Théorie des graphes et optimisation dans les graphes

Christine Solnon

Table des matières

1 Motivations

Pour résoudre de nombreux problèmes concrets, on est amené à tracer sur le papier des petits dessins qui représentent (partiellement) le problème à résoudre. Bien souvent, ces petits dessins se composent de points et de lignes continues reliant deux à deux certains de ces points. On appellera ces petits dessins des **graphes**, les points des **sommets** et les lignes des **arcs** ou **arêtes**, selon que la relation binaire sous-jacente est orientée ou non.

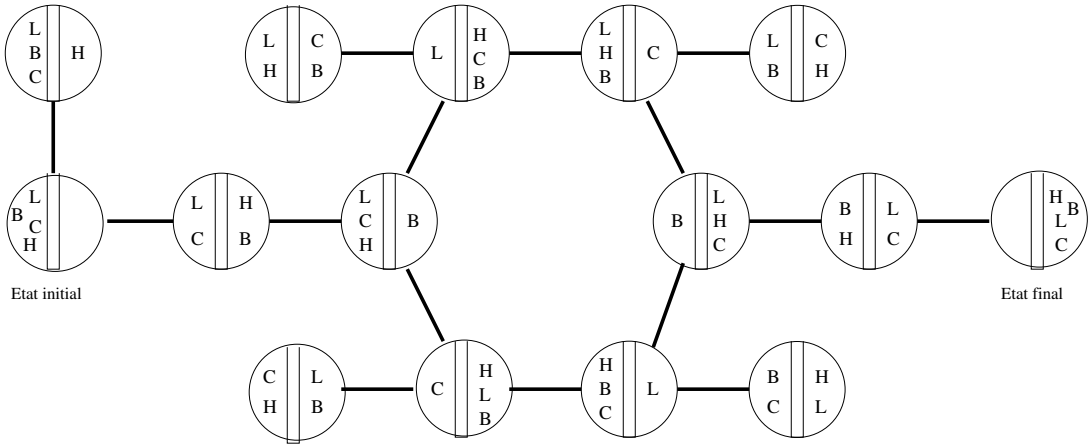
Quelques exemples de modélisation par des graphes

Réseaux routiers : Le réseau routier d'un pays peut être représenté par un graphe dont les sommets sont les villes. Si l'on considère que toutes les routes sont à double sens, on utilisera un graphe non orienté et on reliera par une arête tout couple de sommets correspondant à deux villes reliées par une route (si l'on considère en revanche que certaines routes sont à sens unique, on utilisera un graphe orienté). Ces arêtes pourront être valuées par la longueur des routes correspondantes. Etant donné un tel graphe, on pourra s'intéresser, par exemple, à la résolution des problèmes suivants :
- Quel est le plus court chemin, en nombre de kilomètres, passant par un certain nombre de villes données ?
- Quel est le chemin traversant le moins de villes pour aller d'une ville à une autre ?
- Est-il possible de passer par toutes les villes sans passer deux fois par une même route ?

Processus à étapes : Certains problèmes peuvent être spécifiés par un état initial, un état final, un certain nombre d'états intermédiaires et des règles de transition précisant comment on peut passer d'un état à l'autre. Résoudre le problème consiste alors à trouver une suite de transitions permettant de passer de l'état initial à l'état final. Beaucoup de jeux et autres "casse-tête" peuvent être modélisés ainsi. Considérons, par exemple, le problème du chou, de la brebis et du loup :

> *Un brave homme se trouve au bord d'une rivière qu'il souhaite traverser, en compagnie d'un loup, d'une brebis et d'un chou. Malheureusement, il ne dispose que d'une petite barque, ne pouvant porter en plus de lui-même qu'un seul de ses compagnons (le loup ou la brebis ou le chou). Bien sûr, la brebis refuse de rester seule avec le loup, tandis que le chou refuse de rester seul avec la brebis. Comment peut-il s'y prendre pour traverser la rivière avec ses trois compagnons et continuer son chemin ?*

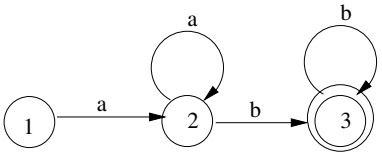
L'état initial est l'état où tout le monde est sur la rive gauche de la rivière, tandis que l'état final est l'état où tout le monde est sur la rive droite de la rivière. La règle de transition est la suivante : si l'homme est sur une rive avec certains de ses compagnons, alors il peut passer sur l'autre rive, soit seul, soit accompagné par un seul de ses compagnons se trouvant sur la même rive que lui, sous réserve qu'il ne laisse pas le loup seul avec la brebis, ou la brebis seule avec le chou. On peut modéliser ce problème par un graphe non orienté, dont les sommets représentent les états possibles, et les arêtes le fait qu'on peut passer d'un état à l'autre par une transition. On obtient alors le graphe non orienté suivant :



où le loup est représenté par la lettre `L`, le chou par `C`, la brebis par `B` et l'homme par `H`, et où un état est représenté par un cercle coupé en deux demi-cercles représentant les rives gauche et droite de la rivière.

Etant donné un tel graphe, on pourra chercher un chemin allant de l'état initial à l'état final.

Automates finis : Un automate fini permet de reconnaître un langage régulier et peut être représenté par un graphe orienté et étiqueté. Par exemple, l'automate fini reconnaissant le langage des mots de la forme $a^n b^m$ (les mots composés d'une suite de 'a' suivie d'une suite de 'b') peut être représenté par le graphe suivant

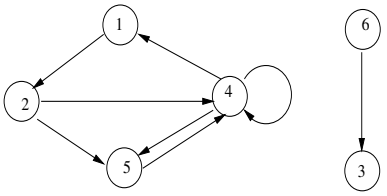


Ce graphe possède 3 sommets et 4 arcs, chaque arc étant étiqueté par un symbole (`a` ou `b`). Etant donné un tel graphe, on peut s'intéresser, par exemple, à la résolution des problèmes suivants :
- Existe-t-il un chemin allant du sommet initial (1) au sommet final (3) ?
- Quel est le plus court chemin entre deux sommets donnés ?
- Existe-t-il des sommets inutiles, par lesquels aucun chemin allant du sommet initial à un sommet final ne peut passer ?

2 Définitions

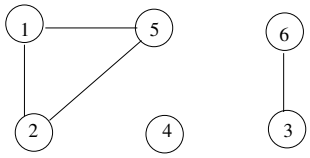
De façon plus formelle, un **graphe** est défini par un couple $G = (S, A)$ tel que
- S est un ensemble fini de sommets,
- A est un ensemble de couples de sommets $(s_i, s_j) \in S^2$.

Un graphe peut être orienté ou non :
– Dans un **graphe orienté**, les couples $(s_i, s_j) \in A$ sont orientés, c'est à dire que (s_i, s_j) est un couple ordonné, où s_i est le sommet initial, et s_j le sommet terminal. Un couple (s_i, s_j) est appelé un **arc**, et est représenté graphiquement par $s_i \to s_j$.
Par exemple,



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

– Dans un **graphe non orienté**, les couples $(s_i, s_j) \in A$ ne sont pas orientés, c'est à dire que (s_i, s_j) est équivalent à (s_j, s_i). Une paire (s_i, s_j) est appelée une **arête**, et est représentée graphiquement par s_i—s_j.
Par exemple,



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

Terminologie

– L'**ordre** d'un graphe est le nombre de ses sommets.
– Une **boucle** est un arc ou 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.
– 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.
– Un **graphe partiel** d'un graphe orienté ou non est le graphe obtenu en supprimant certains arcs ou arêtes.
– Un **sous-graphe** d'un graphe orienté ou non est le graphe obtenu en supprimant certains sommets et tous les arcs ou arêtes incidents aux sommets supprimés.
– Un graphe orienté est dit **élémentaire** s'il ne contient pas de boucle.
– Un graphe orienté 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 $s_i, s_j \in S^2$. De même, un graphe non-orienté est dit complet s'il comporte une arête (s_i, s_j) pour toute paire de sommets différents $s_i, s_j \in S^2$.

Notion d'adjacence entre sommets :

– Dans un graphe non orienté, un sommet s_i est dit **adjacent** à un autre sommet s_j s'il existe une arête entre s_i et s_j. L'ensemble des sommets adjacents à un sommet s_i est défini par :

$$adj(s_i) = \{s_j / (s_i, s_j) \in A \text{ ou } (s_j, s_i) \in A\}$$

– Dans un graphe orienté, on distingue les sommets **successeurs** des sommets **prédécesseurs** :

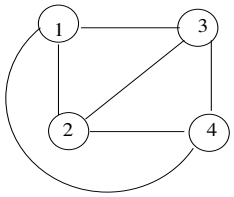
$$\begin{aligned} succ(s_i) &= \{s_j / (s_i, s_j) \in A\} \\ pred(s_i) &= \{s_j / (s_j, s_i) \in A\} \end{aligned}$$

Notion de degré d'un sommet :

- Dans un graphe non orienté, le **degré** d'un sommet est le nombre d'arêtes incidentes à ce sommet (dans le cas d'un graphe simple, on aura $d^{\circ}(s_i) =\mid adj(s_i) \mid$).
- Dans un graphe orienté, le **demi-degré extérieur** d'un sommet s_i, noté $d^{\circ+}(s_i)$, est le nombre d'arcs partant de s_i (dans le cas d'un 1-graphe, on aura $d^{\circ+}(s_i) =\mid succ(s_i) \mid$). De même, le **demi-degré intérieur** d'un sommet s_i, noté $d^{\circ-}(s_i)$, est le nombre d'arcs arrivant à s_i (dans le cas d'un 1-graphe, on aura $d^{\circ-}(s_i) =\mid pred(s_i) \mid$).

Exercice : Dessiner un graphe non orienté complet à 4 sommets. Quel est le degré des sommets de ce graphe ? Combien d'arêtes possède-t-il ? Généralisez ces résultats à un graphe simple complet ayant n sommets.

Correction :



Ce graphe possède 6 arêtes et chaque sommet du graphe est de degré 3.

De façon plus générale, étant donné un graphe simple complet ayant n sommets, chaque sommet étant relié aux $n-1$ autres sommets, le degré de chaque sommet est $n-1$. Le nombre d'arêtes d'un graphe est égal à la moitié de la somme des degrés de tous ses sommets. Par conséquent, un graphe simple complet ayant n sommets aura $n*(n-1)/2$ arêtes.

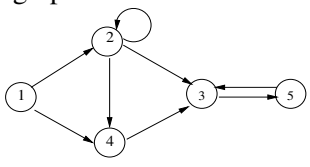
Exercice : On considère le graphe orienté $G = (S, A)$ tel que

$$\begin{aligned} S = &\ \{1,2,3,4,5\} \\ A = &\ \{(1,2),(1,4),(2,2),(2,3),(2,4),(3,5),(4,3),(5,3)\} \end{aligned}$$

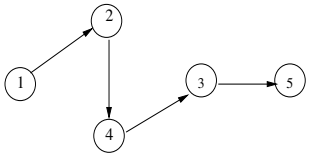
1. représenter graphiquement ce graphe,
2. donner le demi-degré extérieur de 2 et le demi-degré intérieur de 4,
3. donner les sommets prédécesseurs de 4 et les sommets successeurs de 2,
4. donner un graphe partiel et un sous-graphe de ce graphe.

Correction :

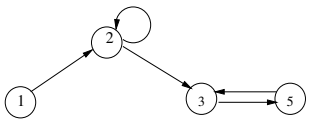
1. Une représentation graphique du graphe est



2. $d^{\circ+}(2) = 3, \quad d^{\circ-}(4) = 2$
3. pred(4) = {1, 2}, succ(2) = {2, 3, 4}
4. Exemple de graphe partiel et de sous-graphe :



Graphe partiel



Sous-graphe induit par 1, 2, 3, 5

Exercice : Au cours d'une soirée, les convives se serrent les mains les uns les autres (jamais plusieurs fois avec la même personne). Chacun se souvient du nombre de mains qu'il a serrées.

1. Montrer qu'il y a au moins 2 personnes ayant serré le même nombre de mains.
2. Montrer que le nombre total de mains serrées est pair.
3. En déduire que le nombre de personnes ayant serré un nombre impair de mains est pair.

Correction : on construit le graphe non orienté $G = (S, A)$, où S associe un sommet à chaque convive, et A associe une arête à chaque couple de convives qui se sont serrés la main. Le nombre de mains serrées par une personne correspond alors au degré du sommet correspondant dans le graphe.

