Chapitre 1

Quelques rappels sur la théorie des graphes

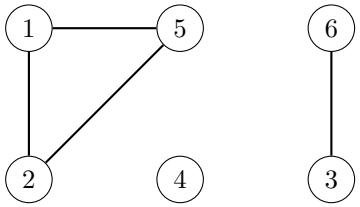
1.1 Définitions

1.1.1 Graphes non orientés

Définition 1.1 Un **graphe** non orienté G est la donnée d'un couple $G = (S, A)$ tel que :
- S est un ensemble fini de sommets,
- A est un ensemble de couples non ordonnés de sommets $\{s_i, s_j\} \in S^2$.

Une paire $\{s_i, s_j\}$ est appelée une **arête**, et est représentée graphiquement par s_i–s_j. On dit que les sommets s_i et s_j sont **adjacents**. L'ensemble des sommets adjacents au sommet $s_i \in S$ est noté $Adj(s_i) = \{s_j \in S, \{s_i, s_j\} \in A\}$.

Par exemple,



représente le graphe non orienté $G = (S, A)$ avec $S = \{1, 2, 3, 4, 5, 6\}$ et
$A = \{\{1, 2\}, \{1, 5\}, \{5, 2\}, \{3, 6\}\}$.

Définition 1.2
- Une **boucle** est une arête reliant un sommet à lui-même.
- Un graphe non-orienté est dit **simple** s'il ne comporte pas de boucle, et s'il ne comporte jamais plus d'une arête entre deux sommets. Un graphe non orienté qui n'est pas simple est un **multi-graphe**. Dans le cas d'un multi-graphe, A n'est plus un ensemble mais un multi-ensemble d'arêtes. On se restreindra généralement dans la suite aux graphes simples.

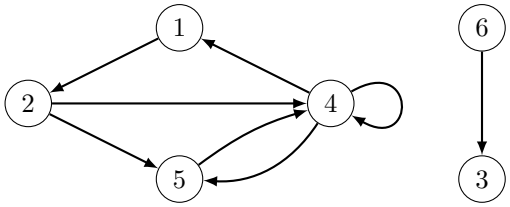
Définition 1.3
- On appelle **ordre** d'un graphe le nombre de ses sommets, *i.e* c'est card(S).
- On appelle **taille** d'un graphe le nombre de ses arêtes, *i.e* c'est card(A).

1.1.2 Graphes orientés

Définition 1.4 Un **graphe** orienté G est la donnée d'un couple $G = (S, A)$ tel que :
- S est un ensemble fini de sommets,
- A est un ensemble de couples ordonnés de sommets $(s_i, s_j) \in S^2$.

Un couple (s_i, s_j) est appelé un **arc**, et est représenté graphiquement par $s_i \to s_j$, s_i est le sommet initial ou origine, et s_j le sommet terminal ou extrémité. L'arc $a = (s_i, s_j)$ est dit **sortant** en s_i et **incident** en s_j, et s_j est un **successeur** de s_i, tandis que s_i est un **prédécesseur** de s_j. L'ensemble des successeurs d'un sommet $s_i \in S$ est noté $Succ(s_i) = \{s_j \in S, (s_i, s_j) \in A\}$. L'ensemble des prédécesseurs d'un sommet $s_i \in S$ est noté $Pred(s_i) = \{s_j \in S, (s_j, s_i) \in A\}$.

Par exemple,



représente le graphe orienté $G = (S, A)$ avec $S = \{1, 2, 3, 4, 5, 6\}$ et
$A = \{(1, 2), (2, 4), (2, 5), (4, 1), (4, 4), (4, 5), (5, 4), (6, 3)\}$.

Définition 1.5
- Une **boucle** est un arc reliant un sommet à lui-même. Un graphe orienté est dit **élémentaire** s'il ne contient pas de boucle.
- Un graphe orienté est un **p-graphe** s'il comporte au plus p arcs entre deux sommets. Le plus souvent, on étudiera des 1-graphes.

1.2 Degré dans un graphe

Définition 1.6 (degré d'un sommet)
- Dans un graphe non-orienté, le **degré** d'un sommet est le nombre d'arêtes incidentes à ce sommet (une boucle comptant pour 2). Dans le cas d'un graphe simple, on aura $d(s) = |Adj(s)|$.
- Dans un graphe orienté, le **demi-degré extérieur** ou **demi-degré sortant** d'un sommet s, noté $d^+(s)$, est le nombre d'arcs partant de s, *i.e.* de la forme (s, v) avec $v \in S$. Dans le cas d'un 1-graphe, on aura $d^+(s) = |Succ(s)|$.
  De même, le **demi-degré intérieur** ou **demi-degré entrant** d'un sommet s, noté $d^-(s)$, est le nombre d'arcs arrivant en s, *i.e* de la forme (v, s) avec $v \in S$. Dans le cas d'un 1-graphe, on aura $d^-(s) = |Pred(s)|$.
  Le degré d'un sommet s est alors la somme des degré entrant et sortant : $d(s) = d^+(s) + d^-(s)$

1.3 Différents types de graphes

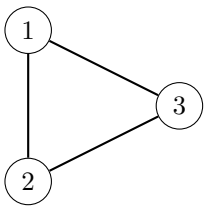
Tout d'abord, on peut vouloir attribuer des valeurs aux arcs ou arêtes pour tenir compte de contraintes : distance, coût ...

Définition 1.7 (graphe valué) Un graphe valué $G = (S, A, \nu)$ est un graphe (S, A) (orienté ou non-orienté) muni d'une application $\nu : A \to \mathbb{R}$. L'application ν est appelée valuation du graphe.
On peut étendre cette valuation en posant $\forall (x, y) \in S^2$, $\nu(x, y) = +\infty$ si $(x, y) \notin A$.

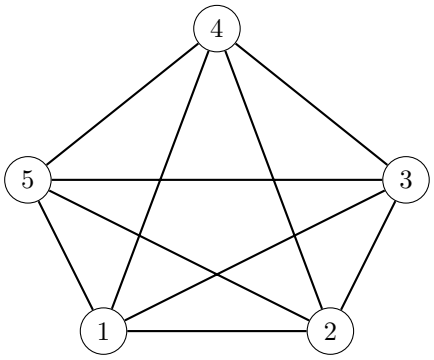
Définition 1.8 (graphe complet) Un 1-graphe orienté élémentaire est dit **complet** s'il comporte un arc (s_i, s_j) et un arc (s_j, s_i) pour tout couple de sommets différents $s_i, s_j \in S$. De même, un graphe non-orienté simple est dit complet s'il comporte une arête $\{s_i, s_j\}$ pour toute paire de sommets différents $s_i, s_j \in S$. On note K_n un graphe complet d'ordre n.

Exemple :

K_3 :



K_5 :



Définition 1.9 (sous-graphe induit, sous-graphe partiel) Soit $G = (S, A)$ un graphe (orienté ou non orienté) alors :
- un **sous-graphe induit** de G est un graphe G' ayant pour sommets **un sous-ensemble S' des sommets de G** et pour arcs/arêtes uniquement ceux de G joignant les sommets de S', ce qui s'écrit :

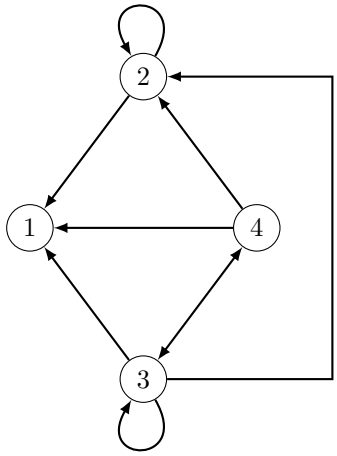
$$G' = (S', A') \quad \text{avec} \quad S' \subset S \quad \text{et} \quad A' = \{(x, y) \in A \ , \ x \in S' \text{ et } y \in S'\}.$$

- un **sous-graphe partiel** de G est un graphe G' ayant pour sommets **un sous-ensemble S' des sommets de G** et pour arcs/arêtes un sous-ensemble de ceux de G joignant les sommets de S', ce qui s'écrit :

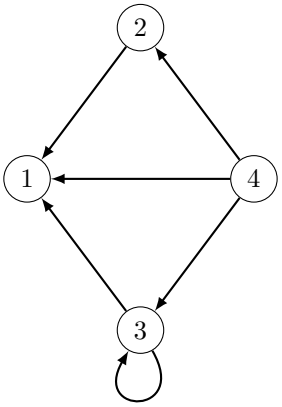
$$G' = (S', A') \quad \text{avec} \quad S' \subset S \quad \text{et} \quad A' \subset \{(x, y) \in A \ , \ x \in S' \text{ et } y \in S'\}.$$

Exemple :
- Graphe partiel défini par $A' = A \setminus \{(2, 2), (3, 2), (3, 4)\}$:

G :

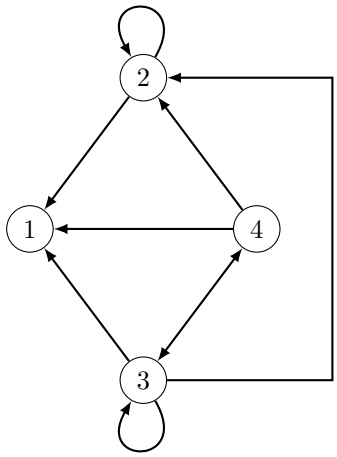


G' :

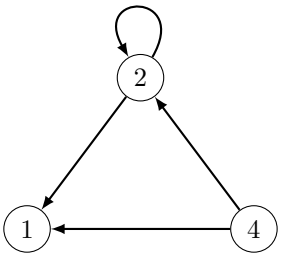


– Sous-graphe de G induit par $S' = \{1, 2, 4\}$:

G :



G' :

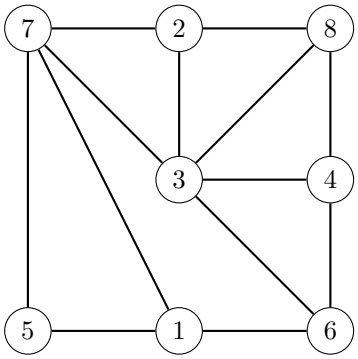


Associé à la notion de sous-graphe, on définit :

Définition 1.10 (clique, stable) Soit $G = (S, A)$ un graphe (orienté ou non orienté), alors :
- une **clique** est un sous-graphe (induit) complet de G,
- un **stable** est un sous-graphe induit de G sans arcs/arêtes.

Trouver une clique d'ordre k dans un graphe est un problème NP-complet, ce qui implique qu'il n'existe pas à ce jour un algorithme résolvant ce problème de façon exacte avec une complexité polynomiale (elle est exponentielle). Le problème consistant à trouver la plus grande clique d'un graphe est appelé « problème de la clique maximum », et est encore plus difficile ! (il est NP-difficile)

Exemple : Trouver le plus grand stable et la plus grande clique du graphe suivant :



Dans le graphe G l'ensemble de sommets $\{1, 5, 7\}$ induit une clique maximum, alors que $\{2, 4, 5\}$ induit un stable maximum (il y en a d'autres).

