\phi_u > 0$, $(j, i) \in U^e(\phi)$ avec la capacité $c'_{(j,i)} = \phi_u$.

Dans le premier cas, un arc est valué par sa capacité résiduelle, c'est-à-dire l'augmentation de flux possible. Dans le second cas, l'idée est de diminuer le flux sur l'arc (i, j) en faisant passer une quantité de flux sur l'arc (j, i).

Soit ϕ un flot admissible sur R et $R^e(\phi) = (X, U^e(\phi))$ le graphe d'écart associé.
Nous allons montrer que :

— S'il existe un chemin de s à t dans le graphe d'écart, alors nous pouvons augmenter la valeur du flot ;

— S'il n'existe pas de chemin de s à t dans le graphe d'écart, alors le flot est optimal. De plus, si l'on note S l'ensemble des sommets accessibles à partir de s dans le graphe d'écart, la capacité de la $s - t$ coupe $\omega^+(S)$ dans R est égale à la valeur du flot.

Soit μ un chemin allant de s à t dans $R^e(\phi)$ et $\delta = min_{u\in\mu} c'_u$. Ce chemin est appelé **chemin augmentant** car il est possible d'augmenter la valeur du flot sur R de δ de la façon suivante pour tout $(i,j) \in \mu$:

— si $u = (i,j) \in U$, alors $\phi_u \leftarrow \phi_u + \delta$,

— si $u = (j,i) \in U$, alors $\phi_u \leftarrow \phi_u - \delta$.

On obtient alors bien un nouveau flot réalisable. En effet, soit un sommet $i \notin \{s,t\}$. Si i n'appartient pas au chemin la conservation du flux n'a pas changé pour i. Si i appartient au chemin, si on a un arc (i,j) dans le chemin on a augmenté de δ le flot sortant de i (ou diminué de δ le flot entrant) et si on a un arc (j,i) on a augmenté de δ le flot entrant en i (ou diminué de δ le flot sortant). Au final, la loi de conservation est bien vérifiée.

S'il n'existe pas de chemin de s à t, soit S l'ensemble des sommets accessibles à partir de s dans le graphe d'écart. Soit $(i,j) \in \omega^+(S)$: (i,j) n'étant pas dans le graphe d'écart, $\phi_u = c_u$. Soit $(i,j) \in \omega^-(S)$: (j,i) n'étant pas dans le graphe d'écart, $\phi_u = 0$. Ainsi :

$$f_\phi = \sum_{u\in\omega^+(S)} \phi_u - \sum_{u\in\omega^-(S)} \phi_u = \sum_{u\in\omega^+(S)} \phi_u = \sum_{u\in\omega^+(S)} c_u = c(S)$$

Ceci démontre le théorème de Ford et Fulkerson.

5.3 Recherche d'un flot maximum : algorithme de Ford-Fulkerson

5.3.1 Algorithme générique

Soit un réseau R, un flot compatible ϕ dans R et le graphe d'écart $R^e(\phi)$. Pour déterminer un flot maximum, l'algorithme générique consiste, à chaque itération, à chercher un chemin μ allant de s à t dans $R^e(\phi)$. Si un tel chemin existe, on augmente le flot ϕ de la quantité $\delta = \min_{u\in\mu} c'_u$. Sinon l'algorithme termine et ϕ est le flot de valeur maximale.

$\phi \leftarrow 0$
Tant que $R^e(\phi)$ contient un chemin de s à t **faire**
    Identifier un chemin μ de s à t
    $\delta = \min_{u\in\mu} c'_u$
    Augmenter de δ unités le flot ϕ sur R
    Mettre à jour $R^e(\phi)$
Fin Tant que

Cet algorithme ne précise pas de quelle façon déterminer un chemin μ de s à t. Dans la section suivante, nous présentons un algorithme de marquage qui permet d'exhiber un chemin augmentant dans $R^e(\phi)$ (ou de façon équivalente une chaîne augmentante dans R).

5.3.2 Algorithme de Ford-Fulkerson

L'algorithme de Ford-Fulkerson est composé de 2 procédures : la procédure de **marquage** et la procédure d'**augmentation**. La procédure de marquage marque les sommets i qui sont accessibles depuis le sommet s dans le graphe d'écart. C'est cette procédure qui permet de déterminer s'il existe un chemin de s à t dans le graphe $R^e(\phi)$. Si un tel chemin n'existe pas, on s'arrête. Sinon, on appelle la procédure d'augmentation qui permet d'augmenter le flot actuel. Ensuite on recommence avec la procédure de marquage après avoir mis à jour le graphe d'écart.

Algorithm 11 ALGORITHME DE FORD-FULKERSON

Données : Un réseau $R = (X, U)$.
Sortie : Un flot de valeur maximum dans R.