1. Montrer qu'il y a au moins 2 personnes ayant serré le même nombre de mains revient à montrer qu'il y a au moins 2 sommets ayant le même degré : S'il y a n sommets, le degré d'un sommet est compris entre 0 (cas où le sommet est isolé, c'est à dire que la personne correspondante n'a serré la main à personne) et $n - 1$ (cas où le sommet est relié à tous les autres, c'est à dire que la personne correspondante a serré la main à toutes les autres). Pour que tous les sommets aient un degré différent, il faut donc qu'il y ait exactement un sommet de degré 0, un sommet de degré 1, ... etc ... et un sommet de degré $n - 1$. Or, s'il y a un sommet de degré $n - 1$, il ne peut pas y avoir de sommet de degré 0.
2. Montrer que le nombre total de mains serrées est pair revient à montrer que la somme de tous les degrés est paire : chaque arête ajoute 1 au degré de 2 sommets. Par conséquent, la somme des degrés est
$$\sum_{s_i \in S} d^\circ(s_i) = 2 \mid A \mid$$
3. Montrer que le nombre de personnes ayant serré un nombre impair de mains est pair revient à montrer que le nombre de sommets de degré impair est pair : on partitionne l'ensemble S des sommets en l'ensemble S_{pairs} des sommets de degré pair et l'ensemble $S_{impairs}$ des sommets de degré impair. On a
$$\sum_{s_i \in S} d^\circ(s_i) = \sum_{s_j \in S_{pairs}} d^\circ(s_j) + \sum_{s_k \in S_{impairs}} d^\circ(s_k)$$
Etant donné que $\sum_{s_i \in S} d^\circ(s_i)$ est pair, et que $\sum_{s_j \in S_{pairs}} d^\circ(s_j)$ est pair, on en déduit que $\sum_{s_k \in S_{impairs}} d^\circ(s_k)$ doit aussi être pair. Par conséquent, $S_{impairs}$ contient un nombre pair de sommets.

De façon plus générale, on retiendra que, pour tout graphe simple non orienté,
– il existe au moins deux sommets du graphe ayant un même degré ;
– la somme des degrés de tous les sommets du graphe est paire et est égale à deux fois le nombre d'arêtes ;
– il y a un nombre pair de sommets qui ont un degré impair.

3 Représentation des graphes

Il existe deux façons classiques de représenter un graphe en machine : par une matrice d'adjacence ou par un ensemble de listes d'adjacence.

3.1 Représentation par matrice d'adjacence

Soit le graphe $G = (S, A)$. On suppose que les sommets de S sont numérotés de 1 à n, avec $n =\mid S \mid$. 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.

Si le graphe est valué (par exemple, si des distances sont associées aux arcs), on peut utiliser une matrice d'entiers, de telle sorte que $M[i][j]$ soit égal à la valuation de l'arc (i, j) si $(i, j) \in A$. S'il n'existe pas d'arc entre 2 sommets i et j, on peut placer une valeur particulière (par exemple 0 ou $-\infty$ ou $null$) dans $M[i][j]$.

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.

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, le calcul du degré d'un sommet, ou l'accès à tous les successeurs d'un sommet, nécessitent le parcours de toute une ligne (ou toute une colonne) de la matrice, quel que soit le degré du sommet. 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.

3.2 Représentation par listes d'adjacence

Soit le graphe $G = (S, A)$. On suppose que les sommets de S sont numérotés de 1 à n, avec $n =\mid S \mid$. 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 chainée de tous les sommets s_j tels qu'il existe un arc ou une arête $(s_i, s_j) \in A$. Autrement dit, $T[s_i]$ contient la liste de tous les sommets successeurs de s_i. Les sommets de chaque liste d'adjacence sont généralement chainés selon un ordre arbitraire.

Si le graphe est valué (par exemple, si les arêtes représentent des distances), on peut stocker dans les listes d'adjacence, en plus du numéro de sommet, la valuation de l'arête.

Dans le cas de graphes non orientés, pour chaque arête (s_i, s_j), on aura s_j qui appartiendra à la liste chainée de $T[s_i]$, et aussi s_i qui appartiendra à la liste chainée de $T[s_j]$.

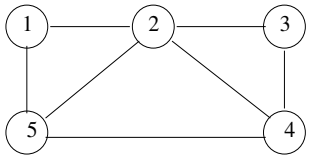
Taille mémoire nécessaire : si le graphe G est orienté, la somme des longueurs des listes d'adjacence est égale au nombre d'arcs de A, puisque l'existence d'un arc (s_i, s_j) se traduit par la présence de s_j dans la liste d'adjacence de $T[s_i]$. En revanche, si le graphe n'est pas orienté, la somme 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 : le test de l'existence d'un arc ou d'une arête (s_i, s_j) avec une représentation par liste d'adjacence est moins direct que dans le cas d'une matrice 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). En revanche, le calcul du degré d'un sommet, ou l'accès à tous les successeurs d'un sommet, est plus efficace que dans le cas d'une matrice d'adjacence : 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 (à comparer avec n^2 dans le cas d'une représentation par matrice d'adjacence).

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 connaitre les prédécesseurs d'un sommet est de maintenir, en plus de la liste d'adjacence des successeurs, la liste d'adjacence des prédécesseurs.

Exercices

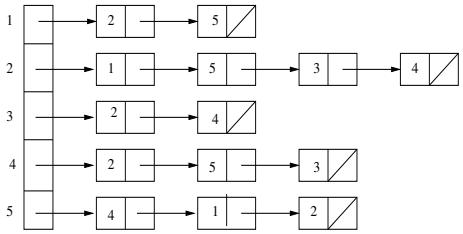
1. Donnez les représentations par matrice d'adjacence et listes d'adjacence du graphe non orienté suivant :



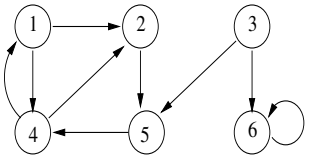
   Correction :
   – Matrice d'adjacence :

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

   – Listes d'adjacence :



2. Donnez les représentations par matrice d'adjacence et listes d'adjacence du graphe orienté suivant :

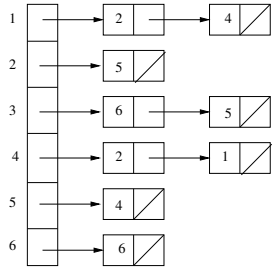


Correction :

– Matrice d'adjacence :

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	1	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

– Listes d'adjacence :



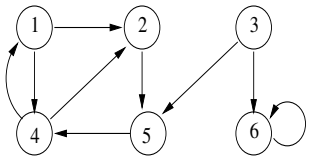
4 Cheminements et connexités

4.1 Notions de chemin, chaine, cycle et circuit

Dans un graphe orienté, un **chemin** d'un sommet u vers un sommet v est une séquence $< s_0, s_1, s_2, ..., s_k >$ de sommets tels que $u = s_0$, $v = s_k$, et $(s_{i-1}, s_i) \in A$ pour $i \in [1..k]$. La **longueur** du chemin est le nombre d'arcs dans le chemin, c'est-à-dire k. On dira que le chemin contient les sommets s_0, s_1, ..., s_k, et les arcs $(s_0, s_1), (s_1, s_2), ..., (s_{k-1}, s_k)$. 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.

Dans un graphe orienté, 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 $s_1, s_2, ..., s_k$ sont tous distincts. Une boucle est un circuit de longueur 1.

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 $< 1, 2, 5, 4, 2, 5, 4, 1 >$.

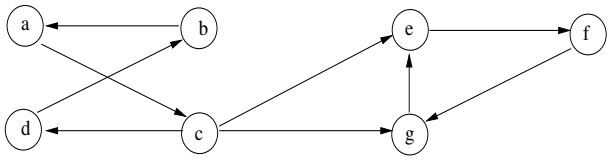
On retrouve ces différentes notions de cheminement dans les graphes non orientés. Dans ce cas, on parlera de **chaine** au lieu de chemin, et de **cycle** au lieu de circuit. Un graphe sans cycle est dit **acyclique**.

Exercice : Montrer que s'il existe un chemin d'un sommet u vers un sommet v dans un graphe orienté, alors il existe un chemin élémentaire de u vers v. idem, pour circuit, chaine et cycle.

4.2 Fermeture transitive d'un graphe

On appelle **fermeture transitive** d'un graphe $G = (S, A)$, le graphe $G^f = (S, A^f)$ tel que pour tout couple de sommets $(s_i, s_j) \in S^2$, l'arc/arête (s_i, s_j) appartient à A^f si et seulement s'il existe un chemin/chaine de s_i vers s_j.

Le calcul de la fermeture transitive d'un graphe peut se faire en additionnant les "puissances" successives de sa matrice d'adjacence. Considérons par exemple le graphe orienté suivant :



La matrice d'adjacence associée à ce graphe est la matrice M suivante :

$$M = \begin{array}{c|ccccccc} & a & b & c & d & e & f & g \\ a & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ b & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ c & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ d & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ e & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ f & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ g & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{array}$$

Dans cette matrice, $M[i][j] = 1$ ssi il existe un chemin de longueur 1 pour aller de i à j. Pour qu'il existe un chemin de longueur 2 pour aller d'un sommet k à un sommet r, il faut qu'il existe un sommet i tel qu'il existe un chemin de longueur 1 de k vers i et un autre chemin de longueur 1 de i vers r. Pour tester cela, il s'agit de parcourir simultanément la ligne k et la colonne r de la matrice M et de regarder s'il y a un 1 à la même position dans la ligne k et la colonne r. Par exemple, il y a un chemin de longueur 2 allant de a vers d car il y a un 1 en troisième position à la fois dans la ligne a et dans la colonne d. Ainsi, en multipliant cette matrice par elle-même, on obtient la matrice M^2 des chemins de longueur 2 :

$$M^2 = \begin{array}{c|ccccccc} & a & b & c & d & e & f & g \\ a & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ b & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ c & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ d & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ e & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ f & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ g & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{array}$$

Dans cette matrice, $M^2[i][j] = 1$ ssi il existe un chemin de longueur 2 pour aller de i à j. Par exemple, $M[a][d] = 1$ car il existe un chemin ($< a, c, d >$) de longueur 2 allant de a à d. De façon plus générale, M^k (la matrice obtenue en multipliant M par elle même k fois successivement) est la matrice des chemins de longueur k.

En additionnant M et M^2, on obtient la matrice $M + M^2$ des chemins de longueur inférieure ou égale à 2 :

$$M + M^2 = \begin{array}{c|ccccccc} & a & b & c & d & e & f & g \\ a & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ b & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ c & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\ d & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ e & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ f & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ g & 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{array}$$

De même, en multipliant $M + M^2$ par M, et en additionnant la matrice résultante avec M, on obtient la matrice $M + M^2 + M^3$ des chemins de longueur inférieure ou égale à 3 :

$$M + M^2 + M^3 = \begin{array}{c|ccccccc} & a & b & c & d & e & f & g \\ a & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ b & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\ c & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\ d & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ e & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ f & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ g & 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{array}$$

Selon le même principe, on calcule la matrice $M + M^2 + M^3 + M^4$ des chemins de longueur inférieure ou égale à 4 :

$$M + M^2 + M^3 + M^4 = \begin{array}{c|ccccccc} & a & b & c & d & e & f & g \\ a & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ b & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ c & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ d & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\ e & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ f & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ g & 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{array}$$

et enfin la matrice $M + M^2 + M^3 + M^4 + M^5$ des chemins de longueur inférieure ou égale à 5 :

$$M + M^2 + M^3 + M^4 + M^5 = \begin{array}{c|ccccccc} & a & b & c & d & e & f & g \\ a & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ b & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ c & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ d & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ e & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ f & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ g & 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{array}$$

Si on recommence une fois de plus, et que l'on calcule la matrice $M+M^2+M^3+M^4+M^5+M^6$ des chemins de longueur inférieure ou égale à 6, on constate que cette matrice est égale à celle des

chemins de longueur inférieure ou égale à 5. Par conséquent $M + M^2 + M^3 + M^4 + M^5$ est la matrice d'adjacence de la fermeture transitive G^f du graphe G de départ.

Notons que pour calculer de cette façon la fermeture transitive d'un graphe comportant n sommets, il faudra faire n multiplications et n additions de matrices (car, dans le pire des cas, le plus long chemin élémentaire entre deux sommets est de longueur n). Chaque multiplication nécessitant de l'ordre de n^3 opérations, cet algorithme a une complexité en $\mathcal{O}(n^4)$. On peut améliorer cet algorithme, et obtenir une complexité en $\mathcal{O}(n^3.log_2(n))$ en multipliant chaque nouvelle matrice calculée par elle-même de la façon suivante :

– En multipliant M par elle-même, et en ajoutant le résultat à M, on obtient l'ensemble des chemins de longueur inférieure ou égale à $2 = M^2 + M$.
– En multipliant $M^2 + M$ par elle-même, et en ajoutant le résultat à M, on obtient l'ensemble des chemins de longueur inférieure ou égale à $4 = M^4 + M^3 + M^2 + M$.
– En multipliant $M^4 + M^3 + M^2 + M$ par elle-même, et en ajoutant le résultat à M, on obtient l'ensemble des chemins de longueur inférieure ou égale à $8 = M^8 + M^7 + M^6 + M^5 + M^4 + M^3 + M^2 + M$.
– etc...

En répétant ce processus k fois de suite, on obtient tous les chemins de longueur inférieure ou égale à 2^k. Donc, pour trouver tous les chemins de longueur inférieure ou égale à n, il faudra répéter ce processus $log_2(n)$ fois.

4.3 Notions de connexité

Cas des graphes non orientés. 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 chaine entre s_i et s_j. 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 défini par les sommets $\{a, b, c, d\}$ est connexe.