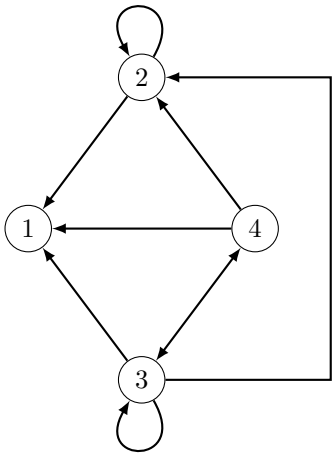
1.4 Représentations des graphes

1.4.1 Représentation par listes d'adjacence

Soit le graphe $G = (S, A)$ d'ordre n. On suppose que les sommets de S sont numérotés de 1 à n. La représentation par listes d'adjacence de G consiste en un tableau T de n listes, une pour chaque sommet de S. Pour chaque sommet $s_i \in S$, la liste d'adjacence $T[s_i]$ est une liste (chaînée) de tous les sommets s_j tels qu'il existe un arc $(s_i, s_j) \in A$ ou une arête $\{s_i, s_j\} \in A$. Les sommets de chaque liste d'adjacence sont généralement listés selon un ordre arbitraire.

Dans le cas de graphes non orientés, pour chaque arête $\{s_i, s_j\}$, on aura s_j qui appartiendra à la liste chaînée de $T[s_i]$, et aussi s_i qui appartiendra à la liste chaînée de $T[s_j]$.

Exemple : Soit le $G = (S, A)$ le graphe orienté suivant :



Alors la liste d'adjacence associée à ce graphe est :

1	→				
2	→	2	1		
3	→	1	4	3	2
4	→	1	2	3	

Taille mémoire nécessaire : si le graphe G est orienté, la somme des longueurs des listes d'adjacence est égale au nombre d'arcs de A, puisque l'existence d'un arc (s_i, s_j) se traduit par la présence de s_j dans la liste d'adjacence de $T[s_i]$. En revanche, si le graphe n'est pas orienté, la somme des longueurs de toutes les listes d'adjacence est égale à deux fois le nombre d'arêtes du graphe, puisque si $\{s_i, s_j\}$ est une arête, alors s_i appartient à la liste d'adjacence de $T[s_j]$, et vice versa. Par conséquent, la liste d'adjacence d'un graphe ayant n sommets et m arcs ou arêtes nécessite de l'ordre de $\mathcal{O}(n + m)$ emplacements mémoires.

Opérations sur les listes d'adjacence : il n'existe pas de moyen plus rapide que de parcourir la liste d'adjacence de $T[s_i]$ jusqu'à trouver s_j pour tester l'existence d'un arc (s_i, s_j) ou d'une arête $\{s_i, s_j\}$ avec une représentation par liste d'adjacence. En revanche, le calcul du degré d'un sommet, ou l'accès à tous les successeurs d'un sommet, est très efficace : il suffit de parcourir la liste d'adjacence associée au sommet. D'une façon plus générale, le parcours de l'ensemble des arcs/arêtes nécessite le parcours de toutes les listes d'adjacence, et prendra un temps de l'ordre de p, où p est le nombre d'arcs/arêtes.
En revanche, le calcul des prédécesseurs d'un sommet est mal aisé avec cette représentation, et nécessite le parcours de toutes les listes d'adjacences de T. Une solution dans le cas où l'on a besoin de connaître les prédécesseurs d'un sommet est de maintenir, en plus de la liste d'adjacence des successeurs, la liste d'adjacence des prédécesseurs.

1.4.2 Représentation par matrices d'adjacence

Soit le graphe $G = (S, A)$ d'ordre n. On suppose que les sommets de S sont numérotés de 1 à n. La représentation par matrice d'adjacence de G consiste en une matrice booléenne M de taille $n \times n$ telle que $M[i][j] = 1$ si $(i, j) \in A$, et $M[i][j] = 0$ sinon.

Dans le cas de graphes non orientés, la matrice est symétrique par rapport à sa diagonale descendante. Dans ce cas, on peut ne mémoriser que la composante triangulaire supérieure de la matrice d'adjacence.

Exemple : On considère le même graphe qu'au cas précédent, sa matrice d'adjacence est :

$$\begin{array}{c} \nearrow \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \end{array}$$

Taille mémoire nécessaire : la matrice d'adjacence d'un graphe ayant n sommets nécessite de l'ordre de $\mathcal{O}(n^2)$ emplacements mémoire. Si le nombre d'arcs est très inférieur à n^2, cette représentation est donc loin d'être optimale.

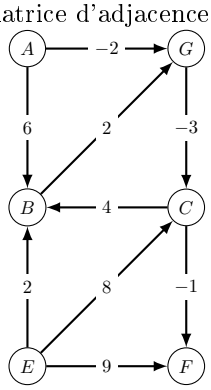
Opérations sur les matrices d'adjacence : le test de l'existence d'un arc ou d'une arête avec une représentation par matrice d'adjacence est immédiat (il suffit de tester directement la case correspondante de la matrice). En revanche, connaître le degré d'un sommet nécessite le parcours de toute une ligne (ou toute une colonne) de la matrice. D'une façon plus générale, le parcours de l'ensemble des arcs/arêtes nécessite la consultation de la totalité de la matrice, et prendra un temps de l'ordre de n^2. Si le nombre d'arcs est très inférieur à n^2, cette représentation est donc loin d'être optimale.

1.4.3 Représentation matricielle des graphes valués

Un graphe valué $G = (S, A, \nu)$ peut être représenté par la matrice des valuations :

$$W \in \mathscr{M}_n(\overline{\mathbb{R}}) \quad \text{telle que} \quad W_{i,j} = \begin{cases} \infty & \text{si } (s_i, s_j) \notin A \\ \nu((s_i, s_j)) & \text{si } (s_i, s_j) \in A \end{cases}$$

Exemple : La matrice d'adjacence du graphe valué suivant est :



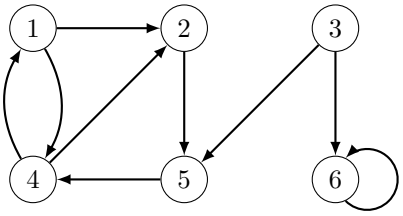
$$W = \begin{pmatrix} \infty & 6 & \infty & \infty & \infty & -2 \\ \infty & \infty & \infty & \infty & \infty & 2 \\ \infty & 4 & \infty & \infty & -1 & \infty \\ \infty & 2 & 8 & \infty & 9 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & -3 & \infty & \infty & \infty \end{pmatrix}$$

1.5 Notions de chemin, chaîne, cycle et circuit

Définition 1.11 (Cas des graphes orientés) Soit $G = (S, A)$ un graphe orienté,

- Un **chemin** d'un sommet u vers un sommet v est une séquence $< s_0, s_1, s_2, \ldots, s_k >$ de sommets tels que $u = s_0$, $v = s_k$ et $(s_{i-1}, s_i) \in A$ pour tout $i \in \{1, \ldots, k\}$. On dira que le chemin contient les sommets $s_0, s_1, \ldots, s_k$ et les arcs $(s_0, s_1), (s_1, s_2), \ldots, (s_{k-1}, s_k)$.
- La **longueur** du chemin est le nombre d'arcs dans le chemin, c'est-à-dire k.
- S'il existe un chemin de u à v, on dira que v est accessible à partir de u.
- Un chemin est **élémentaire** si les sommets qu'il contient sont tous distincts.
- Un chemin $< s_0, s_1, ..., s_k >$ forme un **circuit** si $s_0 = s_k$ et si le chemin comporte au moins un arc ($k \geq 1$). Ce circuit est **élémentaire** si, en plus, les sommets $s_1, s_2, \ldots, s_k$ sont tous distincts. Une boucle est un circuit de longueur 1.

Exemple Considérons par exemple le graphe orienté suivant :



- Un chemin élémentaire dans ce graphe est $< 1, 4, 2, 5 >$.
- Un chemin non élémentaire dans ce graphe est $< 3, 6, 6, 6 >$.
- Un circuit élémentaire dans ce graphe est $< 1, 2, 5, 4, 1 >$.
- Un circuit non élémentaire dans ce graphe est $< 1, 2, 5, 4, 2, 5, 4, 1 >$.

Dans le cas des graphes non orientés, seule la terminologie change.

Définition 1.12 (Cas des graphes non orientés) Si $G = (S, A)$ est un graphe non orienté, on parlera de **chaîne** au lieu de chemin, et de **cycle** au lieu de circuit. Dans le cas d'un cycle, toutes les arêtes doivent être distinctes. Un graphe sans cycle est dit **acyclique**.

Lemme 1.13 (de König) Dans un graphe, s'il existe un chemin/chaîne d'un sommet u vers un sommet v, alors il existe un chemin élémentaire de u vers v.

La notion de longueur de chemin nous permet ensuite de définir la notion de distance dans un graphe.

Définition 1.14 Soit un graphe $G = (S, A)$. On appelle :
- **distance** d'un sommet à un autre la longueur du plus court chemin/chaîne entre ces deux sommets, ou ∞ s'il n'y a pas un tel chemin/chaîne :

$$\forall x, y \in S, \quad d(x, y) = \begin{cases} k & \text{si le plus court chemin de } x \text{ vers } y \text{ est de longueur } k \\ \infty & \text{sinon} \end{cases}$$

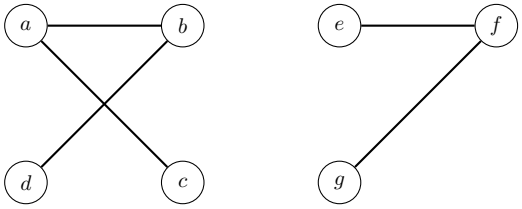
- **diamètre** du graphe la plus grande distance entre deux sommets.

1.6 Notion de connexité

1.6.1 Graphes et sous-graphes connexes

Définition 1.15 Un graphe non orienté est **connexe** si chaque sommet est accessible à partir de n'importe quel autre. Autrement dit, si pour tout couple de sommets distincts $(s_i, s_j) \in S^2$, il existe une chaîne entre s_i et s_j.

Exemple Par exemple, le graphe non orienté suivant :



n'est pas connexe car il n'existe pas de chaîne entre les sommets a et e. En revanche, le sous-graphe induit par les sommets $\{a, b, c, d\}$ est connexe.

Définition 1.16
- Une composante connexe d'un graphe non-orienté G est un sous-graphe G' de G qui est connexe et maximal (c'est-à-dire qu'aucun autre sous-graphe connexe de G ne contient G').
- Un graphe est dit connexe si et seulement si il admet une unique composante connexe.