1. $\phi \leftarrow 0$ (Partir du flot nul)
2. **Faire**
   2.1 construire $R^e(\phi)$;
   2.2 marque(i) $\leftarrow 0\ \forall i \in X$;
   2.3 pred(i) $\leftarrow i\ \forall i \in X$;
   2.4 marque(s) $\leftarrow *$ et LISTE $\leftarrow \{s\}$;
   2.5 **Tant que** LISTE $\neq \emptyset$ et marque(t) $= 0$ **faire**
      2.5.1 Sélectionner $i \in$ LISTE ;
      2.5.2 LISTE $\leftarrow$ LISTE $\setminus \{i\}$;
      2.5.3 **Pour** chaque arc $u = (i, j) \in U^e(\phi)$ **faire**
         2.5.3.1 **Si** marque(j)$= 0$ **alors**
            2.5.3.1.1 marque(j) $\leftarrow *$;
            2.5.3.1.2 pred(j) $\leftarrow i$;
            2.5.3.1.3 LISTE $\leftarrow$ LISTE $\cup \{j\}$;
            Fin Si
         Fin Pour
      Fin Tant que
   2.6 **Si** marque(t)$\neq 0$ **alors**
      2.6.1 *augmenter* ;
      Fin Si

   Tant que marque(t) $\neq 0$.

Exactitude de l'algorithme : l'algorithme consiste à faire un parcours du graphe d'écart à partir de s. Ainsi, à chaque étape soit on marque t et l'on augmente le flot, soit on ne peut marquer t et alors le résultat du paragraphe précédent nous dit que le flot est maximum. Dans ce cas, la coupe engendrée par l'ensemble des sommets marqués est une coupe de capacité minimum (égale à la valeur du flot maximum).

Algorithm 12 PROCÉDURE *augmenter*

1. Identifier la chaîne μ dans $R^e(\phi)$ à l'aide de pred(i) : $\mu = (u_0 = (s, i_0), \ldots, u_{k+1} = (i_k, t))$;
2. $\delta \leftarrow \min_{u \in \mu} c'_u$;
3. $\phi_{(s,i_0)} \leftarrow \phi_{(s,i_0)} + \delta$;
4. **Pour** $j \leftarrow 1$ à k **faire**
   4.1 **Si** (i_{j-1}, i_j) dans $U^e(\phi)$ correspond à (i_{j-1}, i_j) dans $U^e(\phi)$ **alors**
      4.1.1 $\phi_{(i_{j-1},i_j)} \leftarrow \phi_{(i_{j-1},i_j)} + \delta$;
   4.2 **Sinon**
      4.2.1 $\phi_{(i_{j-1},i_j)} \leftarrow \phi_{(i_{j-1},i_j)} - \delta$;
      Fin Si

   Fin Pour
5. $\phi_{(i_k,t)} \leftarrow \phi_{(i_k,t)} + \delta$.

Propriété 5.2 *Si toutes les capacités sont entières, le problème du flot maximum a une solution optimale entière (une solution où tous les flux sont entiers).*

Preuve. Le flot de départ est le flot nul. Comme toutes les capacités sont entières, la chaîne augmentante trouvée, si elle existe, dispose d'une capacité δ entière (car toute les capacités résiduelles sont entières). Le nouveau flot obtenu est donc à coordonnées entières. Et ainsi de suite jusqu'à la dernière itération.
Il nous faut également préciser que l'algorithme termine. Au pire cas, on augmente à chaque itération la valeur du flot d'une unité ($\delta = 1$) et, comme le flot maximum ne peut pas dépasser la capacité (finie, tous les arcs ayant une capacité finie) d'une coupe quelconque, l'algorithme termine en un nombre fini d'itérations. □

La complexité de l'algorithme est maintenant évidente à calculer. Chaque itération comporte $O(m)$ opérations élémentaires car la méthode de marquage examine chaque arc et chaque sommet au plus une fois. En conséquence, la complexité totale est $O(m)$ fois le nombre d'augmentation. Si chaque capacité est entière et bornée par U, la capacité de la coupe $\omega^+(\{s\})$ est au plus nU. Ainsi, la valeur d'un flot maximum est au plus nU. Lors de chaque procédure d'augmentation, la valeur du flot augment d'au moins 1. Ainsi, on termine après au plus nU augmentations. La complexité totale est donc en $O(nmU)$. En conséquence, si par exemple $U = 2^n$, la complexité de l'algorithme est en $O(nm2^n)$ c'est-à-dire exponentielle par rapport au nombre de sommets du réseau. Il existe cependant des règles permettant d'arriver à une complexité polynomiale.

Pour converger plus vite, on peut faire mieux que partir du flot nul. On commence par rechercher un **flot complet**.

Un arc u est dit saturé si $\phi_u = c_u$. Un flot est **complet** si dans chacun des chemins allant de s à t dans R, il existe au moins un arc saturé.

Remarque :
Flot max $\Rightarrow$ Flot complet
Flot complet $\not\Rightarrow$ Flot max

Pour trouver un flot complet, on peut utiliser l'heuristique de bas en haut ou de haut en bas.