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

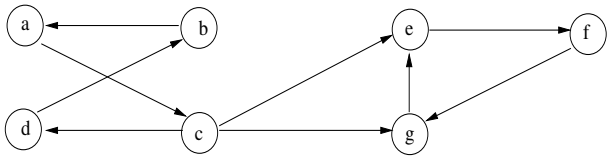
Notons que si l'on calcule la fermeture transitive d'un graphe connexe, on obtient un graphe complet. De même, si l'on calcule la fermeture transitive d'un graphe comportant k composantes connexes, on obtient un graphe contenant k sous-graphes complets (un pour chaque composante connexe). Par exemple, la fermeture transitive du graphe précédent est :

Notons qu'il existe un algorithme plus efficace pour déterminer les composantes connexes d'un graphe non orienté.

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

Plus précisément, un graphe orienté est **fortement 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 un chemin de s_i vers s_j, et un chemin de s_j vers s_i. Par exemple, le graphe de gauche ci-dessous est fortement connexe, tandis que celui de droite ne l'est pas :

Une **composante fortement connexe** d'un graphe orienté G est un sous-graphe G' de G qui est fortement connexe et maximal (c'est à dire qu'aucun autre sous-graphe fortement connexe de G ne contient G'). Par exemple, le graphe orienté suivant



contient 2 composantes fortements connexes : la première est le sous-graphe défini par les sommets $\{a, b, c, d\}$ et la seconde est le sous-graphe défini par les sommets $\{e, f, g\}$.

Comme pour les graphes non orientés, une façon (naïve) de déterminer si un graphe orienté est fortement connexe consiste à calculer sa fermeture transitive : si la fermeture transitive du graphe est le graphe complet, alors il est fortement connexe. Notons qu'il existe un algorithme bien plus efficace pour déterminer les composantes fortement connexes d'un graphe non orienté.

4.4 Notion de graphe eulérien

Dans un graphe non orienté, une **chaine eulérienne** est une chaine qui emprunte une et une seule fois chaque arête du graphe. De même, un **cycle eulérien** est un cycle qui emprunte une et une seule fois chaque arête du graphe. Enfin, un graphe comportant une chaine ou un cycle eulérien est appelé **graphe eulérien**.

Théorème (∃ cycle eulérien) : Un graphe (simple ou multiple) connexe admet un cycle eulérien si et seulement s'il n'a pas de sommet de degré impair.

En effet, supposons qu'un graphe $G = (V, E)$ admette un cycle eulérien. Chaque fois que ce cycle passe par un sommet, il contribue pour 2 dans le degré de ce sommet. Par conséquent, chaque sommet doit avoir un degré pair.

Inversement, soit G un graphe connexe dont tous les sommets sont de degré pair. On montre, par induction sur le nombre d'arêtes du graphe, que ce graphe admet un cycle eulérien. Partant d'un sommet arbitraire a, on peut former de proche en proche un cycle élémentaire retournant sur a. En effet, chaque fois qu'on emprunte une arête, on la retire du graphe. Quand on aboutit en un sommet $x \neq a$, on peut toujours prolonger la chaine car x est de degré pair. Donc, la chaine rencontrera nécessairement le sommet initial a. Soit μ le cycle formé à ce moment là. En vertu de l'hypothèse d'induction, chaque composante connexe de sous-graphe $G_{X-\mu}$ admet un cycle eulérien, et avec le cycle μ ces cycles forment un cycle eulérien de G.

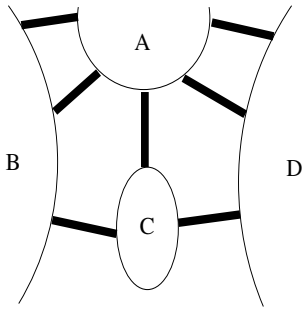
Théorème (∃ chaine eulérienne) : Un graphe (simple ou multiple) connexe admet une chaine eulérienne entre deux sommets u et v si et seulement si le degré de u et le degré de v sont impairs, et les degrés de tous les autres sommets du graphe sont pairs.

On retrouve ces différentes notions sur les graphes orientés : un **chemin eulérien** est un chemin qui emprunte une et une seule fois chaque arc du graphe. De même, un **circuit eulérien** est un circuit qui emprunte une et une seule fois chaque arc du graphe. L'existence d'un circuit eulérien dépend alors des demi-degrés extérieurs et intérieurs des différents sommets : pour chaque sommet il doit y avoir autant d'arcs qui y arrivent que d'arcs qui en partent.

Théorème (∃ circuit eulérien) : Un multigraphe orienté fortement connexe admet un circuit eulérien si et seulement si $d^{\circ+}(s_i) = d^{\circ-}(s_i)$ pour tout sommet $s_i \in S$.

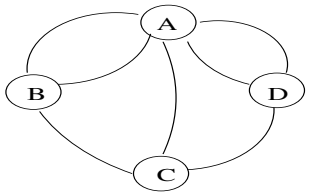
Théorème (∃ chemin eulérienne) : Un multigraphe orienté connexe admet un chemin eulérien de u vers v si et seulement si
- $d^{\circ+}(u) = d^{\circ-}(u) + 1$
- $d^{\circ+}(v) = d^{\circ-}(v) - 1$
- $d^{\circ+}(s_i) = d^{\circ-}(s_i)$ pour tout autre sommet $s_i \in S - \{u, v\}$

Exercice : On considère la disposition des ponts de la ville de Koenigsberg (ville qu'Euler visita lors d'un voyage pour aller en Russie) suivante, où A et C sont deux îles et B et D sont les berges :



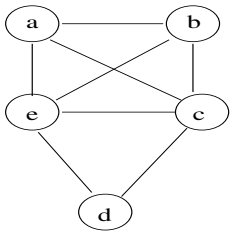
Un piéton peut-il en se promenant traverser chacun des sept ponts de la ville une et une seule fois, et revenir au point de départ ?

Correction : On modélise ce problème sous la forme de la recherche d'un cycle eulérien dans le multi-graphe non orienté suivant :



Dans ce graphe, les degrés des sommets A, B, C et D sont respectivement $5, 3, 3$ et 3. On a donc 4 sommets de degré impair, et il n'y a pas de chaine eulérienne, et encore moins de cycle eulérien, dans ce graphe.

Exercice : Montrer que le graphe suivant est eulérien

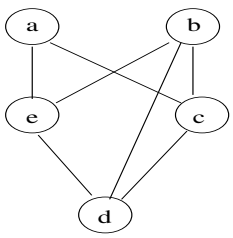


Correction : $d^\circ(d) = 2$, $d^\circ(e) = d^\circ(c) = 4$ et $d^\circ(a) = d^\circ(b) = 3$. Par conséquent, ce graphe admet une chaine eulérienne entre a et b. En revanche, ce graphe n'admet pas de cycle eulérien.

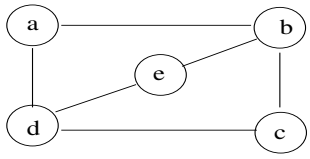
4.5 Notion de graphe hamiltonien

Dans un graphe simple non orienté comportant n sommets, une **chaine hamiltonienne** est une chaine élémentaire de longueur $n - 1$. Autrement dit, une chaine hamiltonienne passe une et une seule fois par chacun des n sommets du graphe. On appelle **cycle hamiltonien** un cycle élémentaire de longueur n. Un graphe possédant un cycle ou une chaine hamiltonien sera dit **graphe hamiltonien**.

Par exemple, le graphe suivant possède un cycle hamiltonien ($< a, e, b, d, c, a >$)



En revanche, le graphe suivant ne possède pas de cycle hamiltonien, mais possède une chaine hamiltonienne ($< a, b, e, d, c >$).



On ne connait aucune condition nécessaire et suffisante d'existance de cycles (chaines, circuits ou chemins) hamiltoniens, valable pour tous les graphes. On peut juste donner des conditions suffisantes, portant en particulier sur les degrés d'un graphe simple.

Remarque : de nombreux problèmes en recherche opérationnelle consistent à chercher un chemin ou un cycle hamiltonien dans un graphe. Le plus connu est probablement le problème du voyageur de commerce, qui doit trouver un itinéraire "optimal" passant par chaque ville de son réseau commercial. On considère dans ce cas le graphe non orienté valué représentant une carte routière : les sommets correspondent aux villes, les arêtes aux routes, et les valuations aux distances. Il s'agit alors de trouver un cycle hamiltonien de valuation minimale. Ce problème est un des premiers à avoir été montré comme étant NP-complet (ce qui implique que l'on ne connait aucun algorithme "efficace" pour résoudre ce problème).

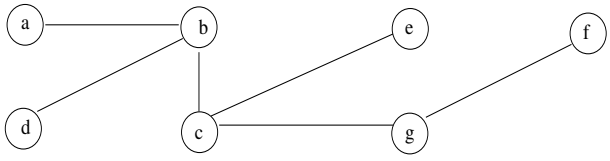
5 Arbres et arborescences

Les arbres et les arborescences sont des graphes particuliers très souvent utilisés en informatique pour représenter des données.

Etant donné un graphe non orienté comportant n sommets, les propriétés suivantes sont équivalentes pour caractériser un **arbre** :

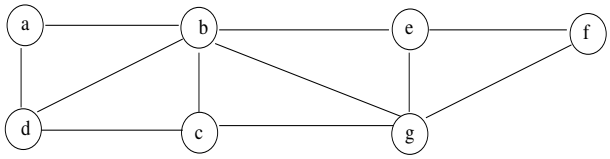
1. G est connexe et sans cycle,
2. G est sans cycle et possède $n-1$ arêtes,
3. G est connexe et admet $n-1$ arêtes,
4. G est sans cycle, et en ajoutant une arête, on crée un et un seul cycle élémentaire,
5. G est connexe, et en supprimant une arête quelconque, il n'est plus connexe,
6. Il existe une chaine 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.

Exercice : On considère le graphe non orienté suivant :

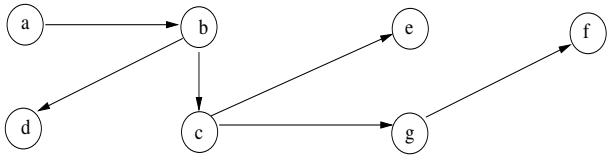


Combien faut-il enlever d'arêtes à ce graphe pour le transformer en arbre ? Donnez un graphe partiel de ce graphe qui soit un arbre.

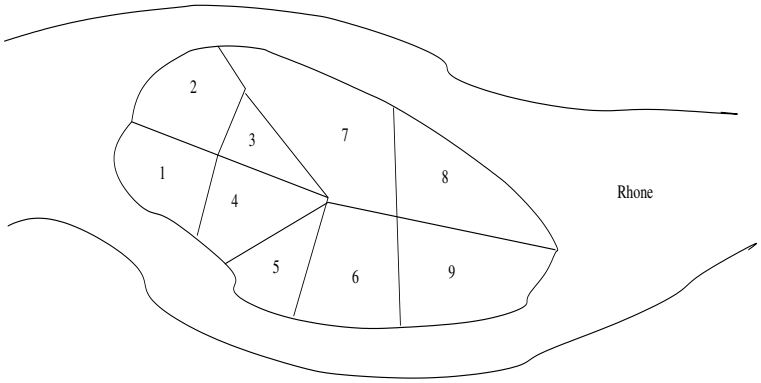
Correction : le graphe comporte 7 sommets et 11 arêtes. Pour le transformer en arbre il faudra donc enlever 5 arêtes. Par exemple, les arêtes (f, g), (b, g), (b, c), (b, d) et (a, d).

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` :

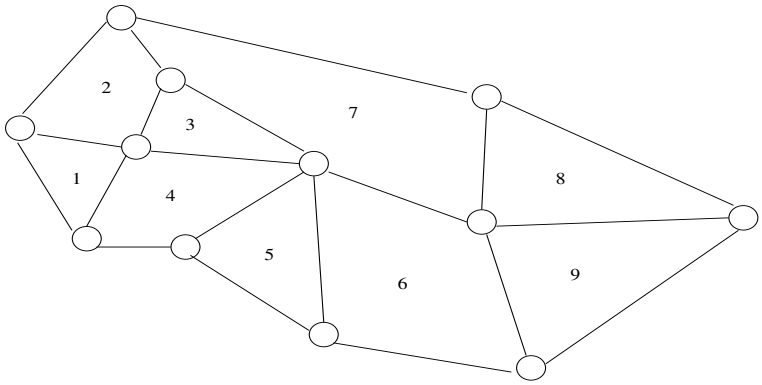


Exercice : L'ile du Nivéou, en Camargue, se consacre à la culture du riz. Sur cette ile se trouvent 9 champs entourés de murs et disposés de la façon suivante :

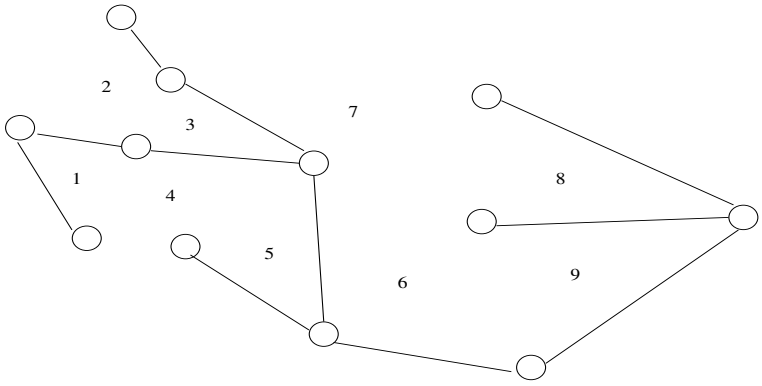


La culture du riz suppose que l'on puisse périodiquement inonder l'ensemble des champs. Cela est réalisé en ouvrant des vannes placées dans les murs séparant les champs et le Rhône ou les champs entre eux. Etant donné que l'installation d'une vanne est coûteuse, il s'agit de déterminer le nombre minimum de vannes et leur emplacement pour pouvoir, quand on le désire, inonder tous les champs.