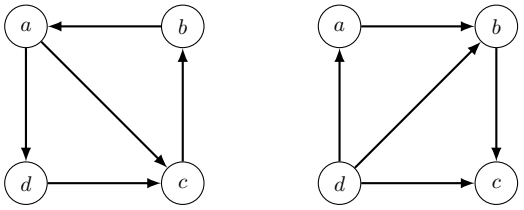
Par exemple, le graphe précédent est composé de 2 composantes connexes : la première est le sous-graphe induit par les sommets $\{a, b, c, d\}$, et la seconde est le sous-graphe induit par les sommets $\{e, f, g\}$.

Définition 1.17
- Un **point d'articulation** d'un graphe est un sommet dont la suppression augmente le nombre de composantes connexes.
- Un **isthme** est une arête dont la suppression a le même effet.
- Un **ensemble d'articulation** $E \subset S$ d'un graphe connexe $G = (S, A)$ est un ensemble de sommets tel que le sous-graphe G' déduit de G par suppression des sommets de E ne soit plus connexe.

1.6.2 Graphes et sous-graphes fortement connexes

On retrouve les différentes notions de connexités dans les graphes orientés, en remplaçant naturellement la notion de chaîne par celle de chemin : on parle de graphe **fortement connexe** au lieu de connexe, de **composante fortement connexe** au lieu de composante connexe.

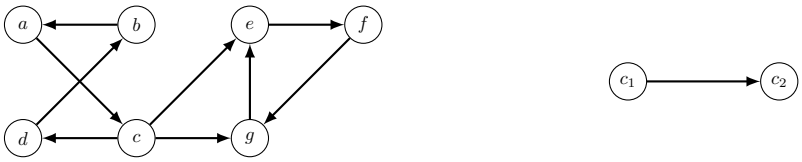
Exemple Le graphe de gauche est fortement connexe, tandis que celui de droite ne l'est pas.



Le graphe non fortement connexe possède en fait quatre composantes fortement connexes : les sous-graphes possèdant un seul sommet : $\{a\}$ ou $\{b\}$ ou $\{c\}$ ou $\{d\}$.

Définition 1.18 Soit $G = (S, A)$ un graphe orienté. On appelle graphe réduit de G le graphe G_r dont les sommets $c_1, \ldots, c_p$ sont les composantes fortement connexes de G, et il existe un arc entre c_i et c_j si et seulement s'il existe au moins un arc entre un sommet de G_i et un sommet de G_j dans le graphe G. On vérifie que le graphe G_r est sans circuit.

Exemple Un graphe (à gauche) ayant deux composantes fortement connexes, et son graphe réduit (à droite) :

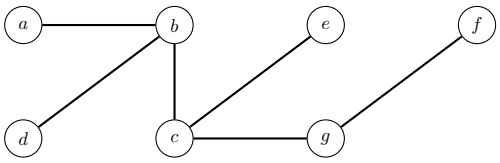


Le graphe G a pour composantes connexes les sous-graphes G_1, ayant pour sommets $\{a, b, c, d\}$, et G_2, ayant pour sommets $\{e, f, g\}$. Il existe un arc entre un sommet de G_1 et un sommet de G_2, on a donc un arc entre c_1 et c_2, mais il n'existe aucun arc entre un sommet de G_2 et un sommet de G_1, et il n'existe donc pas d'arc (c_2, c_1).

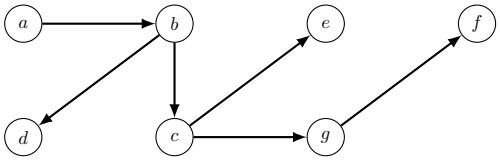
1.7 Arbres et Arborescences

Définition 1.19 Un arbre est un graphe non orienté G qui vérifie une des conditions équivalentes suivantes :
- G est connexe et acyclique
- G est sans cycle et possède $n - 1$ arêtes
- G est connexe et admet $n - 1$ arêtes
- G est sans cycle, et en ajoutant une arête, on crée un et un seul cycle élémentaire,
- G est connexe, et en supprimant une arête quelconque, il n'est plus connexe,
- Il existe une chaîne et une seule entre 2 sommets quelconques de G.

Par exemple, le graphe suivant est un arbre :



On appelle forêt un graphe dont chaque composante connexe est un arbre.
Une arborescence est un graphe orienté sans circuit admettant une racine $s_0 \in S$ telle que pour tout autre sommet $s_i \in S$, il existe un chemin unique allant de s_0 vers s_i. Si l'arborescence comporte n sommets, alors elle comporte exactement $n - 1$ arcs. Par exemple, le graphe suivant est une arborescence de racine a :



Chapitre 2

Parcours de graphes

Beaucoup de problèmes sur les graphes nécessitent que l'on parcourt l'ensemble des sommets et des arcs/arêtes du graphe. Nous allons étudier dans ce chapitre les deux principales stratégies d'exploration :

- le parcours en largeur consiste à explorer les sommets du graphe niveau par niveau, à partir d'un sommet donné ;
- le parcours en profondeur consiste, à partir d'un sommet donné, à suivre un chemin le plus loin possible, puis à faire des retours en arrière pour reprendre tous les chemins ignorés précédemment.

Dans les deux cas, l'algorithme procède par coloriage des sommets :

- Initialement, tous les sommets sont coloriés à blanc (traduisant le fait qu'ils n'ont pas encore été "découverts").
- Lorsqu'un sommet est "découvert" (autrement dit, quand on arrive pour la première fois sur ce sommet), il est colorié en gris. Le sommet reste gris tant qu'il reste des successeurs de ce sommet qui sont blancs (autrement dit, qui n'ont pas encore été découverts).
- Un sommet est colorié en noir lorsque tous ses successeurs sont gris ou noirs (autrement dit, lorsqu'ils ont tous été découverts).

De façon pratique, on va utiliser une liste "d'attente au coloriage en noir" dans laquelle on va stocker tous les sommets gris : un sommet est mis dans la liste d'attente dès qu'il est colorié en gris. Un sommet gris dans la liste d'attente peut faire rentrer dans la liste ses successeurs qui sont encore blancs (en les coloriant en gris). Quand tous les successeurs d'un sommet gris de la liste d'attente sont soit gris soit noirs, il est colorié en noir et il sort de la liste d'attente.

La différence fondamentale entre le parcours en largeur et le parcours en profondeur provient de la façon de gérer cette liste d'attente au coloriage en noir : le parcours en largeur utilise une file d'attente, où le premier sommet arrivé dans la file est aussi le premier à en sortir, tandis que le parcours en profondeur utilise une pile, où le dernier sommet arrivé dans la pile est le premier à en sortir.

2.1 Arborescence couvrante associée à un parcours

On parcourt un graphe à partir d'un sommet donné s_0. Ce parcours va permettre de découvrir tous les sommets accessibles depuis s_0, c'est à dire tous les sommets pour lesquels il existe un chemin (ou une chaîne) depuis s_0. En même temps que l'on effectue ce parcours, on construit l'arborescence de découverte des sommets accessibles depuis s_0, appelée arborescence couvrante de s_0. Cette arborescence contient un arc (s_i, s_j) si et seulement s_i le sommet s_j a été découvert à partir du sommet s_i (autrement dit, si c'est le sommet s_i qui a fait entrer s_j dans la file d'attente). Ce graphe est effectivement une arborescence, dans la mesure où chaque sommet a au plus un prédécesseur, à partir duquel il a été découvert. La racine de cette arborescence est s_0, le sommet à partir duquel on a commencé le parcours.

L'arborescence associée à un parcours de graphe sera mémorisée dans un tableau Π tel que $\Pi[s_j] = s_i$ si s_j a été découvert à partir de s_i, et $\Pi[s_k] = nil$ si s_k est la racine, ou s'il n'existe pas de chemin de la racine vers s_k.

2.2 Parcours en largeur (Breadth First Search = BFS)

Le parcours en largeur est obtenu en gérant la liste d'attente au coloriage comme une file d'attente (FIFO = First In First Out). Autrement dit, on enlève à chaque fois le plus vieux sommet gris dans la file d'attente, et on introduit tous les successeurs blancs de ce sommet dans la file d'attente, en les coloriant en gris.
Structures de données utilisées :

- On utilise une file F, pour laquelle on suppose définies les opérations init_file(F) qui initialise la file F à vide, ajoute_fin_file(F, s) qui ajoute le sommet s à la fin de la file F, est_vide(F) qui retourne vrai si la file F est vide et faux sinon, et enleve_debut_file(F, s) qui enlève le sommet s au début de la file F.
- On utilise un tableau Π qui associe à chaque sommet le sommet qui l'a fait entrer dans la file, et un tableau *couleur* qui associe à chaque sommet sa couleur (blanc, gris ou noir).
- On va en plus utiliser un tableau d qui associe à chaque sommet son niveau de profondeur par rapport au sommet de départ s_0 (autrement dit, $d[si]$ est la longueur du chemin dans l'arborescence Π de la racine s_0 jusque s_i). Ce tableau sera utilisé plus tard.

L'algorithme 1 présente la méthode du parcours d'un graphe en largeur.

Algorithme 1 : parcours_en_largeur(BFS)(S, A, s_0, Π, d)

Entrées : S ensemble des sommets, A ensemble des arc/arêtes, s_0 sommet de départ
Sorties : Π arborescence couvrante, d tableau longueur des chemins depuis s_0

```
1  Declaration : couleur tableau des couleurs des sommets, F file
2  init_file(F)
3  pour tout sommet s_i ∈ S faire
4      Π[s_i] ← nil
5      d[s_i] ← ∞
6      couleur[s_i] ← blanc
7  d[s_0] ← 0
8  ajoute_fin_file(F, s_0)
9  couleur[s_0] ← gris
10 tant que est_vide(F) = faux faire
11     enleve_debut_file(F, s_i)
12     pour tout s_j ∈ succ(s_i) faire
13         si couleur[s_j] = blanc alors
14             ajoute_fin_file(F, s_j)
15             couleur[s_j] ← gris
16             Π[s_j] ← s_i
17             d[s_j] ← d[s_i] + 1
18     couleur[s_i] ← noir
```

Complexité : Chaque sommet (accessible depuis s_0) est mis, puis enlevé, une fois dans la file. À chaque fois qu'on enlève un sommet de la file, on parcourt tous ses successeurs, de telle sorte que chaque arc (ou arête) du graphe sera utilisé une fois dans l'algorithme. Par conséquent, si le graphe contient n sommets (accessibles à partir de s_0) et p arcs/arêtes, alors BFS sera en :

- $O(n^2)$ dans le cas d'une implémentation par matrice d'adjacence,
- $O(n + p)$ dans le cas d'une implémentation par listes d'adjacence.