5.3.3 Une application du problème du flot maximum : le couplage dans un graphe biparti

Un **couplage** dans un graphe $G = (X, E)$ est un ensemble d'arêtes $E' \subseteq E$ deux à deux non adjacentes. Le cardinal d'un couplage E' est le nombre d'arêtes de E'.

Un couplage E' est dit **parfait** lorsque tout sommet de X est incident à une arête de E' (autrement dit, lorsque dans le graphe partiel (X, E') tous les sommets du graphe sont de degré 1).

Un graphe est dit **biparti** si l'ensemble des sommets peut être partitionné en deux sous-ensembles X_1 et X_2 de façon à ce que chaque arête ait une extrémité dans X_1 et l'autre dans X_2.

Exemple : Le problème d'affectation.
Lorsqu'on cherche à affecter des personnes à des tâches (une personne par tâche), on cherche un couplage dans un graphe simple où les sommets représentent les individus et les tâches, et les arêtes représentent les affectations (individu, tâche) possibles. Ce graphe, dit d'affectation, est un graphe biparti. On peut rechercher un couplage de cardinalité maximale lorsqu'on cherche à couvrir le plus de tâches possibles. On peut aussi rechercher un couplage maximum de poids minimal si le graphe d'affectation est valué par des coûts d'affectation.

Pour trouver un couplage maximum dans un graphe biparti il suffit de déterminer un flot de valeur maximale dans le graphe biparti transformé comme suit : on ajoute une source reliée à tous les sommets de X_1 et un puits relié à tous les sommets de X_2, ces nouveaux arcs ayant une capacité 1 (la capacité des arcs du graphe biparti peut valoir 1 ou $+\infty$).

5.4 Recherche d'un flot maximum à coût minimal

Considérons un réseau $R = (X, U)$ où chaque arc u possède, outre sa capacité c_u, un coût d_u. Soit f^* la valeur d'un flot maximum dans R. Le coût du flux ϕ_u sur l'arc u est $\phi_u d_u$. Le coût d'un flot est la somme des coûts sur chacun des arcs. Le problème du flot maximum à coût minimal dans R peut donc s'écrire comme suit :

$$\begin{cases} \min & \sum_{u \in U} d_u \phi_u \\ s.c. & \sum_{j \in N^+(i)} \phi_{i,j} - \sum_{j \in N^-(i)} \phi_{j,i} = 0 \ \forall i \in X \setminus \{s, t\} \\ & \sum_{j \in N^+(s)} \phi_{s,j} = f^* \\ & 0 \leq \phi_u \leq c_u \ \forall u \in U \end{cases}$$

On retrouve aussi la notion de coût dans le graphe d'écart. Soit ϕ un flot admissible dans R. Le graphe d'écart associé à ϕ sur R est le graphe $R^e(\phi) = (X, U^e(\phi))$ défini comme suit : $\forall u = (i, j) \in U$

- Si $\phi_u < c_u$, $(i, j) \in U^e(\phi)$ avec la capacité (résiduelle) $c'_{(i,j)} = c_u - \phi_u$ et le coût $d'_{ij} = d_u$
- Si $\phi_u > 0$, $(j, i) \in U^e(\phi)$ avec la capacité $c'_{(j,i)} = \phi_u$ et le coût $d'_{ji} = -d_u$.

5.4.1 Condition d'optimalité

Théorème 5.3 *Un flot ϕ^* (de valeur f^*) est une solution optimale du problème du flot maximum à coût minimal si et seulement si le graphe d'écart associé $R^e(\phi^*)$ ne comporte pas de circuit de longueur (i.e., coût) strictement négative.*

S'il comporte un circuit de longueur strictement négative cela signifie qu'il est possible de passer du flot ϕ^* à un nouveau flot de coût inférieur, en augmentant/diminuant le flux d'une quantité δ sur les arcs du circuit. Nous admettrons la réciproque.

5.4.2 Un algorithme de détermination du flot maximum à coût minimal

De la condition d'optimalité présentée dans la section précédente, il est possible de définir un algorithme très simple de détermination d'un flot maximum à coût minimal comme suit :

Etablir un flot de valeur maximale
Tant que le graphe d'écart associé contient des circuits de valeur negative **faire**
    Choisir un circuit négatif C avec δ la capacité résiduelle minimale de C ;
    Augmenter/diminuer de δ unités le flot dans le circuit ;
    Mettre à jour le graphe d'écart associé.
Fin Tant que.

Pour détecter un circuit de longueur négative, on utilisera p.ex. l'algorithme de Ford ou l'algorithme de Floyd.

Références bibliographiques

A. Aho, J. Hopcroft, J. Ullman, Structures de données et algorithmes, InterEditions, 1987.
M. Minoux, G. Bartnik, Graphes Algorithmes Logiciels, Dunod, 1986.
M. Gondran, M. Minoux, Graphes et Algorithmes, Eyrolles, 1985.