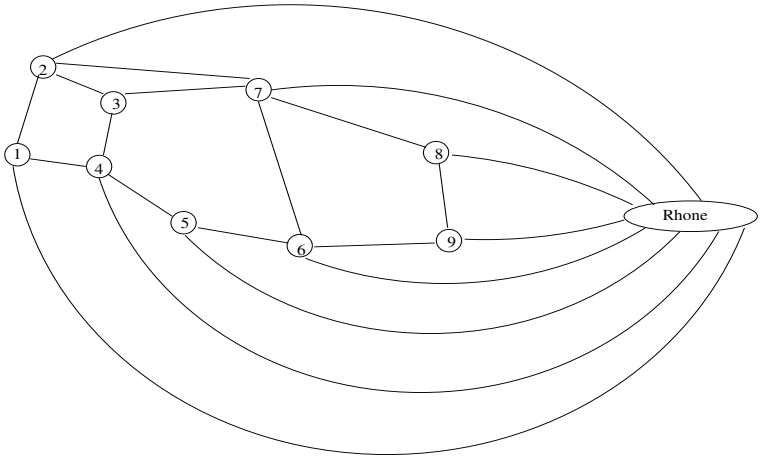
Correction : Pour résoudre ce problème, on peut considérer le graphe non orienté comportant un sommet pour chaque intersection de mur, et une arête pour chaque mur :



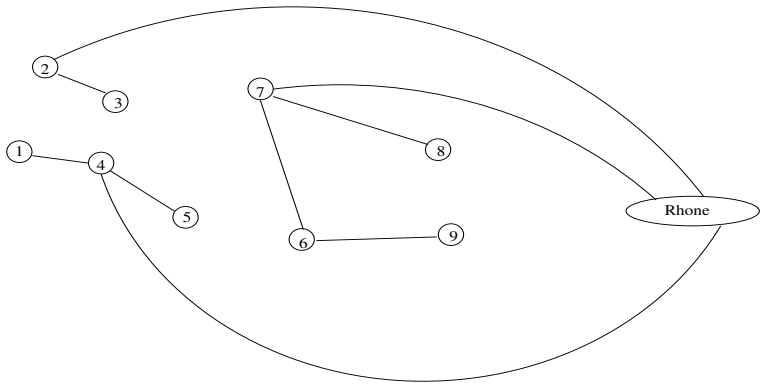
En considérant que lorsqu'on place une vanne sur un mur, on supprime l'arête correspondante dans le graphe, le problème revient à supprimer des arêtes jusqu'à ce que le graphe ne comporte plus de cycles (autrement dit, placer des vannes jusqu'à ce qu'il n'y ait plus de champ entouré de murs sans vanne). Comme on souhaite poser le moins de vannes possible, on cherche un graphe partiel sans cycle tel que si l'on rajoute une arête on crée un cycle : selon la proposition 4, il s'agit d'un arbre. Ici, étant donné que le graphe a 12 sommets et 20 arêtes, l'arbre devra posséder 12-1 = 11 arêtes (selon la proposition 2), et on devra donc installer 20-11=9 vannes. On obtiendra (par exemple) l'arbre suivant :



Autre modélisation possible : Pour résoudre ce problème, on peut également considérer le graphe non orienté comportant un sommet pour chaque champ plus un sommet représentant le Rhône. Ce graphe comporte une arête entre deux sommets si les champs correspondants, ou le Rhône, sont voisins. On obtient alors le graphe suivant :



En considérant que lorsqu'on place une vanne sur un mur séparant deux champs (ou une vanne séparant un champ du Rhône) on conserve l'arête joignant les deux sommets correspondant aux champs (ou au champ et au Rhône), le problème revient à chercher un graphe partiel connexe (autrement dit, tous les champs doivent être reliés par un "chemin de vannes" au Rhône). Comme on souhaite poser le moins de vannes possible, il s'agit de garder le moins d'arêtes possible. On cherche donc un graphe partiel connexe tel que si l'on supprime une arête de plus il ne soit plus connexe : selon la proposition 5, il s'agit d'un arbre. Ici, étant donné que le graphe possède 10 sommets, l'arbre devra comporter 10-1=9 arêtes (selon la proposition 3). Là encore, on devra installer 9 vannes. On obtiendra (par exemple) l'arbre suivant :



Cet exemple permet d'introduire (très succintement) la notion de graphe dual d'un graphe planaire, notion développée dans la section suivante : les deux modélisations utilisées pour résoudre le problème de l'ile du Nivéou sont duales. L'intérêt de cette notion de dualité est de montrer qu'à tout graphe planaire on peut toujours associer un autre graphe représentant, d'une façon différente, la même chose. Le meilleur modèle du problème étant le plus commode de ces deux graphes, on aura toujours intérêt, lorsqu'un problème peut être modélisé par un graphe planaire, à étudier si le graphe dual ne permet pas une formalisation plus simple du problème.

6 Graphes planaires

Trois chatelains voisins sont ennemis entre eux. Pour se mettre au goût du jour, ils ont décidé d'installer l'eau, le gaz et l'électricité dans leurs chateaux respectifs. Naturellement, ils ne veulent pas que leurs fils et leurs canalisations se croisent ! Il s'agit de trouver le moyen de relier chacun des chatelains aux compagnies d'eau, de gaz et d'électricité sans provoquer de disputes entre eux.

On peut modéliser ce problème par un graphe ayant les 6 sommets suivants :

```
Chateau 1          Chateau 2          Chateau 3
    x                  x                  x




    x                  x                  x
   eau                gaz            electricite
```

Il s'agit alors de dessiner pour chaque chateau 3 arêtes le reliant à l'eau, au gaz et à l'électricité, sans que deux arêtes ne se croisent. Ce problème n'a en fait pas de solutions (même en changeant la disposition des sommets). On dira que le graphe correspondant n'est pas planaire.

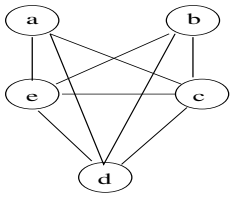
De façon plus formelle, on appelle **graphe planaire** tout graphe non orienté pouvant être dessiné sur un plan de telle sorte que les sommets soient des points distincts, et que les arêtes ne se rencontrent pas en dehors de leurs extrémités (les arêtes pouvant être représentées par des courbes).

Si on considère par exemple une carte de géographie, et si on associe un sommet à chaque pays et une arête entre deux sommets si les pays correspondants ont une frontière commune (non réduite à un seul point), alors on obtient un graphe planaire.

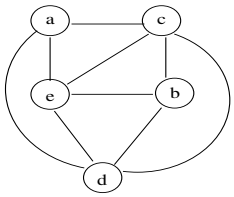
Exercice : On appelle K_i le graphe complet comportant i sommets. Parmi K_2, K_3, K_4 et K_5, lesquels sont planaires ?

Correction : K_2, K_3 et K_4 sont planaires ; en revanche K_5 ne l'est pas.

Exercice : Montrer que le graphe suivant est planaire :

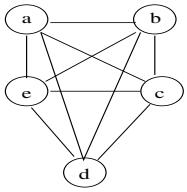


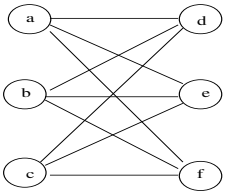
Correction : on peut le représenter de la façon suivante.



Remarque : l'adjectif "planaire" qualifie un graphe, indépendemment de sa représentation graphique choisie. Autrement dit, un même graphe peut être dessiné de différentes façons (comme dans l'exercice précédent). Certaines de ces représentations peuvent ne pas être planaires, alors que d'autres peuvent l'être. Il suffit, pour un graphe donné, de trouver une représentation planaire pour en conclure que le graphe planaire.

Caractérisation des graphes planaires : Les deux graphes non planaires les plus simples sont K_5 et $K_{3,3}$:

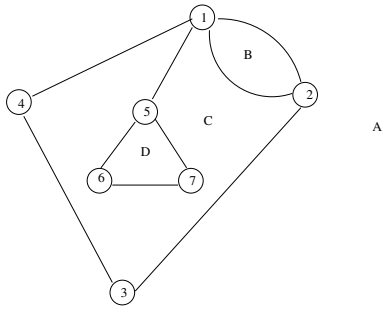




Théorème de Kuratowski : un graphe est planaire si et seulement s'il ne possède aucun mineur isomorphe à K_5 ou $K_{3,3}$. (Un mineur d'un graphe est obtenu par une succession d'opérations de suppression et de "fusion" de sommets). Autrement dit, tout graphe non planaire contient une "copie" d'au moins un de ces deux graphes.

Faces d'un graphe planaire : Etant donnée une représentation planaire d'un graphe G, le plan se retrouve divisé en un certain nombre de régions qu'on appelle les **faces de la représentation planaire**.

Par exemple, le graphe suivant



possède 4 faces (notées A, B, C et D). On dira que les arêtes $(1,2), (1,4), (4,3), (3,2), (5,6)$ et $(5,7)$ constituent des **frontières** entre des faces différentes, tandis que l'arête $(5,1)$ constitue un **isthme**.

Remarque : Le nombre de faces d'un graphe planaire connexe est indépendant de la représentation planaire choisie.

Formule d'Euler : Soit G un graphe planaire connexe possédant s sommets, a arêtes et f faces, on a $f + s = a + 2$.

Exercice : Vérifiez la formule d'Euler dans le cas d'un arbre.

Correction : un arbre a une seule face, car il n'a pas de cycle. Par ailleurs, le nombre d'arêtes d'un arbre est égal au nombre de sommets - 1. La formule d'Euler est donc bien vérifiée.

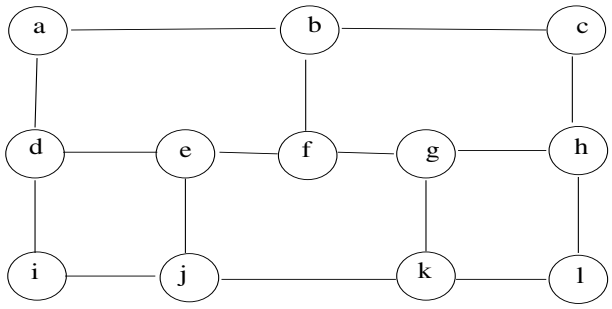
Exercice : Montrez, en utilisant la formule d'euler que K_5 n'est pas planaire.

Correction : en effet, K_5 possède 10 arêtes et 5 sommets. D'après la formule d'Euler, si K_5 est planaire, alors il a $10 - 5 + 2 = 7$ faces. Or chaque face de K_5 est bordée par au moins 3 arêtes (car K_5 ne possède ni boucle, ni arête multiple), et chaque arête borde au plus 2 faces (si c'est une arête frontière, elle borde 2 faces, si c'est un isthme, elle borde une seule face). Par conséquent, le nombre de faces doit être inférieur ou égal à (2/3)*le nombre d'arêtes, autrement dit $f \leq (2/3)*10$ et on a une contradiction avec la formule d'Euler qui nous dit qu'on doit avoir 7 faces.

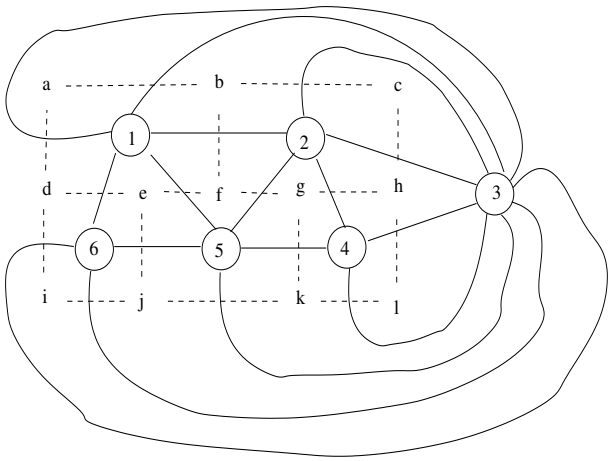
Graphe Dual : On appelle **dual** d'un graphe planaire —appelé **primal**— le graphe obtenu de la façon suivante :

- dans toute face du primal on dessine un sommet du dual,
- pour toute arête séparant deux faces du primal, on dessine une arête joignant les deux sommets correspondants du dual (et qui traverse l'arête correspondante du primal).
Remarquons que cette relation est symétrique : si G_2 est le dual de G_1, alors G_1 est le dual de G_2.

Exercice : Est-il possible de dessiner sans lever la main un lacet qui traverse chaque arête de ce graphe planaire une et une seule fois ?