2.3 Applications du parcours en largeur

depuis s_0. À la fin de l'exécution de BFS(S, A, s_0), chaque sommet est soit noir soit blanc. Les Le parcours en largeur peut être utilisé pour rechercher l'ensemble des sommets accessibles sommets noirs sont ceux accessibles depuis s_0 ; les sommets blancs sont ceux pour lesquels il n'existe pas de chemin/chaîne à partir de s_0. D'une façon plus générale, le parcours en largeur permet de déterminer les composantes connexes d'un graphe non orienté. Pour cela, il suffit d'appliquer l'algorithme de parcours en largeur à partir d'un sommet blanc quelconque. À la suite de quoi, tous les sommets en noirs appartiennent à la première composante connexe. S'il reste des sommets blancs, cela implique qu'il y a d'autres composantes connexes. Il faut alors relancer le parcours en largeur sur le sous-graphe induit par les sommets blancs, pour découvrir une autre composante connexe. Le nombre de fois où l'algorithme de parcours en largeur a été lancé correspond au nombre de composantes connexes. Le parcours en largeur peut aussi être utilisé pour chercher le plus court chemin (en nombre d'arcs ou arêtes) entre la racine s_0 et chacun des autres sommets du graphe accessibles depuis s_0. Pour cela, il suffit de remonter dans l'arborescence Π du sommet concerné jusqu'à la racine s_0. L'algorithme 2 (récursif) affiche le plus court chemin pour aller de s_0 à s_j.

Algorithme 2 : plus_court_chemin(s_0, s_j, Π)

Entrées : s_0 sommet de départ, s_j sommet d'arrivée, Π arborescence couvrante

```
1 si s_0 = s_j alors
2     afficher(s_0)
3 sinon si Π[s_j] = nil alors
4     afficher("pas de chemin")
5 sinon
6     plus_court_chemin(s_0, Π[s_j], Π)
7     afficher(s_j)
```

2.4 Parcours en profondeur (Depth First Search = DFS)

Le parcours en profondeur est obtenu en gérant la liste d'attente au coloriage en noir comme une pile (LIFO = Last In First Out). Autrement dit, on considère à chaque fois le dernier sommet gris entré dans la pile, et on introduit devant lui tous ses successeurs blancs.
Structures de données utilisées :

- On utilise une pile P, pour laquelle on suppose définies les opérations init_pile(P) qui initialise la pile P à vide, empile(P, s) qui ajoute s au sommet de la pile P, est_vide(P) qui retourne vrai si la pile P est vide et faux sinon, sommet(P) qui retourne le sommet s au sommet de la pile P, et depile(P, s) qui enlève s du sommet de la pile P.
- On utilise, comme pour le parcours en largeur, un tableau Π qui associe à chaque sommet le sommet qui l'a fait entrer dans la pile, et un tableau *couleur* qui associe à chaque sommet sa couleur (blanc, gris ou noir).
- On va en plus mémoriser pour chaque sommet s_i :
  - $dec[s_i]$ = date de découverte de s_i (passage en gris)
  - $fin[s_i]$ = date de fin de traitement de s_i (passage en noir) où l'unité de temps est une itération. La date courante est mémorisée dans la variable *tps*.

L'algorithme 3 présente la méthode du parcours en profondeur.

Algorithme 3 : parcours_en_profondeur(DFS)(S, A, s_0, Π, dec, fin)

Entrées : S ensemble des sommets, A ensemble des arc/arêtes, s_0 sommet de départ
Sorties : Π arborescence couvrante, dec tableau des dates de découverte, fin tableau des dates de fin de traitement

```
1  Declaration : couleur tableau des couleurs des sommets, tps date courante, P pile
2  init_file(P)
3  pour tout sommet s_i ∈ S faire
4  |  Π[s_i] ← nil
5  |  couleur[s_i] ← blanc
6  tps ← 0
7  dec[s_0] ← tps
8  empile(P, s_0)
9  couleur[s_0] ← gris
10 tant que est_vide(P) = faux faire
11 |  tps ← tps + 1
12 |  s_i ← sommet(P)
13 |  si ∃s_j ∈ succ(s_i) tel que couleur[s_j] = blanc alors
14 |  |  empile(P, s_j)
15 |  |  couleur[s_j] ← gris
16 |  |  Π[s_j] ← s_i
17 |  |  dec[s_j] ← tps
18 |  sinon
19 |  |  /* tous les successeurs de s_i sont gris ou noirs */
20 |  |  depile(P, s_i)
21 |  |  couleur[s_i] ← noir
22 |  |  fin[s_i] ← tps
```

Complexité : Chaque sommet (accessible depuis s_0) est mis, puis enlevé, une fois dans la pile, comme dans BFS. Par conséquent, si le graphe contient n sommets (accessibles à partir de s_0) et p arcs/arêtes, alors DFS sera en :

- $O(n^2)$ dans le cas d'une implémentation par matrice d'adjacence,
- $O(n + p)$ dans le cas d'une implémentation par listes d'adjacence.

Cet algorithme peut s'écrire récursivement sans utiliser de pile explicite (voir algorithme 4).

Dans ce cas, les structures de données Π, *couleur*, *tps*, *dec* et fin sont des variables globales et il est nécessaire de les initialiser auparavant, à l'exception des deux dernières (voir algorithme 5).

La complexité de cet algorithme est la même que sa version itérative.

Algorithme 4 : parcours en profondeur récursif (DFSrec)(S, A, s_0)

Entrées : S ensemble des sommets, A ensemble des arcs/arêtes, s_0 sommet de départ

```
1  dec[s_0] ← tps
2  tps ← tps + 1
3  couleur[s_0] ← gris
4  pour tout s_j ∈ succ(s_0) faire
5      si couleur[s_j] = blanc alors
6          Π[s_j] ← s_0
7          DFSrec(S, A, s_j)
8  couleur[s_0] ← noir
9  fin[s_0] ← tps
10 tps ← tps + 1
```

Algorithme 5 : initDFSrec(S)

Entrées : S ensemble des sommets

```
1  pour tout sommet s_i ∈ S faire
2      Π[s_i] ← nil
3      couleur[s_i] ← blanc
4  tps ← 0
```

2.5 Applications du parcours en profondeur

2.5.1 Recherche des composantes connexes

Pour rechercher les composantes connexes d'un graphe non orienté, on peut procéder comme avec le parcours en largeur, c'est-à-dire appeler itérativement DFSrec à partir de sommets blancs, jusqu'à ce que tous les sommets soient noirs ; le nombre d'appels à DFSrec correspond au nombre de composantes connexes (algorithme 6)

Algorithme 6 : parcours en profondeur global DFSglobal(S, A)

Entrées : S ensemble des sommets, A ensemble des arêtes

```
1  initDFSrec()
2  tant que ∃s_i ∈ S tel que couleur[s_i] = blanc faire
3      /* s_i appartient à une nouvelle composante connexe */
4      DFSrec(S, A, s_i)
```

2.5.2 Recherche de circuits/cycles

Lors du parcours en profondeur d'un graphe orienté (resp. non orienté), si un successeur s_j du sommet "courant" s_i au sommet de la pile est déja gris, cela implique qu'il existe un chemin (resp. une chaîne (à moins que ce ne soit son père dans l'arborescence)) permettant d'aller de s_j vers s_i , et donc qu'il existe un circuit (resp. un cycle). Ainsi, un algorithme pour détecter les circuits (resp. les cycles) d'un graphe peut être obtenu en rajoutant dans l'algorithme DFSrec l'instruction si $couleur[s_j] = gris$ alors afficher("existence d'un circuit") (resp. si $couleur[s_j] = gris$ et $s_j \neq \Pi[s_0]$ alors afficher("existence d'un cycle")) juste à la fin de la boucle "pour tout $sj \in succ(s_0)$ faire" (après la ligne 7). De la même façon, le parcours en profondeur permet de découvrir si un graphe possède plusieurs chemins élémentaires entre deux sommets. En effet, si lors du parcours en profondeur, un successeur s_j du sommet "courant" s_i au sommet de la pile est déjà noir, cela implique qu'il existe déjà un chemin permettant d'aller d'un ancêtre de s_i vers s_j.

2.5.3 Tri topologique des sommets d'un graphe orienté

Le tri topologique d'un graphe orienté sans circuit $G = (S, A)$ consiste à ordonner linéairement tous ses sommets de telle sorte que si l'arc $(u, v) \in A$, alors u apparaisse avant v. (Si le graphe comporte des circuits, aucun ordre linéaire n'est possible.) Le tri topologique d'un graphe peut être vu comme un alignement de ses sommets le long d'une ligne horizontale de telle sorte que tous les arcs du graphe soient orientés de gauche à droite.

Théorème 2.1 *Après l'exécution de DFSglobal(S, A), pour tout arc $(s_i, s_j) \in A$, on a $fin[s_j] < fin[s_i]$.*

En effet, à l'appel de DFSrec(S, A, s_i),
- si s_j est noir alors $fin[s_j] < tps < fin[s_i]$
- si s_j est blanc alors $dec[s_i] = tps < dec[s_j] < fin[s_j] < fin[s_i]$
- s_j ne peut pas être gris car ça impliquerait l'existence d'un circuit.

Par conséquent, pour obtenir un tri topologique des sommets d'un graphe, il suffit d'exécuter DFSglobal, puis de trier les sommets par ordre de valeur de fin décroissante. D'une façon plus générale, les graphes orientés sans circuit sont utilisés dans de nombreuses applications pour représenter des précédences entre événements : les sommets représentent les événements, et les arcs les relations de précédence. Dans ce cas, un tri topologique permet de trier les événements de telle sorte qu'un événement n'apparaît qu'après tous les événements qui doivent le précéder.

2.5.4 Recherche des composantes fortement connexes d'un graphe orienté

L'algorithme 7 permet de rechercher les composantes fortement connexes d'un graphe orienté.

Algorithme 7 : recherche des composantes fortement connexes $(S, A, nbcfc)$

Entrées : S ensemble des sommets, A ensemble des arcs/arêtes
Sorties : $nbcfc$ nombre de composantes fortement connexes

```
1 DFSglobal(S, A)
2 inverser les sens de tous les arcs du graphe
3 trier les sommets par ordre de valeurs de fin décroissant dans un tableau t
4 initDFSrec(S)
5 nbcfc ← 0
6 tant que ∃si ∈ S tel que couleur[si] = blanc faire
7 |   soit s_j le prochain sommet blanc dans le tableau t
8 |   DFSrec(S, A, s_j)
9 |   nbcfc ← nbcfc + 1
```

Complexité : Cet algorithme a la même complexité qu'un parcours en profondeur, pour peu que l'on trie les sommets par ordre de numéro de fin décroissant au fur et à mesure du parcours en profondeur (autrement dit, à chaque fois que l'on affecte un numéro de fin à un sommet, on l'insère au sommet d'une liste).

Chapitre 3

Plus courts chemins

3.1 Problème du plus court chemin

On se place dans le cas des graphes orientés valués $G = (S, A, \nu)$. Mais les résultats et algorithmes présentés se généralisent facilement aux cas des graphes non orientés valués. Une autre solution consiste à transformer le graphe non-orienté en un graphe orienté en remplaçant une arête entre deux sommets par deux arcs de sens inverse entre ces sommets.

Définition 3.1
- Le **coût ou poids d'un chemin** $c =< s_0, s_1, \ldots, s_k >$ est égale à la somme des valuations des arcs composant le chemin, c'est à dire,

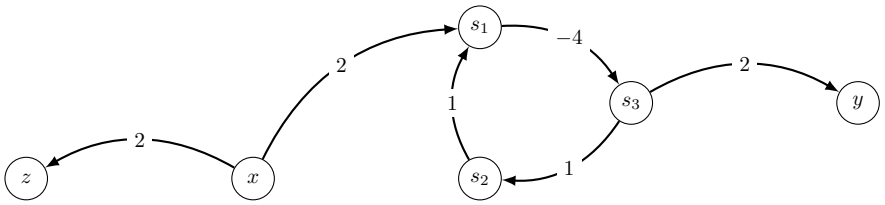
$$L(c) = \sum_{i=1}^{k} \nu(s_{i-1}, s_i)$$

- Le **coût d'un plus court chemin** entre deux sommets s_i et s_j est noté $\delta(s_i, s_j)$ et est défini par :

$$\delta(s_i, s_j) = \begin{cases} \min\{L(c) \,/\, c = \text{chemin de } s_i \text{ à } s_j\} & \text{s'il existe au moins un chemin entre } s_i \text{ et } s_j \\ +\infty & \text{sinon} \end{cases}$$

Dans la recherche d'un plus court chemin de x à y, trois cas peuvent se présenter :
- Il n'existe aucun chemin de x à y (par exemple, si x et y appartiennent à deux composantes fortement connexes différentes de G).
- Il existe des chemins de x à y mais pas de plus court chemin.
- Il existe un plus court chemin de x à y.

Exemple On considère le graphe suivant :



Il existe des chemins de x à y (et même une infinité), mais il n'existe pas de plus court chemin de x à y (il suffit d'emprunter le cycle de poids négatif autant de fois que nécessaire). Il existe un plus court chemin de x à z mais pas de chemin de z à x.

Définition 3.2 Un circuit de longueur négative est appelé **circuit absorbant**.

Et alors, une condition nécessaire et suffisante d'existence de plus court chemin est donnée par le résultat suivant.

Proposition 3.3 *Dans un graphe orienté valué fortement connexe $G = (S, A, \nu)$, il existe un plus court chemin entre tout couple de sommets si et seulement s'il n'existe pas de circuit absorbant dans G.*

Preuve : On montre que si un chemin entre deux sommets x et y possède un circuit absorbant, alors $\pi(x, y) = -\infty$. En effet, dès que l'on parcourt le circuit absorbant, on diminue la longueur du chemin, et on peut donc diminuer cette longueur Ã l'infini. En augmentant "infiniment" le nombre de tours dans le circuit, on obtient $\pi(x, y) = -\infty$.

Définition 3.4 Étant donné un graphe orienté valué $G = (S, A, \nu)$ et un sommet origine $s_0 \in S$, le **problème des plus courts chemins à origine unique** consiste à calculer pour chaque sommet $s_j \in S$ le coût $\delta(s_0, s_j)$ du plus court chemin de s_0 à s_j.

On supposera que le graphe G ne comporte pas de circuit absorbant.

Variantes du problème :

- Si l'on souhaite calculer le plus court chemin allant d'un sommet s_0 vers un autre sommet s_i (la destination est unique), on pourra utiliser la résolution du problème précédant (qui calcule tous les plus courts chemins partant de s_0). En effet, on ne connaît pas d'algorithme plus efficace pour résoudre ce problème.
- Si l'on souhaite calculer tous les plus courts chemins entre tous les couples de sommets possibles, on pourrait aussi utiliser la résolution du problème précédent, mais dans ce cas, on n'obtiendrait pas un algorithme optimal. Il faudra utiliser dans ce cas un algorithme spécifique, par exemple l'algorithme de Floyd-Warshall.

Arborescence des plus courts chemins : On va en fait calculer non seulement les coût des plus courts chemins, mais aussi les sommets présents sur ces plus courts chemins. La représentation utilisée pour ces plus courts chemins est la même que celle utilisée pour les arborescences couvrantes calculées lors d'un parcours en largeur ou en profondeur d'un graphe. Cette arborescence est mémorisée dans un tableau Π tel que :
- $\Pi[s_0] = nil$,
- $\Pi[s_j] = s_i$ si $s_i \rightarrow s_j$ est un arc de l'arborescence.

Pour connaître le plus court chemin entre s_0 et un sommet s_k donné, il faudra alors "remonter" l'arborescence Π de s_k jusqu'à s_0 (cf l'algorithme d'affichage du plus court chemin trouvé par le parcours en largeur).

Nous allons étudier les deux algorithmes suivant qui permettent de résoudre ce problème :
- l'algorithme de Dijkstra résout ce problème lorsque tous les coûts sont positifs ou nuls.
- l'algorithme de Bellman-Ford résout ce problème lorsque les coûts sont positifs, nuls ou négatifs.

Les deux algorithmes procèdent de la même façon, selon une stratégie dite "gloutonne". L'idée est d'associer à chaque sommet $s_i \in S$ une valeur $d[s_i]$ qui représente une borne maximale du coût du plus court chemin entre s_0 et s_i (c'est-à-dire $\delta(s_0, s_i)$).
Ainsi, au départ :
- $d[s_0] = 0 = \delta(s_0, s_0)$, et
- $d[s_i] = +\infty \geq \delta(s_0, s_i)$ pour tout sommet $s_i \neq s_0$.

L'algorithme diminue alors progressivement les valeurs $d[s_i]$ associées aux différents sommets, jusqu'à ce qu'on ne puisse plus les diminuer, autrement dit, jusqu'à ce que $d[s_i] = \delta(s_0, s_i)$. Pour diminuer les valeurs de d, on va itérativement examiner chaque arc $s_i \rightarrow s_j$ du graphe, et regarder si on ne peut pas améliorer la valeur de $d[s_j]$ en passant par s_i.
Cette opération de diminution est appelée "relâchement de l'arc $s_i \rightarrow s_j$", et s'écrit :

Algorithme 8 : relacher_arc (s_i, s_j, ν, d, Π)

Entrées : s_i et s_j sont des sommets du graphe, ν valuation des arcs, d tableau des bornes max des coûts, Π arborescence couvrante
Sorties : d tableau des bornes maximum des coûts, Π arborescence couvrante

```
1 si d[s_j] > d[s_i] + ν(s_i, s_j) alors
2     /* il vaut mieux passer par s_i pour aller à s_j */
3     d[s_j] ← d[s_i] + ν(s_i, s_j)
4     Π[s_j] ← s_i
```

Les algorithmes de Dijkstra et Bellman-Ford procèdent tous les deux par relâchements successifs d'arcs. La différence entre les deux est que dans l'algorithme de Dijkstra, chaque arc est relâché une et une seule fois, tandis que dans l'algorithme de Bellman-Ford, chaque arc peut être relâché plusieurs fois.

3.1.1 Algorithme de Dijkstra

L'idée consiste à maintenir 2 ensembles disjoints E et F tels que $E \cup F = S$. L'ensemble E contient chaque sommet s_i pour lequel on connaît un plus court chemin depuis s_0 (c'est-à-dire pour lequel $d[s_i] = \delta(s_0, s_i)$). L'ensemble F contient tous les autres sommets. À chaque itération de l'algorithme, on choisit le sommet s_i dans F pour lequel la valeur $d[s_i]$ est minimale, on le rajoute dans E, et on relâche tous les arcs partant de ce sommet s_i.

Correction de l'algorithme de Dijkstra : On peut montrer qu'à chaque fois qu'un sommet s_i entre dans l'ensemble E, on a $d[s_i] = \delta(s_0, s_i)$. En effet, le premier sommet à entrer dans l'ensemble E est s_0, pour lequel $d[s_0] = 0 = \delta(s_0, s_0)$. À chaque itération, on fait entrer dans E un sommet $s_i \in F$ tel que $d[s_i]$ soit minimal.

Algorithme 9 : Dijkstra(S, A, ν, s_0, d, Π)

Entrées : S ensemble des sommets, A ensemble des arcs, ν valuations des arcs, s_0 sommet de départ
Sorties : d tableau des bornes maximum des coûts, Π arborescence couvrante

```
pour chaque sommet s_i ∈ S faire
    d[s_i] ← +∞
    Π[s_i] ← nil
d[s_0] ← 0
E ← ∅
F ← S
tant que F ≠ ∅ faire
    soit s_i le sommet de F tel que d[s_i] soit minimal
    /* d[s_i] = δ(s_0, s_i) */
    F ← F − {s_i}
    E ← E ∪ {s_i}
    pour tout sommet s_j ∈ succ(s_i) ∩ F faire
        relacher_arc(s_i, s_j, ν, d, Π)
```

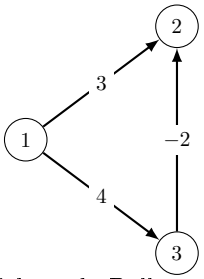
L'idée est que, dans ce cas, s'il existe un autre chemin allant de s_0 jusque s_i, alors il passera nécessairement par un sommet $s_j \in F$ tel que $d[s_j] > d[s_i]$ (puisque $d[s_i]$ est minimal). Sachant que la fin de ce chemin (de s_j à s_i) ne peut faire qu'augmenter la distance du chemin, cet autre chemin sera forcément plus long. Par conséquent, on ne pourra pas trouver de chemin plus court pour aller de s_0 à s_i, et on peut faire entrer s_i dans E, et relâcher tous les arcs qui partent de s_i.

Complexité : La complexité de cet algorithme dépend de l'implémentation du graphe (par matrice ou par listes d'adjacence), mais aussi de la façon de gérer l'ensemble F. On suppose que le graphe possède n sommets et m arcs. Si on utilise une matrice d'adjacence, l'algorithme sera en $O(n^2)$.
En revanche, si on utilise des listes d'adjacence, alors :

- Si F est implémenté par une liste linéaire, ou un tableau, il faudra chercher, à chaque itération, le sommet dans F ayant la plus petite valeur de d. Étant donné qu'il y a n itérations, et qu'au premier passage F contient n éléments, et qu'à chaque passage suivant F contient un élément de moins, il faudra au total faire de l'ordre de $n + (n-1) + (n-2) + ... + 2 + 1$ opérations, soit $O(n^2)$. En revanche, chaque arc étant relâché une seule fois, les opérations de relâchement prendront de l'ordre de m opérations. Au total on aura donc une complexité en $O(n^2)$.
- Pour améliorer les performances de l'algorithme, il faut trouver une structure de données permettant de trouver plus rapidement la plus petite valeur dans l'ensemble F. Pour cela, on peut utiliser un tas binaire : un tas binaire permet de trouver le plus petit élément d'un ensemble en temps constant. En revanche, l'ajout, la suppression ou la modification d'un élément dans un tas binaire comportant n éléments prendra de l'ordre de $log_2(n)$ opérations. Par conséquent, si on implémente F avec un tas binaire, on obtient une complexité pour Dijkstra en $O(m * log(n))$.