Correction : On considère le graphe dual qui associe un sommet à chaque face, et une arête à chaque arête du primal de telle sorte qu'une arête dans le dual modélise le fait que l'on a traversé l'arête correspondante du graphe primal.



Traverser l'ensemble des arêtes du graphe primal revient alors à trouver une chaine qui passe une et une seule fois par chaque arête du dual, c'est-à-dire, une chaine eulérienne. Etant donné qu'il y a 4 sommets de degré impair dans le graphe dual, il ne peut y avoir de chaine eulérienne, et le problème n'a pas de solution.

7 Coloriage de graphes, cliques et stables

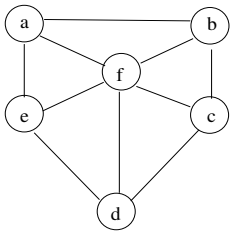
Le problème du coloriage de graphes, pour un graphe non orienté G, consiste à attribuer une couleur à chaque sommet, de telle sorte qu'une même couleur ne soit pas attribuée à deux sommets adjacents (reliés par une arête). Lorsque le graphe est planaire, ce problème peut s'exprimer par rapport au graphe dual, en coloriant les faces, et non les sommets, et en imposant que deux faces voisines soient coloriées de couleurs différentes (il s'agit du problème de coloriage d'une carte).

Le nombre minimum de couleurs nécessaires pour colorier un graphe G est appelé le **nombre chromatique** de G, et noté $X(G)$.

Remarque : le problème du coloriage d'un graphe avec un nombre limité de couleurs est un problème combinatoire (on dit qu'il s'agit d'un problème NP-complet). Cela signifie que tout algorithme résolvant ce problème de façon exacte pourra prendre un temps exponentiel par rapport au nombre de sommets du graphe (de l'ordre de 2^n pour n sommets). Le problème de déterminer le nombre chromatique d'un graphe est également exponentiel, et est en fait encore plus difficile.

L'algorithme de Brélaz (également appelé DSATUR) est un algorithme glouton qui permet de calculer une borne maximale de $X(G)$. Cet algorithme a une complexité polynomiale et procède de façon itérative. Plus précisément, tant qu'il existe un sommet non colorié, il choisit à chaque itération le sommet non colorié ayant le plus grand nombre de voisins coloriés avec des couleurs différentes, les *ex aequo* étant départagés en choisissant le sommet de plus fort degré. Le sommet choisi est alors colorié par la plus petite couleur possible (en partant du principe que les couleurs sont triées selon un ordre donné) ; si toutes les couleurs existantes sont utilisées par au moins un voisin du sommet à colorier, alors une nouvelle couleur est ajoutée à l'ensemble des couleurs disponibles.

Exercice : Déterminez le nombre chromatique du graphe suivant :



Conjecture des 4 couleurs : Le nombre chromatique d'un graphe planaire est inférieur ou égal à 4.

Cette conjecture a passionné de nombreux mathématiciens, pendant plus d'un siècle. Elle a été démontrée en 1976.

Une application de cette conjecture est que l'on peut colorier les pays d'une carte géographique avec seulement 4 couleurs de telle sorte que deux pays voisins soient coloriés avec des couleurs différentes.

Exercice : 5 étudiants (Dupont, Dupond, Durand, Duval et Duduche) doivent passer certains examens. Les examens que doivent passer chaque étudiant sont récapitulés dans le tableau suivant :

Dupont	Français, Anglais, Mécanique
Dupond	Dessin, Couture
Durand	Anglais, Solfège
Duval	Dessin, Couture, Mécanique
Duduche	Dessin, Solfège

On désire que tous les étudiants devant subir une même épreuve le fassent en même temps. Chaque étudiant ne peut se présenter qu'à une épreuve au plus par jour. Quel est le nombre minimal de jours nécessaires à l'organisation de toutes les épreuves ?

Correction : On modélise ce problème par le graphe non orienté $G = (S, A)$ tel que S associe un sommet à chaque épreuve et A associe une arête entre deux sommets si un même étudiant doit subir les épreuves correspondantes. Il s'agit alors de trouver le nombre chromatique du graphe, c'est-à-dire 3.

Cliques : Etant donné un graphe non orienté $G = (S, A)$, une clique est un sous-ensemble de sommets $S' \subseteq S$ qui sont tous connectés 2 à 2 par des arêtes de sorte que

$$\forall (i, j) \in S' \times S', i \neq j \Rightarrow (i, j) \in A$$

Dit autrement, une clique est un sous-graphe complet.

Trouver une clique d'ordre k dans un graphe est également un problème NP-complet, ce qui implique là encore que tout algorithme résolvant ce problème de façon exacte aura une complexité exponentielle. Le problème de trouver la plus grande clique d'un graphe est appelé "problème de la clique maximum", et est encore plus difficile...

Exercice : Montrer que pour tout graphe G, l'ordre de la clique maximum est inférieur ou égal à $X(G)$.

Stables : Etant donné un graphe non orienté $G = (S, A)$, un stable est un sous-ensemble de sommets $S' \subseteq S$ non connectés par des arêtes de sorte que

$$\forall (i, j) \in S' \times S', i \neq j \Rightarrow (i, j) \notin A$$

Le problème de rechercher un stable d'ordre k dans un graphe $G = (S, A)$ est équivalent au problème de rechercher une clique d'ordre k dans le graphe inverse $G' = (S, A')$ tel que A' contient une arête entre 2 sommets i et j ssi A ne contient pas d'arête entre ces 2 sommets.

8 Parcours de graphes

Beaucoup de problèmes sur les graphes nécessitent que l'on parcourt l'ensemble des sommets et des arcs/arêtes du graphe. On étudie dans la suite 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 (jusqu'à un cul-de-sac ou un cycle), 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 blancs. On dira qu'un sommet blanc n'a pas encore été découvert.
- 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.

8.1 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 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 si 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.

8.2 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, $d[s_i]$ est la longueur du chemin dans l'arborescence π de la racine s_0 jusque s_i). Ce tableau sera utilisé plus tard, cf. 7.3.

`Algorithme de parcours en largeur (BFS)`(S, A, s_0)
init_file(F)

```
    pour tout sommet s_i ∈ S faire
        π[s_i] ← nil
        d[s_i] ← ∞
        couleur[s_i] ← blanc
    fin pour
    d[s_0] ← 0
    ajoute_fin_file(F, s_0)
    couleur[s_0] ← gris
    tant que est_vide(F) = faux faire
        enleve_debut_file(F, s_i)
        pour tout s_j ∈ succ(s_i) faire
            si couleur[s_j] = blanc alors
                ajoute_fin_file(F, s_j)
                couleur[s_j] ← gris
                π[s_j] ← s_i
                d[s_j] ← d[s_i] + 1
            finsi
        fin pour
        couleur[s_i] ← noir
    fin tant
fin BFS
```

Complexité : Chaque sommet (accessible depuis s_0) est mis, puis enlevé, une fois dans la file. A 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
- $\mathcal{O}(n^2)$ dans le cas d'une implémentation par matrice d'adjacence,
- $\mathcal{O}(n+p)$ dans le cas d'une implémentation par listes d'adjacence.

8.3 Applications du parcours en largeur

Le parcours en largeur peut être utilisé pour **rechercher l'ensemble des sommets accessibles depuis** s_0. A la fin de l'exécution de $BFS(S, A, s_0)$, chaque sommet est soit noir soit blanc. Les sommets noirs sont ceux accessibles depuis s_0 ; les sommets blancs sont ceux pour lesquels il n'existe pas de chemin/chaine 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. A 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 (récursif) est le suivant :

```
Algorithme plus_court_chemin(s_0, s_j, π)
```

/ affiche le plus court chemin pour aller de s_0 a s_j */*
si $s_0 = s_j$ alors afficher(s_0)
sinon si $\pi[s_j] = nil$ alors afficher("pas de chemin")
sinon plus_court_chemin($s_0, \pi[s_j], \pi$)
afficher(s_j)
finsi
fin plus court chemin

8.4 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*.

Algorithme de parcours en profondeur des sommets accessibles depuis s_0 DFS
init_pile(P)
pour tout sommet $s_i \in S$ faire
$\pi[s_i] \leftarrow nil$
couleur$[s_i] \leftarrow$ blanc
fin pour
$tps \leftarrow 0$
$dec[s_0] \leftarrow tps$
empile(P, s_0)
couleur$[s_0] \leftarrow$ gris
tant que *est_vide*(P) = faux faire
$tps \leftarrow tps + 1$
$s_i \leftarrow sommet(P)$
si $\exists s_j \in succ(s_i)$ tel que *couleur*$[s_j]$=blanc alors
empile(P, s_j)
couleur$[s_j] \leftarrow$ gris
$\pi[s_j] \leftarrow s_i$
$dec[s_j] \leftarrow tps$
sinon /* tous les successeurs de s_i sont gris ou noirs */
depile(P, s_i)
couleur$[s_i] \leftarrow$ noir

$fin[s_i] \leftarrow tps$
finsi
fin tant
fin DFS

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

Cet algorithme peut s'écrire récursivement sans utiliser de pile explicite de la façon suivante :

Algorithme récursif de parcours en profondeur des sommets accessibles depu
DFSrec(S, A, s_0)
$dec[s_0] \leftarrow tps$
$tps \leftarrow tps + 1$
couleur$[s_0] \leftarrow$ gris
pour tout $s_j \in succ(s_0)$ faire
si *couleur*$[s_j]$=blanc alors
$\pi[s_j] \leftarrow s_0$
DFSrec(S, A, s_j)
finsi
finfaire
couleur$[s_0] \leftarrow$ noir
$fin[s_0] \leftarrow tps$
$tps \leftarrow tps + 1$
fin DFSrec

Dans ce cas, les structures de données π, couleur, dec et tps sont des variables globales et il est nécessaire de les initialiser auparavant.

Algorithme d'initialisation avant l'appel de DFSrec initDFSrec(S, A)
pour tout sommet $s_i \in S$ faire
$\pi[s_i] \leftarrow nil$
couleur$[s_i] \leftarrow$ blanc
fin pour
$tps \leftarrow 0$
fin

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

8.5 Applications du parcours en profondeur

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 global de parcours en profondeur DFSglobal(S, A)

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

Recherche de circuits

Lors du parcours en profondeur d'un graphe non orienté (resp. orienté), si un successeur s_j du sommet "courant" s_i au sommet de la pile est déja gris, cela implique qu'il existe une chaine (resp. un chemin) permettant d'aller de s_j vers s_i, et donc qu'il existe un cycle (resp. un circuit). Ainsi, un algorithme pour détecter les cycles/circuits d'un graphe peut être obtenu en rajoutant dans l'algorithme DFSrec l'instruction

si couleur[s_j]=gris alors afficher("existence d'un cycle")

juste après la boucle "pour tout $s_j \in succ(s_0)$ faire".

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 déja noir, cela implique qu'il existe déjà un chemin permettant d'aller d'un ancêtre de s_i vers s_j.

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 : 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 évènements : les sommets représentent les évènements, et les arcs les relations de précédence. Dans ce cas, un tri topologique permet de trier les évènements de telle sorte qu'un évènement n'apparait qu'après tous les évènements qui doivent le précéder. Le problème général de la planification sera étudié plus en détail au chapitre 5.

Recherche des composantes fortement connexes d'un graphe orienté

Un algorithme pour rechercher les composantes fortement connexes d'un graphe orienté est donné ci-dessous. Une preuve de la correction de cet algorithme pourra être trouvée dans l'ouvrage "Introduction à l'algorithmique" référencé au chapitre 12.

```
Algorithme de recherche des composantes fortement connexes(S, A)
    DFSglobal(S, A)
    inverser les sens de tous les arcs du graphe
    trier les sommets par ordre de numéro de fin décroissant dans un tableau t
    initDFSrec(S,A)
    nbcfc ← 0
    tant que ∃s_i ∈ S tel que couleur[s_i] = blanc faire
        soit s_j le prochain sommet blanc dans le tableau t
        DFSrec(S, A, s_j)
        nbcfc ← nbcfc + 1
    fin tant
fin
```

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).

9 Plus courts chemins

Un automobiliste souhaite trouver le plus court chemin possible (en nombre de kilomètres) pour aller de Strasbourg à Bordeaux. Etant donnée une carte routière de France, avec les distances de chaque portion de route, comment peut-il déterminer la route la plus courte ?

Un parcours en largeur du graphe associé à la carte de France ne permettra pas de résoudre ce problème : il permettra de trouver l'itinéraire comportant le moins d'étapes (traversant le moins de sommets), mais cet itinéraire n'est pas nécessairement le plus court en nombre de kilomètres.

Une possibilité consiste à énumérer tous les chemins entre Strasbourg et Bordeaux, additionner les distances pour chacun d'eux, et choisir le plus court. Cependant, on s'aperçoit rapidement que, même en n'autorisant pas les chemins qui contiennent des circuits, il existe des milliers de possibilités, dont la plupart ne valent même pas la peine d'être considérées.

On s'intéresse maintenant à la résolution de ce type de problème. On considère des graphes orientés valués, tels qu'une valeur est associée à chaque arc, et l'on cherche le plus court chemin entre deux sommets du graphe. Cela permettra de résoudre des problèmes comme la recherche d'un itinéraire en train coûtant le moins cher, ou encore étant le plus rapide.

9.1 Définitions

Soit $G = (S, A)$ un 1-graphe orienté valué tel que la fonction $cout : A \to R$ associe à chaque arc (s_i, s_j) de A un coût réel $cout(s_i, s_j)$.

Le **coût d'un chemin** $p =< s_0, s_1, s_2, \ldots, s_k >$ est égal à la somme des coûts des arcs composant le chemin, c'est à dire,

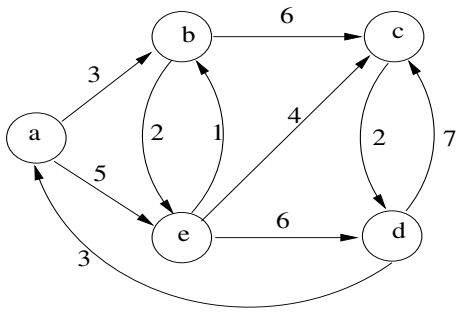
$$cout(p) = \sum_{i=1}^{k} cout(s_{i-1}, s_i)$$

Le coût d'un chemin sera aussi appelé **poid du chemin**.

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

$$\begin{array}{lcll} \delta(s_i, s_j) & = & +\infty & \text{si } \nexists \text{ de chemin entre } \mathrm{s_i} \text{ et } \mathrm{s_j} \\ \delta(s_i, s_j) & = & \min\{cout(p)/p = \text{chemin de } \mathrm{s_i} \text{ a } \mathrm{s_j}\} & \text{sinon} \end{array}$$

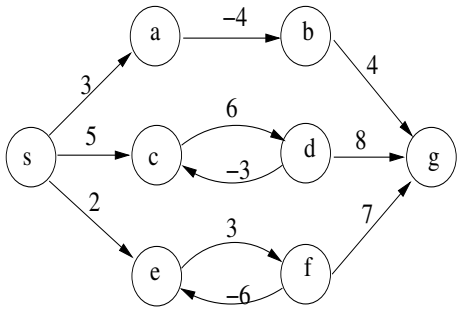
Considérons par exemple le graphe valué orienté suivant :



on a $\delta(a, b) = 3$, $\delta(a, e) = 5$, $\delta(a, c) = 9$ et $\delta(a, d) = 11$.

Conditions d'existence d'un plus court chemin : s'il existe un chemin entre deux sommets u et v contenant un circuit de coût négatif, alors $\delta(u, v) = -\infty$, et il n'existe pas de plus court chemin entre u et v. Un circuit négatif est appelé un **circuit absorbant**.

Considérons par exemple le graphe orienté valué suivant :



Le chemin $< s, e, f, e, f, g >$ contient le circuit $< e, f, e >$ de coût négatif -3. Autrement dit, à chaque fois que l'on passe dans ce circuit, on diminue de 3 le coût total du chemin. Par conséquent, $\delta(s, g) = -\infty$ et il n'existe pas de plus court chemin entre s et g.

Définition du problème des plus courts chemins à origine unique : Etant donné un graphe orienté $G = (S, A)$, une fonction $cout : A \to R$ et un sommet origine $s_0 \in S$, on souhaite calculer

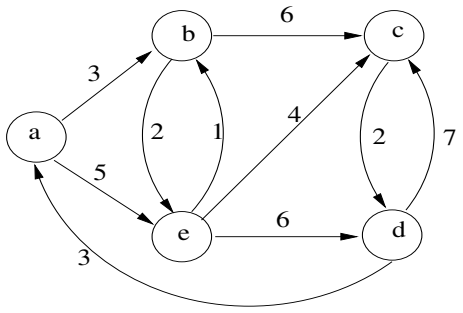
pour chaque sommet $s_j \in S$ le coût $\delta(s_0, s_j)$ du plus court chemin de s_0 à s_j. On supposera que le graphe G ne comporte pas de circuit absorbant.

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). On pourra aussi utiliser l'algorithme A* qui est généralement plus efficace dès lors que l'on dispose d'une borne minimale de la longueur d'un plus court chemin entre deux points (par exemple, la distance euclidienne).
– 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 à ce problème, par exemple, l'algorithme de Floyd-Warshall.

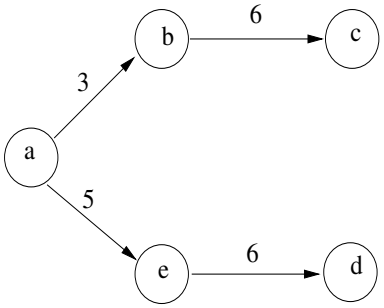
Arborescence des plus courts chemins : On va en fait calculer non seulement les coûts des plus courts chemins, mais aussi les sommets présents sur ces plus courts chemins. La représentation utilisée pour représenter ces plus courts chemins est la même que celle utilisée pour les arborescences couvrantes 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 connaitre le plus court chemin entre s_0 et un sommet s_k donné, il faudra alors "remonter" de s_k jusque s_0 en utilisant π (cf l'algorithme d'affichage du plus court chemin trouvé par le parcours en largeur au chapitre 7).

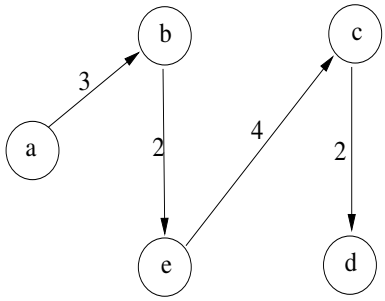
Remarque : $\delta(s_0, s_i) = \delta(s_0, \pi[s_i]) + cout(\pi[s_i], s_i)$.

Considérons par exemple le graphe valué orienté suivant :



Ce graphe possède plusieurs arborescences des plus courts chemins dont l'origine est a, par exemple





La première de ces 2 arborescences est représentée par le tableau π tel que $\pi[a] = nil$, $\pi[b] = a$, $\pi[c] = b$, $\pi[d] = e$ et $\pi[e] = a$.

On va maintenant étudier 2 algorithmes qui permettent de résoudre des problèmes de recherche de plus courts chemins à origine unique :
- l'algorithme de Dijkstra résoud ce problème lorsque tous les coûts sont positifs ou nuls,
- l'algorithme de Ford-Bellman résoud ce problème lorsque les coûts sont positifs, nuls ou négatifs, sous réserve qu'il n'y ait pas de circuit absorbant (de coût négatif).

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 :

```
proc relacher(s_i, s_j)
    si d[s_j] > d[s_i] + cout(s_i, s_j) alors
        /* il vaut mieux passer par s_i pour aller à s_j */
        d[s_j] ← d[s_i] + cout(s_i, s_j)
        π[s_j] ← s_i
    finsi
fin relacher
```

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.

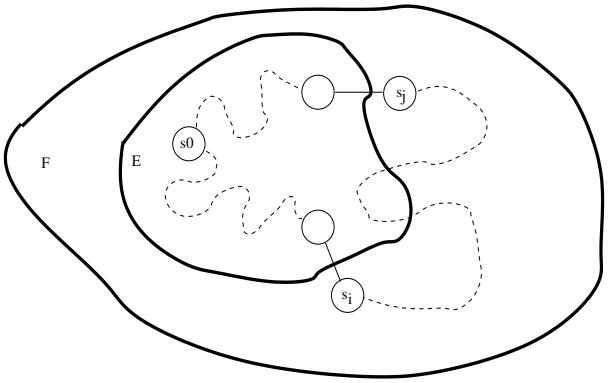
9.2 Algorithme de Dijkstra

Idée : On maintient 2 ensembles disjoints E et F tels que $E \cup F = S$. L'ensemble E contient chaque sommet s_i pour lequel on connait 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. A 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.

```
fonc Dijkstra(G = (S, A), cout : A → R+, s_0 ∈ S)
retourne une arborescence des plus courts chemins d'origine s_0
    pour chaque sommet s_i ∈ S faire
        d[s_i] ← +∞
        π[s_i] ← nil
    fin pour
    d[s_0] ← 0
    E ← ∅
```

$F \leftarrow S$
tant que $F \neq \emptyset$ faire
soit s_i le sommet de F tel que $d[s_i]$ soit minimal
/* $d[s_i] = \delta(s_0, s_i)$ */
$F \leftarrow F - \{s_i\}$
$E \leftarrow E \cup \{s_i\}$
pour tout sommet $s_j \in succ(s_i)$ faire : relacher(s_i, s_j)
fin tant
retourner(π)
fin Dijkstra

Correction de l'algorithme de Dijkstra : On peut se convaincre de la correction de l'algorithme de Dijkstra en montrant 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 $d[s_0] = 0 = \delta(s_0, s_0)$. A chaque itération, on fait entrer dans E un sommet $s_i \in F$ tel que $d[s_i]$ soit minimal. 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).



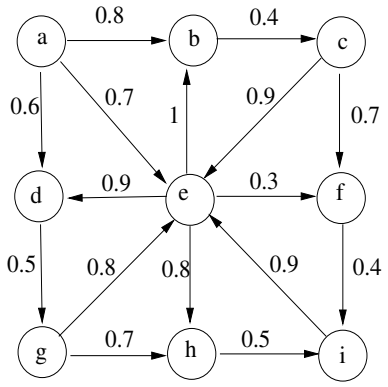
Sachant que la fin de ce chemin (de s_j par s_i) ne peut faire qu'augmenter la distance du chemin, cet autre chemin sera forcément plus long. Par conséquent, on ne pourra pas trouver de chemin plus court pour aller de s_0 à s_i, et on peut faire entrer s_i dans E, et relâcher tous les arcs qui partent de s_i.

Complexité : La complexité de cet algorithme dépend de l'implémentation du graphe (par matrice ou par liste d'adjacence), mais aussi de la façon de gérer l'ensemble F. On suppose que le graphe possède n sommets et p arcs. Si on utilise une matrice d'adjacence, l'algorithme sera en $\mathcal{O}(n^2)$. En revanche, si on utilise une liste 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. Etant 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 $n + (n-1) + (n-2) + \ldots + 2 + 1$ opérations, soit $\mathcal{O}(n^2)$. En revanche, chaque arc étant relaché une seule fois, les opérations de relachement prendront de l'ordre de p opérations. Au total on aura donc une complexité en $\mathcal{O}(n^2)$.
– 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 (immédiatement). 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 $\mathcal{O}(p * log(n))$.

Exercice : On considère un réseau de télécommunication, composé d'émetteurs/récepteurs pouvant s'envoyer des messages, avec une certaine fiabilité de communication, c'est à dire une certaine probabilité pour que la communication ne soit pas interrompue. On modélise ce problème à l'aide du graphe orienté et valué suivant, où la valuation d'un arc est une valeur réelle comprise entre 0 et 1 et indiquant la probabilité pour que la communication se passe sans problème.



Quel est le chemin le plus fiable pour envoyer un message de a vers i ?

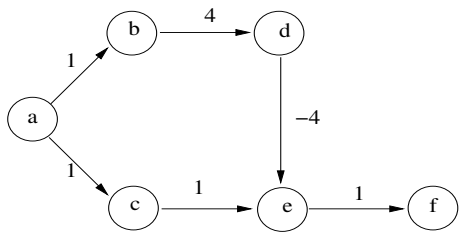
Correction : on cherche le chemin pour lequel le produit des probabilités soit maximal. Pour cela, on peut adapter l'algorithme de Dijkstra de la façon suivante :

```
pour chaque sommet s_i ∈ S faire
    d[s_i] ← 0
    π[s_i] ← nil
fin pour
d[a] ← 1
F ← S
tant que F ≠ ∅ faire
    soit s_i ∈ F le sommet tel que d[s_i] soit maximal
    F ← F − {s_i}
    pour tout sommet s_j ∈ succ(s_i) faire :
        si d[s_j] < d[s_i] * cout(s_i, s_j) alors
            d[s_j] ← d[s_i] * cout(s_i, s_j)
            π[s_j] ← π[s_i]
        finsi
    fin pour
fin tant
```

Attention : cet algorithme ne marche que si les poids des arcs sont tous compris entre 0 et 1 de telle sorte qu'à chaque fois qu'on ajoute un arc à un chemin, on diminue le coût total du chemin (pour la même raison que Dijkstra ne marche que pour des coûts positifs).

9.3 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 :



Pour aller de a à f, l'algorithme de Dijkstra va trouver le chemin $< a, c, e, f >$, de poids 3, alors que le chemin $< a, b, d, e, f >$ est de poids 2.

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'agorithme de Bellman-Ford va détecter l'existance 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 kième série de relâchement des arcs, tous les plus courts chemins de longueur k, partant de s_0, auront été trouvés. Etant 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 relachement permettrait de diminuer la valeur de $d[s_j]$. L'algorithme utilise cette propriété pour détecter la présence de circuits absorbants.)

```
fonc Bellman-Ford(G = (S, A), cout : S → R, s_0 ∈ S)
retourne une arborescence des plus courts chemins d'origine s_0
    pour chaque sommet s_i ∈ S faire
        d[s_i] ← +∞
        π[s_i] ← nil
    fin pour
    d[s_0] ← 0
    pour k variant de 1 a | S | −1 faire
        pour chaque arc (s_i, s_j) ∈ A faire relacher(s_i, s_j) fin pour
    fin pour
    pour chaque arc (s_i, s_j) ∈ A faire
        si d[s_j] > d[s_i] + cout(s_i, s_j) alors afficher("circuit absorbant") finsi
    fin pour
    retourner(π)
```

fin Bellman-ford

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

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.