3.1.2 Algorithme de Bellman-Ford

L'algorithme de Dijkstra ne marche pas toujours quand le graphe contient des arcs dont les coûts sont négatifs.
Considérons par exemple le graphe suivant, où l'on cherche les plus courts chemins à partir du somemt 1. On voit immédiatement que la plus courte distance entre les sommets 1 et 2 est 2, alors que l'algorithme de Dijkstra donne 3.



			Dist			Préd		
s	E	F	1	2	3	1	2	3
1	$\{1\}$	$\{2,3\}$	0	3	4	0	1	1
2	$\{1,2\}$	$\{3\}$	0	3	4	0	1	1
3	$\{1,2,3\}$	$\{\}$	0	3	4	0	1	1

L'algorithme de Bellman-Ford permet de trouver les plus courts chemins à origine unique dans le cas où le

graphe contient des arcs dont le coût est négatif, sous réserve que le graphe ne contienne pas de circuit absorbant (dans ce cas, l'algorithme de Bellman-Ford va détecter l'existence de circuits absorbants).

L'algorithme de Bellman-Ford fonctionne selon le même principe que celui de Dijkstra : on associe à chaque sommet s_i une valeur $d[s_i]$ qui représente une borne maximale du coût du plus court chemin entre s_0 et s_i. L'algorithme diminue alors progressivement les valeurs $d[s_i]$ en relâchant les arcs. Contrairement à Dijkstra, chaque arc va être relâché plusieurs fois. On relâche une première fois tous les arcs ; après quoi, tous les plus courts chemins de longueur 1, partant de s_0, auront été trouvés. On relâche alors une deuxième fois tous les arcs ; après quoi tous les plus courts chemins de longueur 2, partant de s_0, auront été trouvés... et ainsi de suite... Après la $k$ième série de relâchement des arcs, tous les plus courts chemins de longueur k, partant de s_0, auront été trouvés. Étant donné que le graphe ne comporte pas de circuit absorbant, un plus court chemin est nécessairement élémentaire. Par conséquent, si le graphe comporte n sommets, et s'il ne contient pas de circuit absorbant, un plus court chemin sera de longueur inférieure à n et au bout de $n-1$ passages, on aura trouvé tous les plus courts chemins partant de s_0. (Si le graphe contient un circuit absorbant, au bout de $n-1$ passages, on aura encore au moins un arc (s_i, s_j) pour lequel un relâchement permettrait de diminuer la valeur de $d[s_j]$. L'algorithme utilise cette propriété pour détecter la présence de circuits absorbants.)

Algorithme 10 : Bellman-ford(S, A, ν, s_0, d, Π)

Entrées : S ensemble des sommets, A ensemble des arcs, ν valuations des arcs, s_0 sommet de départ
Sorties : d tableau des bornes maximum des coûts, Π arborescence couvrante

```
1  pour chaque sommet s_i ∈ S faire
2      d[s_i] ← +∞
3      Π[s_i] ← nil
4  d[s_0] ← 0
5  pour k variant de 1 à |S| − 1 faire
6      pour chaque arc (s_i, s_j) ∈ A faire
7          relacher_arc(s_i, s_j, ν, d, Π)
8  pour chaque arc (s_i, s_j) ∈ A faire
9      si d[s_j] > d[s_i] + ν(s_i, s_j) alors
10         afficher("circuit absorbant")
```

Complexité : Si le graphe comporte n sommets et m arcs, chaque arc sera relâché $n-1$ fois, et on effectuera donc au total $(n-1)m$ relâchements successifs. Si le graphe est représenté par une matrice d'adjacence, on aura une complexité en $O(n^3)$, alors que s'il est représenté par des listes d'adjacence, on aura une complexité en $O(nm)$.

Remarque : En pratique, on pourra arrêter l'algorithme dès lors qu'aucune valeur de d n'a été modifiée pendant une itération complète. On pourra aussi mémoriser à chaque itération l'ensemble des sommets pour lesquels la valeur de d a changé, afin de ne relâcher lors de l'itération suivante que les arcs partant de ces sommets.

3.2 Synthèse

En résumé, en fonction des caractéristiques du problème à résoudre il faudra choisir le bon algorithme :

- Si le graphe ne comporte pas de circuit alors, que l'on recherche un plus court chemin ou un plus long chemin, il suffit de trier les sommets topologiquement avec un parcours en profondeur d'abord, puis de considérer chaque sommet dans l'ordre ainsi défini et relâcher à chaque fois tous les arcs partant de ce sommet ;
- Si le graphe comporte des circuits, alors
  - Si on recherche un plus court chemin, alors
    - Si la fonction coût est monotone croissante (le coût d'un chemin ne peut qu'augmenter lorsqu'on rajoute un arc à la fin du chemin ; par exemple, quand les coûts de tous les arcs sont positifs et que le coût d'un chemin est égal à la somme des coûts des arcs empruntés), alors on pourra appliquer Dijkstra ;
    - Sinon, on appliquera Bellman-Ford (on vérifiera en même temps que le graphe ne comporte pas de circuits absorbants) ;
  - Si on recherche un plus long chemin, alors
    - Si la fonction coût est monotone décroissante (le coût d'un chemin ne peut que diminuer lorsqu'on rajoute un arc à la fin du chemin ; par exemple, quand les coûts de tous les arcs sont compris entre

0 et 1 et que le coût d'un chemin est égal au produit des coûts des arcs empruntés), alors on pourra appliquer Dijkstra ;
- Sinon, on appliquera Bellman-Ford (on vérifiera en même temps que le graphe ne comporte pas de circuits absorbants).

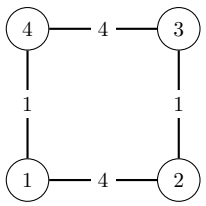
Chapitre 4

Arbres couvrants minimaux (ACM)

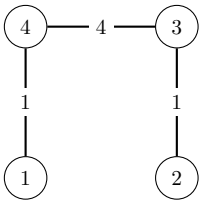
Vous êtes chargés de l'installation du câble dans la région Rhône-Alpes. Vous disposez pour cela d'une carte de l'ensemble du réseau routier (le câble est généralement disposé le long des routes). On vous demande de définir le réseau câblé de telle sorte que la longueur totale de câble soit minimale et qu'un certain nombre de lieux soient desservis.
On peut modéliser ce problème de câblage à l'aide d'un graphe non orienté connexe $G = (S, A)$, où S associe un sommet à chaque lieu devant être desservi, et A contient une arête pour chaque portion de route entre 2 lieux. Ce graphe est valué par une fonction de coûts qui spécifie pour chaque arête $\{s_i, s_j\}$ la longueur de câble nécessaire pour connecter s_i à s_j. Il s'agit alors de trouver un sous-graphe connexe et sans cycle de ce graphe (autrement dit, un arbre) qui recouvre l'ensemble des sommets du graphe. Ce graphe est appelé arbre couvrant. On cherche à minimiser le poids total des arêtes de l'arbre. On dira qu'on cherche l'arbre couvrant minimal (ACM).
De façon plus formelle, un ACM d'un graphe $G = (S, A)$ est un graphe partiel $G' = (S, A')$ de G tel que G' est connexe et sans cycle (G' est un arbre), et la somme des coûts des arêtes de A' est minimale.

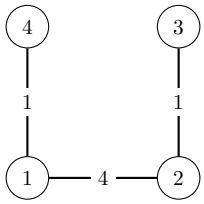
Remarque : il peut exister plusieurs ACM, de même coût, associés à un même graphe.

Par exemple, le graphe suivant :



possède les 2 ACMs suivants :

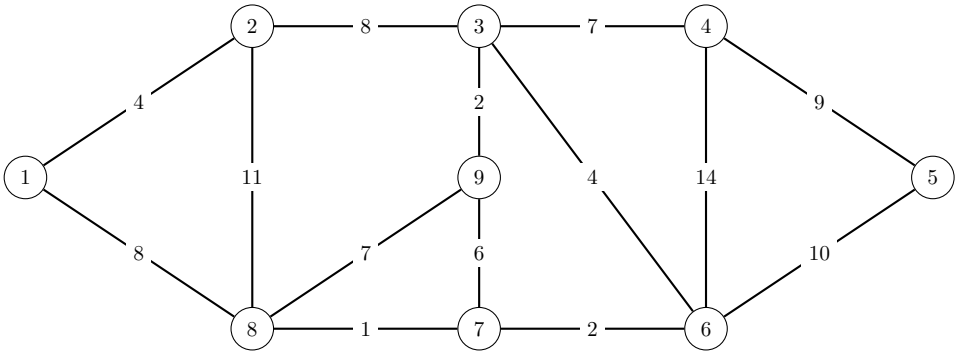




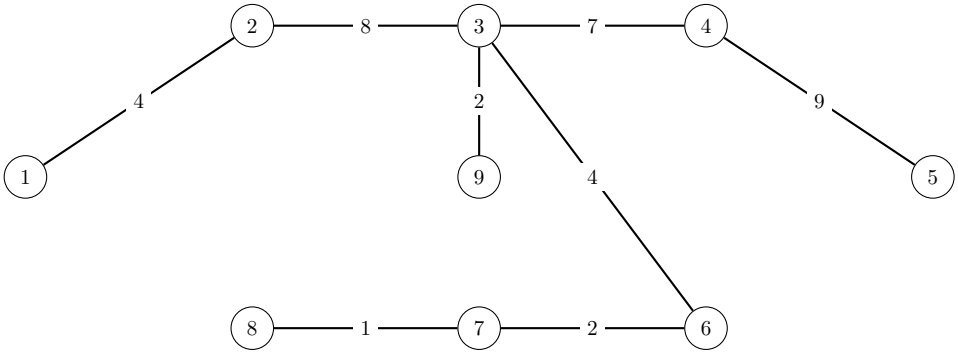
Pour construire un ACM, on adopte une stratégie locale "gloutonne" qui consiste à sélectionner, de pas en pas, une arête devant faire partie de l'ACM. À chaque fois, on choisira la "meilleure" arête selon un certain critère. Il existe deux algorithmes différents suivant une telle stratégie gloutonne et permettant de calculer un ACM à partir d'un graphe : l'algorithme de Kruskal et l'algorithme de Prim. Ces deux algorithmes ont des complexités équivalentes. On étudiera ici l'algorithme de Kruskal.

Principe de l'algorithme de Kruskal : On commence par trier l'ensemble des arêtes du graphe par ordre de coût croissant. On va sélectionner de proche en proche les arêtes devant faire partie de l'ACM. Au début, cet ensemble est vide. On considère ensuite chacune des arêtes du graphe selon l'ordre que l'on vient d'établir (de l'arête de plus faible coût jusqu'à l'arête de plus fort coût). À chaque fois, si l'arête que l'on est en train de considérer peut être ajoutée à l'ensemble des arêtes déjà sélectionnées pour l'ACM sans générer de cycle, alors on la sélectionne, sinon on l'abandonne.

Considérons par exemple le graphe suivant :



On trie les arêtes du graphe. On obtient l'ordre suivant : $\{7,8\} < \{3,9\} = \{6,7\} < \{1,2\} = \{3,6\} < \{7,9\} < \{8,9\} = \{3,4\} < \{2,3\} = \{1,8\} < \{4,5\} < \{5,6\} < \{2,8\} < \{4,6\}$.
On ajoute alors successivement dans l'ACM les arêtes : $\{7,8\}$, $\{3,9\}$, $\{6,7\}$, $\{1,2\}$, $\{3,6\}$, $\{3,4\}$, $\{2,3\}$, $\{4,5\}$.



Mise en œuvre : La difficulté majeure pour implémenter l'algorithme de Kruskal réside dans la façon de déterminer si l'arête en cours d'examen doit ou non être sélectionnée. Il s'agit de savoir si, en rajoutant l'arête $\{s_i, s_j\}$, on crée un cycle ou non, autrement dit, il s'agit de savoir s'il existe déjà une chaîne entre s_i et s_j. Afin de pouvoir répondre à cette question, on va partitionner l'ensemble des sommets du graphe en composantes connexes. Pour savoir si on peut sélectionner une arête $\{s_i, s_j\}$, il suffira de vérifier que s_i et s_j appartiennent à deux composantes connexes différentes. À chaque fois qu'on sélectionnera une arête $\{s_i, s_j\}$, on fusionnera les deux composantes connexes correspondantes en une seule.

Représentation d'une composante connexe : Chaque composante connexe étant un arbre, on choisit de représenter les différentes composantes connexes par un vecteur Π de telle sorte que si $\Pi[s_i] = nil$ alors s_i est la racine d'un arbre, et si $\Pi[s_i] = s_j$, alors s_j est un prédécesseur de s_i dans l'arbre. Ainsi, pour savoir si deux sommets s_i et s_j appartiennent à la même composante connexe, il suffira de remonter dans le vecteur Π de s_i jusqu'à la racine r_i de l'arbre contenant s_i, puis de s_j jusqu'à la racine r_j de l'arbre contenant s_j, et enfin de comparer r_i et r_j : si $r_i = r_j$ alors les deux sommets s_i et s_j appartiennent à la même composante connexe.

Algorithme 11 : Kruskal(S, A, ν, K)

Entrées : S ensemble des sommets, A ensemble des arêtes, ν valuations des arêtes
Sorties : K ensemble des arêtes de l'ACM ($K \subset A$)

```
1  pour chaque sommet s_i ∈ S faire
2      Π[s_i] ← nil
3  trier les arêtes de A par ordre de valeurs de ν croissant
4  K ← ∅
5  tant que |K| < |S| − 1 faire
6      soit {s_i, s_j} la (|K| + 1)ieme plus petite arête de A
7      /* recherche de la racine r_i de la composante connexe de s_i */
8      r_i ← s_i
9      tant que Π[r_i] ≠ nil faire
10         r_i ← Π[r_i]
11     /* recherche de la racine r_j de la composante connexe de s_j */
12     r_j ← s_j
13     tant que Π[r_j] ≠ nil faire
14         r_j ← Π[r_j]
15     si r_i ≠ r_j alors
16         /* on ajoute {s_i, s_j} à l'ACM */
17         K ← K ∪ {{s_i, s_j}}
18         /* on fusionne les deux composantes connexes */
19         Π[r_i] ← r_j
```

Remarque : En pratique, afin d'éviter d'obtenir des arbres déséquilibrés dans Π, lorsqu'on fusionne les deux arbres de racines r_i et r_j, on prendra soin de rattacher l'arbre le moins profond sous l'arbre le plus profond. Pour cela, on gérera un vecteur $prof$ qui associe à chaque racine r sa profondeur $prof[r]$, et si $prof[r_i] > prof[r_j]$ alors on fera $\Pi[r_j] \leftarrow r_i$, sinon on fera $\Pi[r_i] \leftarrow r_j$. En procédant de cette façon, on garantit que le chemin de n'importe quel nœud de l'arbre jusqu'à sa racine est de longueur inférieure ou égale à $log_2(n)$ si le graphe comporte n sommets.

Complexité : On considère un graphe non orienté de n sommets et m arêtes. Pour trier l'ensemble des arêtes, avec une procédure de tri efficace (e.g., quicksort, mergesort ou heapsort), il faut exécuter de l'ordre de $m*log(m)$ opérations. On passe ensuite, dans le pire des cas, m fois dans la boucle "tant que $|K| < |S| - 1$" (une fois pour chaque arête $\{s_i, s_j\}$). À chaque fois, il faut remonter des sommets s_i et s_j jusqu'aux racines des arbres correspondants. En gérant astucieusement la représentation des arbres par Π, on a vu que cette opération pouvait être faite en $O(log(n))$. Par conséquent, on a une complexité totale en $O(m * log(m))$ (sous réserve d'utiliser une représentation par listes d'adjacence).

Chapitre 5

Réseaux de transport

Les réseaux de transport peuvent être utilisés pour modéliser l'écoulement de liquide à l'intérieur de tuyaux, la circulation de pièces dans une chaîne de montage, du courant dans les réseaux électriques, de l'information à travers les réseaux de communication,... D'une façon plus générale, un réseau de transport désigne le fait qu'un "matériau" (de l'eau, de l'électricité, de l'information, ...) doit s'écouler depuis une **source**, où il est produit, jusqu'à un **puits**, où il est consommé. La source produit le matériau à un certain débit, et le puits consomme ce matériau avec le même débit. Entre la source et le puits, ce matériau est transporté par des conduits; chacun de ces conduits a une capacité qui représente la quantité maximale de matériau pouvant transiter par le conduit pendant une unité de temps (par exemple, 200 litres d'eau par heure dans un tuyau, ou 20 ampères de courant électrique à travers un câble).

Les réseaux de transport peuvent être modélisés par des graphes :
- Chaque arc du graphe correspond à un conduit du réseau de transport, par lequel le matériau est acheminé. Chaque arc est valué par la capacité du conduit correspondant.
- Chaque sommet du graphe correspond à une jonction de plusieurs conduits du réseau de transport. Le graphe possède en plus deux sommets particuliers, notés s et p et correspondant respectivement à la source et au puits du réseau de transport.

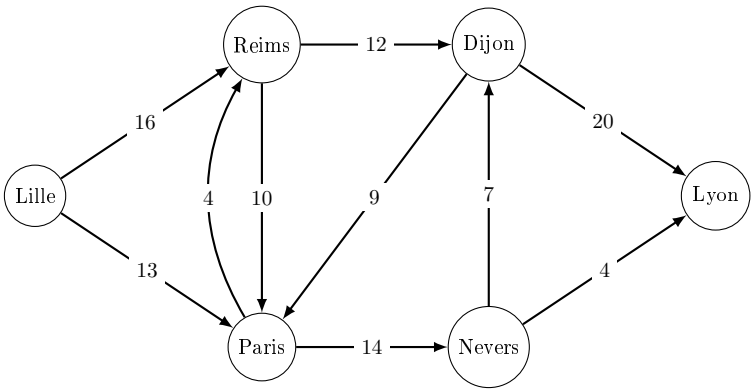
De façon plus formelle, un **réseau de transport** sera défini par un triplet (G, s, p) tel que
- $G = (S, A, c)$ est un graphe orienté valué (c associe à chaque arc sa capacité),
- $s \in S$ est la source, et
- $p \in S$ est le puits.

On suppose qu'il n'y a pas de sommet "inutile", c'est-à-dire que pour tout sommet $s_i \in S$, il existe un chemin de s à p passant par s_i.

Pour des raisons de commodité d'écriture, on supposera que c est définie pour tout couple de sommets s_i, s_j de telle sorte que si (s_i, s_j) n'est pas un arc du réseau, alors $c(s_i, s_j) = 0$.

Exemple : L'usine "Max & Fils", localisée à Lille, produit des voitures. Ces voitures sont acheminées en train jusqu'à Lyon, où elles sont stockées dans un entrepôt puis vendues. Les capacités des trains sont :
- sur la ligne Lille/Reims : 16 voitures par jour,
- sur la ligne Lille/Paris : 13 voitures par jour,
- sur la ligne Paris/Reims : 4 voitures par jour,
- sur la ligne Reims/Paris : 10 voitures par jour,
- sur la ligne Reims/Dijon : 12 voitures par jour,
- sur la ligne Paris/Nevers : 14 voitures par jour,
- sur la ligne Dijon/Paris : 9 voitures par jour,
- sur la ligne Nevers/Dijon : 7 voitures par jour,
- sur la ligne Nevers/Lyon : 4 voitures par jour,
- sur la ligne Dijon/Lyon : 20 voitures par jour.

Ce réseau de transport sera modélisé par le graphe suivant :



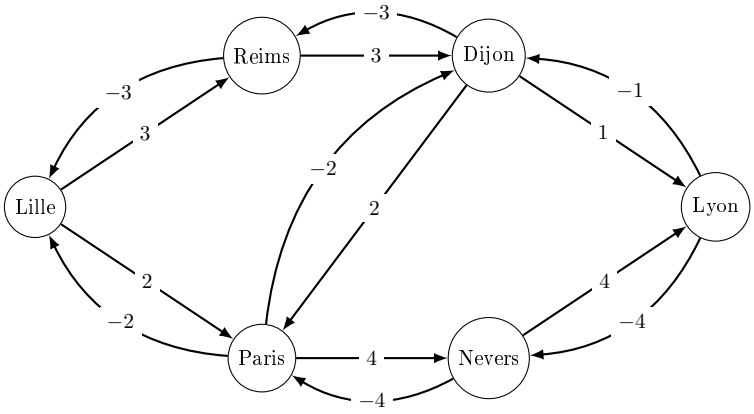
La source est Lille, et le puits Lyon.