Exercice : Peut-on modifier l'algorithme de Bellman-Ford pour qu'il trouve les plus longs chemins à partir d'un sommet donné ?

Correction : oui, sous réserve que le graphe ne contienne pas de circuit absorbant de poids **positif**. On peut par exemple inverser les valuations des arcs $cout(s_i, s_j) \leftarrow -cout(s_i, s_j)$ pour tout arc (s_i, s_j), puis appliquer Bellman-Ford tel quel. On peut aussi modifier légèrement l'algorithme, en remplaçant les initialisations de valeurs de $d[s_i]$ de $+\infty$ à $-\infty$, et en remplaçant le test $d[s_j] > d[s_i] + cout(s_i, s_j)$ par $d[s_j] < d[s_i] + cout(s_i, s_j)$.

Attention, on ne peut pas faire la même chose pour l'algorithme de Dijkstra.

Exercice : En l'an de grâce 1479, le sire Gwendal, paludier à Guérande, désire aller vendre sa récolte de sel à l'une des grandes foires du Duché. Il connaît les gains qu'il pourra réaliser dans chacune des foires, mais ceux-ci seront diminués des octrois qu'il devra acquitter le long du chemin emprunté pour s'y rendre. A quelle foire, et par quel chemin le paludier doit-il se rendre de façon à réaliser le plus grand bénéfice possible ?

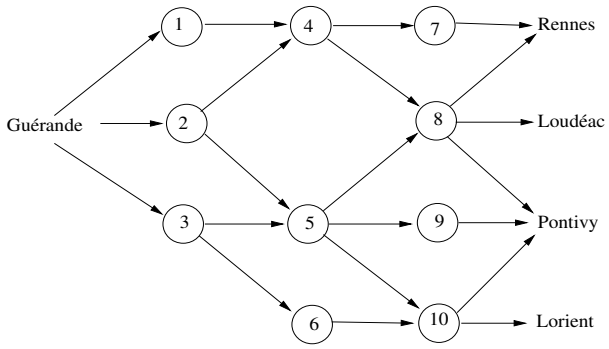
Tableau des gains en écus dans les différentes foires :

Foires	Rennes	Loudéac	Pontivy	Lorient
Gains	550	580	590	600

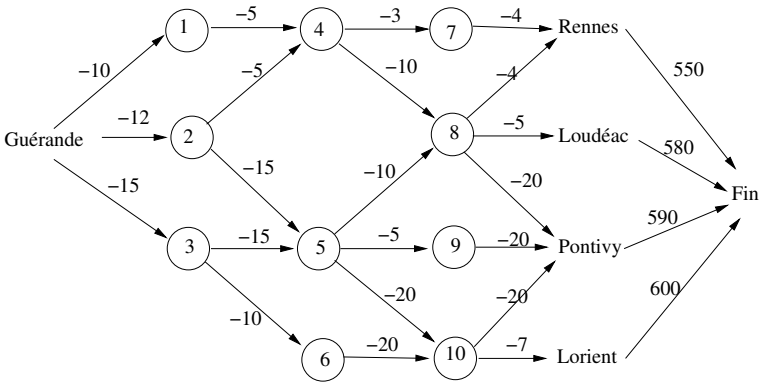
Tableau des octrois en écus dans les différentes villes :

Villes	1	2	3	4	5	6	7	8	9	10	Rennes	Loudéac	Pontivy	Lorient
Octrois	10	12	15	5	15	10	3	10	5	20	4	5	20	7

Graphe des chemins possibles de Guérande aux différentes foires :



Correction : On peut chercher le plus long chemin de Guérande à Fin dans le graphe suivant :



Remarque : Dans cet exercice, le graphe ne contient pas de circuit. Comment peut-on utiliser cette propriété pour améliorer l'algorithme de Bellman-Ford.

Correction : on peut résoudre ce problème en ne relâchant chaque arc qu'une seule fois. L'idée est de ne relâcher un arc (s_i, s_j) que si tous les arcs arrivant à s_i ont déjà été relâchés. Pour cela, il suffit de trier topologiquement les sommets du graphe (en utilisant un parcours en profondeur d'abord par exemple, cf chapitre 7), puis de considérer chaque sommet dans l'ordre ainsi défini et relâcher à chaque fois tous les arcs partant de ce sommet. Cet algorithme sera utilisé au chapitre 11 pour calculer des plus longs chemins dans un problème de planification.

9.4 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 la fonction coût est telle que tout sous chemin d'un chemin optimal est également optimal, alors on pourra appliquer Dijkstra ;
  - Sinon, on appliquera Bellman-Ford, et on vérifiera en même temps que le graphe ne comporte pas de circuits absorbants ;

10 Arbres couvrants minimaux (ACM)

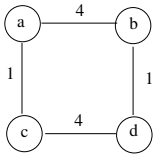
Vous êtes chargés de l'installation du câble dans la région Rhône-Alpes. Vous disposez pour cela d'une carte de l'ensemble du réseau routier (le câble est généralement disposé le long des routes). On vous demande de définir le réseau câblé de telle sorte que la longueur totale de câble soit minimale 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

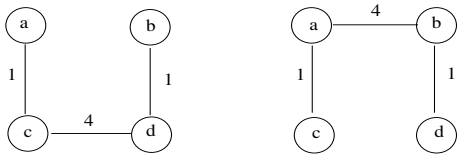
portion de route entre 2 lieux. Ce graphe est valué par une fonction coût qui spécifie pour chaque arête (s_i, s_j) la longueur de câble nécessaire pour connecter s_i à s_j. Il s'agit alors de trouver un sous-graphe connexe et sans cycle de ce graphe (autrement dit, un arbre) qui recouvre l'ensemble 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**, abrégé par 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



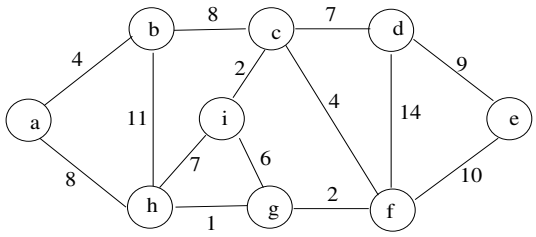
possède les 2 ACMs suivants :



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. A chaque fois, on choisira l'arête la "meilleure" 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). A 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 :

```
hg < ic = gf < ab = cf < ig < hi = cd < bc = ah < de < fe
< bh < df
```

On ajoute alors successivement dans l'ACM les arêtes :

```
hg, ic, gf, ab, cf, cd, bc, de
```

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 chaine 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. A 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_i est un prédécesseur de s_j 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_i, 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 de Kruskal

```
fonc Kruskal(G = (S, A), cout : A → R)
retourne un ACM G' = (S, K)
    pour chaque sommet s_i ∈ S faire π[s_i] ← nil
    trier les arêtes de A par ordre de coût croissant
    K ← ∅
    tant que | K |<| S | −1 faire
        soit (s_i, s_j) la k^ieme plus petite arête de A
        /* recherche de la racine r_i de la composante connexe de s_i */
        r_i ← s_i
        tant que π[r_i] ≠ nil faire r_i ← π[r_i]
        /* recherche de la racine r_j de la composante connexe de s_j */
        r_j ← s_j
        tant que π[r_j] ≠ nil faire r_j ← π[r_j]
        si r_i ≠ r_j alors
            /* on ajoute (s_i, s_j) à l'ACM */
            K ← K ∪ {(s_i, s_j)}
            /* on fusionne les deux composantes connexes */
            π[r_i] ← r_j
        fin si
    fin pour
    retourne(S, K)
fin Kruskal
```

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 gèrera 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 noeud de l'arbre jusque 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 p 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 $p * log(p)$ opérations. On passe ensuite, dans le pire des cas, p fois dans la boucle "tant que $\mid K \mid < \mid S \mid -1$" (une fois pour chaque arête (s_i, s_j)). A 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 $\mathcal{O}(log(n))$. Par conséquent, on a une complexité totale en $\mathcal{O}(p * log(p))$ (sous réserve d'utiliser une représentation par listes d'adjacence).

11 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 quadruplet (G, c, s, p) tel que
- $G = (S, A)$ est un graphe orienté,
- $c : A \rightarrow R^+$ est une fonction qui associe à chaque arc sa capacité,
- $s \in S$ est la source, et
- $p \in S$ est le puits.

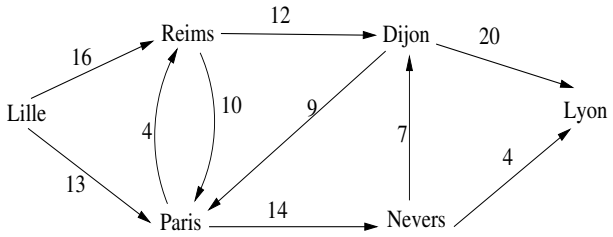
On suppose qu'il n'y a pas de sommet "inutile", c'est-à-dire que pour tout sommet $s_i \in S$, il existe un chemin de s à p passant par s_i.

Pour des raisons de commodité d'écriture, on supposera que la fonction de capacité 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 $f : S^2 \to R$ 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 :

$$\forall (s_i, s_j) \in S^2, f(s_i, s_j) = -f(s_j, s_i)$$

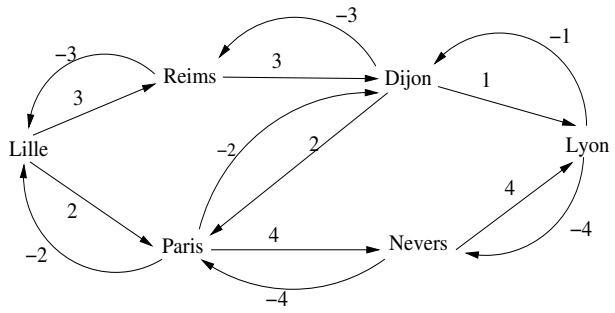
3. conservation du flot :

$$\forall s_i \in S - \{s, p\}, \sum_{s_j \in S} f(s_i, s_j) = 0$$

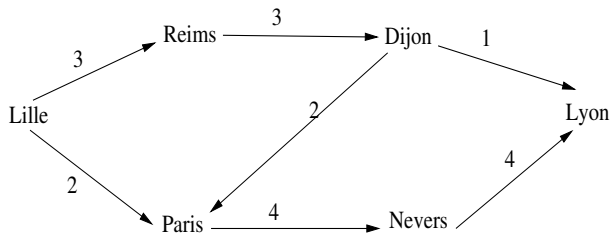
La **valeur d'un flot** f, notée $\mid f \mid$, est égale à la somme des flots partant de la source, et du fait de la propriété de conservation des flots, est aussi égale à la somme des flots arrivant au puits :

$$\mid f \mid = \sum_{s_i \in S} f(s, s_i) = \sum_{s_i \in S} f(s_i, p)$$

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



Ce flot sera généralement représenté en ne faisant figurer que les arcs de valeurs positives :



Définition du problème du flot maximal : Etant donné un réseau de transport (G, c, s, p), il s'agit de trouver un flot f tel que $\mid f \mid$ 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. Etant donné le réseau de transport $(G = (S, A), c, s, p)$, il s'agit de résoudre le problème linéaire suivant :

$$\begin{array}{lll} \text{maximiser} & \sum_{s_i \in S} f(s, s_i) & \\ \text{telque} & f(s_i, s_j) \leq c(s_i, s_j) & \forall (s_i, s_j) \in A \\ & f(s_i, s_j) = -f(s_j, s_i) & \forall (s_i, s_j) \in S^2 \\ & \sum_{s_j \in S} f(s_i, s_j) = 0 & \forall s_i \in S \end{array}$$

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 la 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, c, 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 :

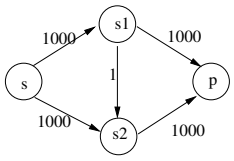
$$f'(s_i, s_j) = \begin{array}{ll} f(s_i, s_j) + c_f(ch) & \text{si } (s_i, s_j) \in ch \\ f(s_i, s_j) - c_f(ch) & \text{si } (s_j, s_i) \in ch \\ f(s_i, s_j) & \text{sinon} \end{array}$$

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

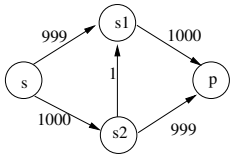
Algorithme de Ford-Fulkerson

fonc Ford-Fulkerson$(G = (S, A), c : S \times S \to R^+, s \in S, p \in S)$
retourne un flot maximal
  pour chaque $(s_i, s_j) \in S^2$ faire
    $f(s_i, s_j) \leftarrow 0$
    $c_f(s_i, s_j) \leftarrow c(s_i, s_j)$
  fin pour
  tant que il existe un chemin améliorant ch dans le graphe résiduel faire
    / calcul de la capacité résiduelle du chemin améliorant */*
    $c_f(ch) \leftarrow min_{(s_i,s_j) \in ch}(c_f(s_i, s_j))$
    / mise à jour du flot et de la capacité résiduelle le long des arcs de ch */*
    pour tout arc (s_i, s_j) du chemin améliorant ch faire
      $f(s_i, s_j) \leftarrow f(s_i, s_j) + c_f(ch)$
      $f(s_j, s_i) \leftarrow f(s_j, s_i) - c_f(ch)$
      $c_f(s_i, s_j) \leftarrow c_f(s_i, s_j) - c_f(ch)$
      $c_f(s_j, s_i) \leftarrow c_f(s_j, s_i) + c_f(ch)$
    fin pour
  fin tant
  retourne(f)
fin Ford-Fulkerson

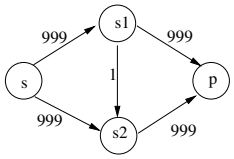
Recherche d'un chemin améliorant dans le réseau résiduel : Il s'agit d'un point critique de l'algorithme, qui peut faire varier considérablement l'efficacité de l'algorithme. Considérons par exemple le réseau de transport suivant :



Au départ, on peut trouver plusieurs chemins améliorants différents pour ce réseau, à savoir <s,s1,s2,p>, ou <s,s1,p>, ou encore <s,s2,p>. Si l'on choisit le premier chemin (<s,s1,s2,p>), de capacité résiduelle 1, alors le réseau résiduel devient :



On peut alors trouver un deuxième chemin améliorant <s,s2,s1,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 $n * p$ calculs de chemins améliorants successifs (avec n = nombre de sommets et p = 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* (étudié au chapitre 7), permettant de trouver un plus court chemin améliorant (en nombre d'arcs).