On s'intéresse ici au **problème du flot maximal** dans un tel réseau de transport. Il s'agit de déterminer la plus grande quantité de matériau pouvant voyager depuis la source jusqu'au puits, sans violer aucune contrainte de capacité, et tout en préservant la propriété de "conservation de flot" : excepté la source et le puits, le matériau doit s'écouler d'un sommet à l'autre sans perte ni gain. Autrement dit, le débit à l'entrée d'un sommet doit être égal au débit en sortie. De façon plus formelle, un **flot** d'un réseau de transport $(G = (S, A, c), s, p)$ est une fonction $f : S^2 \to \mathbb{R}$ telle que :

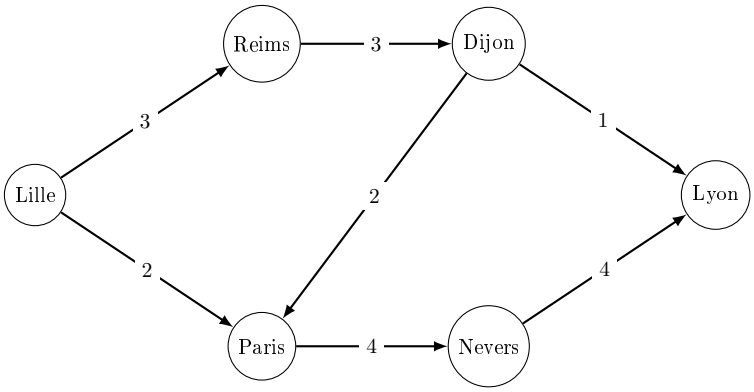
1. contrainte de capacité : $\forall (s_i, s_j) \in S^2, f(s_i, s_j) \leq c(s_i, s_j)$
2. contrainte de symétrie : $\forall (s_i, s_j) \in S^2, f(s_i, s_j) = -f(s_j, s_i)$
3. conservation du flot : $\forall s_i \in S - \{s, p\}, \sum_{s_j \in S} f(s_i, s_j) = 0$

La **valeur d'un flot** f, notée $|f|$, est égale à la somme des flots partant de la source, et du fait de la propriété de conservation des flots, est aussi égale à la somme des flots arrivant au puits :
$|f| = \sum_{s_i \in S} f(s, si) = \sum_{s_i \in S} f(si, p)$.

Exemple : Un flot pour le réseau de transport de l'usine "Max et Fils" est :



Généralement, ce flot sera représenté en ne faisant figurer que les arcs de valeurs positives :



Définition du problème du flot maximal : Étant donné un réseau de transport (G, s, p), il s'agit de trouver un flot f tel que $|f|$ soit maximal.

Modélisation en programmation linéaire : Le problème du flot maximal peut être exprimé comme un problème de programmation linéaire, c'est-à-dire comme une fonction linéaire à maximiser tout en respectant un certain nombre de contraintes linéaires. Étant donné le réseau de transport $(G = (S, A, c), s, p)$, il s'agit de résoudre le problème linéaire consistant à maximiser $\sum_{s_i \in S} f(s, s_i)$ tel que :

- $f(s_i, s_j) \leq c(s_i, s_j), \forall (s_i, s_j) \in A$
- $f(s_i, s_j) = -f(s_j, s_i), \forall \{s_i, s_j\} \in S^2$
- $\sum_{s_j \in S} f(s_i, s_j) = 0, \forall s_i \in S$

Un tel problème linéaire peut être résolu, en utilisant par exemple l'algorithme du simplexe ou l'algorithme du point intérieur. On étudie ici l'algorithme de Ford-Fulkerson permettant de résoudre le problème du flot maximal sans passer par sa modélisation linéaire. L'algorithme procède selon une approche "gloutonne", en augmentant progressivement un flot :

- Au départ, le flot est nul, c'est-à-dire que $f(s_i, s_j) = 0$ pour tout couple de sommets $\{s_i, s_j\} \in S^2$.
- On augmente ensuite itérativement le flot f en cherchant à chaque fois un "chemin améliorant", c'est-à-dire un chemin allant de la source s jusqu'au puits p et ne passant que par des arcs dont le flot actuel est inférieur à leur capacité.
  Pour cela, à chaque itération, on calcule la "capacité résiduelle" de chaque arc, c'est-à-dire la quantité de flot pouvant encore passer.

De façon plus formelle, étant donné un réseau de transport $(G = (S, A, c), s, p)$, et un flot f , on définit :

- la **capacité résiduelle** d'un couple de sommets $(s_i, s_j) \in S^2$, notée $c_f(s_i, s_j)$, est la quantité de flot pouvant encore passer par (s_i, s_j) sans dépasser la capacité : $c_f(s_i, s_j) = c(s_i, s_j) - f(s_i, s_j)$
- le **réseau résiduel** de G, noté $G_f = (S, A_f)$, est le graphe partiel de G ne contenant que les arêtes dont la capacité résiduelle est positive : $A_f = \{(s_i, s_j) \in S^2 / c_f(s_i, s_j) > 0\}$
- un **chemin améliorant** est un chemin sans circuit allant de s à p dans le réseau résiduel G_f
- la **capacité résiduelle d'un chemin améliorant** ch , notée $c_f(ch)$, est la plus grande quantité de flot transportable par les arcs du chemin, et correspond donc à la valeur minimale de la capacité résiduelle des arcs du chemin.

Théorème : Soient

- (G, s, p), un réseau de transport,
- f , un flot de G,
- ch, un chemin améliorant dans le réseau résiduel G_f, et
- f' un flot défini par :
  - $f'(s_i, s_j) = f(s_i, s_j) + c_f(ch)$, si $(s_i, s_j) \in ch$,
  - $f'(s_i, s_j) = f(s_i, s_j) - c_f(ch)$, si $(s_j, s_i) \in ch$
  - $f'(s_i, s_j) = f(s_i, s_j)$ sinon

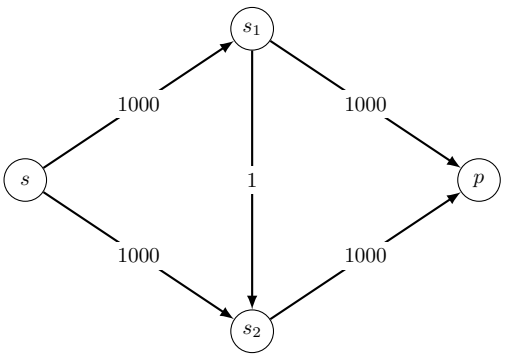
alors, f' est un flot de (G, s, p) tel que $|f'| > |f|$.

Algorithme 12 : Ford-Fulkerson(S, A, c, s, p, f)

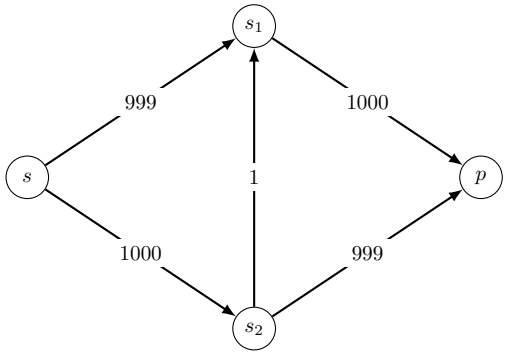
Entrées : S ensemble des sommets, A ensemble des arcs, c valuations des arcs, s sommet source, p sommet puits
Sorties : f flot maximum

```
1  pour chaque (s_i, s_j) ∈ S² faire
2      f(s_i, s_j) ← 0
3      c_f(s_i, s_j) ← c(s_i, s_j)
4  tant que il existe un chemin améliorant ch dans le graphe résiduel faire
5      /* calcul de la capacité résiduelle du chemin améliorant */
6      c_f(ch) ← min_{(s_i,s_j)∈ch} c_f(s_i, s_j)
7      /* mise à jour du flot et de la capacité résiduelle le long des arcs de ch */
8      pour tout arc (s_i, s_j) du chemin améliorant ch faire
9          f(s_i, s_j) ← f(s_i, s_j) + c_f(ch)
10         f(s_j, s_i) ← f(s_j, s_i) − c_f(ch)
11         c_f(s_i, s_j) ← c_f(s_i, s_j) − c_f(ch)
12         c_f(s_j, s_i) ← c_f(s_j, s_i) + c_f(ch)
```

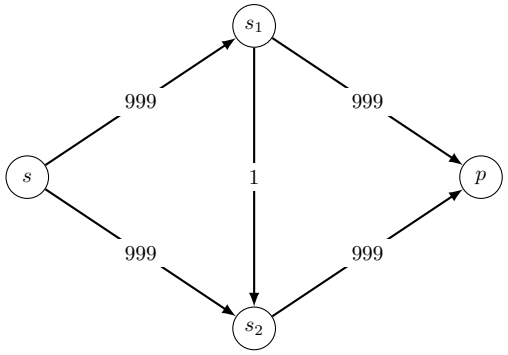
Recherche d'un chemin améliorant dans le réseau résiduel : Il s'agit d'un point critique de l'algorithme, qui peut faire varier considérablement l'efficacité de l'algorithme. Considérons par exemple le réseau de transport suivant :



Au départ, on peut trouver plusieurs chemins améliorants différents pour ce réseau, à savoir $< s, s_1, s_2, p >$, ou $< s, s_1, p >$, ou encore $< s, s_2, p >$. Si l'on choisit le premier chemin ($< s, s_1, s_2, p >$), de capacité résiduelle 1, alors le réseau résiduel devient :



On peut alors trouver un deuxième chemin améliorant $< s, s_2, s_1, p >$, de capacité résiduelle 1, et le réseau résiduel devient :



On peut continuer ainsi, de telle sorte qu'à chaque fois on trouve un chemin améliorant de capacité résiduelle égale à 1. Par conséquent, ce n'est qu'au bout de 2000 étapes successives que l'on trouvera le flot maximal et que l'algorithme s'arrêtera. Cet exemple a été proposé par Edmonds et Karp, qui ont ensuite montré qu'en choisissant à chaque étape le chemin améliorant le plus court (celui qui comporte le moins d'arcs), l'algorithme converge plus rapidement, et nécessite au plus $n * m$ calculs de chemins améliorants successifs (avec n le nombre de sommets et m le nombre d'arcs). Par conséquent, pour chercher le chemin améliorant à chaque étape de l'algorithme de Ford-Fulkerson, il faudra utiliser un parcours en largeur d'abord, permettant de trouver un plus court chemin améliorant (en nombre d'arcs).

Complexité : Si on considère un réseau de transport ayant n sommets et m arcs, l'initialisation (première boucle pour) nécessitera de l'ordre de n^2 opérations. On passera ensuite au plus $n * m$ fois dans la boucle "tant que". À chaque passage dans cette boucle, on effectue un parcours en largeur pour chercher le chemin améliorant, ce qui nécessite de l'ordre de $n + m$ opérations, puis on parcourt les arcs du chemin améliorant trouvé pour calculer la capacité résiduelle du chemin et mettre à jour le flot et la capacité résiduelle du réseau. Le chemin améliorant étant acyclique, il comporte au plus $n-1$ arcs, et donc cette série de traitements nécessite de l'ordre de n opérations. Au total, on fera de l'ordre de $n^2 + (n * m) * (n + m)$ opérations. Étant donné que le réseau est connexe, on a $m \geq n - 1$. Par conséquent, la complexité globale de l'algorithme est en $O(n * m^2)$.