Complexité : Si on considère un réseau de transport ayant n sommets et p arcs, l'initialisation (première boucle pour) nécessitera de l'ordre de n^2 opérations. On passera ensuite au plus $n * p$ fois dans la boucle "tant que". A chaque passage dans cette boucle, on effectue un parcours en largeur pour chercher le chemin améliorant, ce qui nécessite de l'ordre de $n + p$ 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 $n^2 + (n * p) * (n + p)$ opérations. Etant donné que le réseau est connexe, on a $p \geq n - 1$. Par conséquent, la complexité globale de l'algorithme est en $\mathcal{O}(n * p^2)$.

12 Planification de projet par les réseaux

On s'intéresse ici au problème de la planification des différentes étapes d'un projet. L'objectif de cette planification est notamment de déterminer, pour chaque étape du projet, sa date de début de réalisation. Cette date doit être fixée en tenant compte d'un certain nombre de paramètres liés à la tâche (en particulier sa durée et son coût), ainsi que des différents contraintes imposées par le projet. Il est important de noter que cette planification doit pouvoir être ajustée dynamiquement dans le temps, au fur et à mesure de l'avancement du projet, afin de pouvoir prendre en compte les modifications (généralement nombreuses) survenant sur les données ou les contraintes.

Dans le contexte de ce cours, on va plus particulièrement étudier comment les graphes, et les différents algorithmes que nous avons vus jusqu'ici, peuvent aider à la modélisation et la résolution de problèmes de planification.

12.1 Coût et durée d'une tâche

Un projet est généralement décomposé en différentes tâches à effectuer. Chaque tâche correspond à une étape "élémentaire" dans la réalisation du projet et a une durée et un coût. La durée et le coût d'une tâche sont deux données inter-dépendantes : en augmentant le coût d'une tâche, on peut généralement en réduire la durée (dans certaines limites), et inversement. D'autre part, ces données ne sont généralement pas "certaines", mais elles sont estimées, notamment à partir des expériences passées dans la gestion de projets similaires. Les aspects concernant la manière d'estimer le coût et la durée d'une tâche ne seront pas étudiés dans ce cours, et dans la suite, on considèrera que la durée d'une tâche est une donnée du problème, et on notera $duree(i)$ la durée de la tâche i.

12.2 Contraintes

La réalisation des différentes tâches doit être planifiée en respectant les contraintes du problème que l'on peut classer en quatre catégories :

- Les **contraintes de précédence** imposent que certaines tâches ne peuvent commencer que si d'autres tâches sont effectivement terminées. Par exemple, le toit d'une maison ne peut être posé que si les murs sont terminés. Il est bien évidemment nécessaire que toutes les contraintes de précédence soient compatibles, autrement dit, qu'il n'y ait pas de circuit dans le graphe correspondant.
- Les **contraintes de localisation temporelle** expriment le fait qu'une tâche doit être réalisée à l'intérieur d'une période de temps imposée. Par exemple, la charpente d'une maison devra être posée pendant l'été.
- Les **contraintes d'exclusion** expriment le fait que plusieurs tâches ne peuvent être faites en même temps. Ces contraintes interviennent généralement lorsque plusieurs tâches ont besoin d'une ressource commune.
- Les **contraintes cumulatives** expriment le fait que l'on dispose d'un nombre limité de certaines ressources, et qu'à tout moment la quantité de ces ressources demandée par les tâches en cours de réalisation soit inférieure à la quantité disponible. Il s'agit d'une généralisation des contraintes d'exclusion. Par exemple, si l'on dispose de 5 ouvriers, et si une tâche i demande 3 ouvriers, une tâche j en demande 4, une tâche k en demande 2, ... alors on ne pourra pas exécuter en même temps les tâches j et k, ...

Traitement des contraintes disjonctives : Les contraintes d'exclusion et les contraintes cumulatives sont appelées des contraintes disjonctives et sont particulièrement difficiles à prendre en compte. En effet, la seule façon de les traiter est d'envisager chaque possibilité séparément : si deux tâches i et j ne peuvent pas être effectuées en même temps, alors il faudra envisager séparément deux ordonnancements. Un premier où la tâche i précède la tâche j, et un deuxième où la tâche j précède la tâche i. Ainsi, si le projet comporte n contraintes d'exclusion, il faudra envisager de l'ordre de 2^n ordonnancements différents. Rappelons que $2^{10} = 10^3$, $2^{20} = 10^6$, $2^{30} = 10^9$, ... Par conséquent, même avec un ordinateur surpuissant, on ne pourra pas envisager, de façon complète, plus d'une trentaine de contraintes disjonctives. Dans la pratique, l'exploration de ces différentes possibilités se fait par une approche par "séparation et évaluation" ("branch and bound"), qui tente de "couper" la combinatoire, généralement en utilisant des heuristiques. Cet aspect n'est pas étudié dans ce cours.

Traitement des contraintes de localisation temporelle : Ces contraintes peuvent être facilement ramenées à des contraintes de précédence en rajoutant des tâches "fictives" de durée égale à zéro, et dont la date de début est fixée. Par exemple, si la tâche i doit être effectuée entre le 1.5.99 et le 31.8.99, alors on créera les tâches T_1 et T_2 telles que $duree(T_1) = duree(T_2) = 0$, et T_1 commence le 1.5.99, et T_2 commence le 31.8.99, puis on ajoutera les contraintes de précédences $T_1 \leq i \leq T_2$.

12.3 Modélisation des contraintes de précédence par un graphe

Les contraintes de précédence peuvent être aisément modélisées à l'aide d'un graphe. Il existe essentiellement deux façons de modéliser ces contraintes : la méthode d'origine américaine P.E.R.T., aussi appelée méthode "potentiels-étapes", et la méthode d'origine française M.P.M., due à M.B. Roy, aussi appelée méthode "potentiels-tâches".

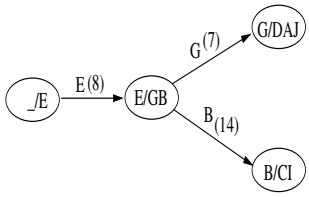
Graphe potentiels-étapes

L'idée est d'associer un arc du graphe à chaque tâche du projet, tandis que les sommets du graphe représentent des étapes dans l'avancement des travaux : chaque sommet correspond au fait que les tâches qui y arrivent doivent être terminées avant que ne débutent les tâches qui en partent.

Considérons par exemple le problème suivant :

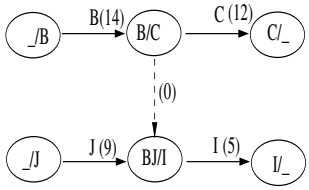
Tâche	Durée	Contraintes
A	15	G achevée
B	14	E achevée
C	12	B achevée
D	14	G achevée
E	8	
F	10	A et I achevées
G	7	E achevée
H	9	D achevée
I	5	B et J achevées
J	9	G achevée

Pour exprimer le fait que les tâches G et B ne peuvent commencer qu'une fois que la tâche E est terminée, on dessinera le graphe :

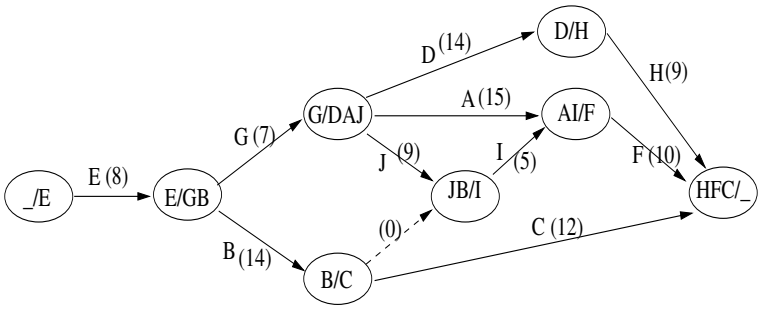


Pour plus de clarté, on désigne un sommet par un couple `(xx/yy)` où `xx` correspond à l'ensemble des tâches se terminant sur ce sommet tandis que `yy` correspond à l'ensemble des tâches commençant sur ce sommet.

Introduction de tâches fictives : Pour représenter certaines contraintes de précédence, il est nécessaire d'introduire des tâches fictives. Considérons par exemple les contraintes de précédence portant sur les tâche C et I : la tâche B doit être réalisée avant la tâche C, tandis que les tâches B et J doivent être réalisées avant la tâche I. Pour représenter ces contraintes, il est nécessaire d'introduire une tâche fictive, de durée nulle, de la façon suivante :



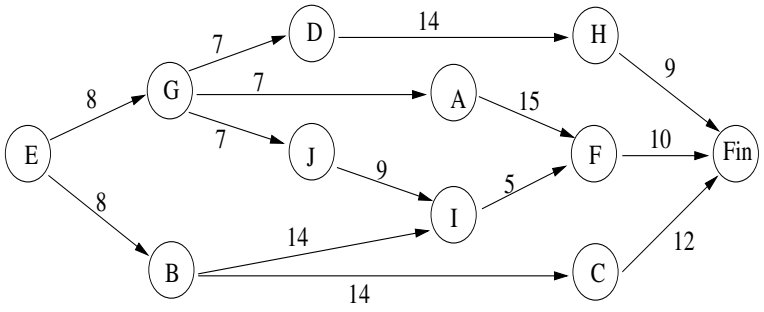
Sur l'exemple précédent, on dessine le graphe potentiel-étapes suivant :



Critique de cette représentation : Cette représentation a l'avantage d'être "descriptive" et de donner une représentation visuelle de l'enchaînement des tâches. En revanche, cette représentation est assez rigide, dans la mesure où elle ne permet pas de prendre en compte facilement l'introduction, ou la suppression, de contraintes de précédence.

Graphe potentiels-tâches

L'idée est d'associer un sommet du graphe à chaque tâche du projet, tandis que les arcs traduisent les contraintes de précédence entre tâches. A chaque arc du graphe sera associée la durée de réalisation de la tâche correspondant au sommet de départ de l'arc. Pour plus de clarté on rajoute un sommet final fictif et on rajoute un arc entre tout sommet n'ayant pas de successeur et le sommet final fictif. Sur l'exemple précédent, on obtient le graphe suivant :



Critique de cette représentation : Cette représentation est bien moins descriptive que la précédente. En contrepartie, elle est plus flexible dans le sens où elle permet de prendre en compte très facilement l'ajout ou la suppression de contraintes de précédence (il suffit de supprimer ou ajouter les arcs correspondants).

12.4 Durée minimale d'exécution

Il s'agit de calculer la date au plus tôt de fin de réalisation du projet. La **date au plus tôt** d'une tâche i, notée t_i, est le temps minimum qui sépare le début de cette tâche du début du projet, si toutes les tâches ont la durée prévue.

Quelle que soit la représentation choisie (graphe potentiels/étapes ou potentiels/tâches), on procède de façon similaire, aux détails de représentation près. On choisit ici de développer ce calcul pour la représentation "potentiels/tâches" (où un sommet est associé à chaque tâche).

Etant donné que chaque arc est valué par la durée de la tâche associée à l'extrémité initiale de l'arc, il s'agit, pour chaque tâche i de trouver le plus long chemin allant du sommet initial jusqu'au sommet associé à i. Etant donné que le graphe est acyclique (sans quoi le problème est insoluble), on peut utiliser l'algorithme introduit au chapitre 8 de la façon suivante :

Algorithme de calcul de la date au plus tôt t_i d'une tâche i :

1. Trier les sommets du graphe topologiquement (cf chapitre 7), de telle sorte que si une tâche i doit être réalisée avant une autre tâche j, alors le sommet correspondant à i soit placé avant le sommet correspondant à j dans l'ordre topologique. En même temps, on pourra vérifier que le graphe des contraintes est effectivement acyclique.
2. Pour chaque tâche i, initialiser t_i à 0.
3. Considérer chaque tâche i selon l'ordre topologique et faire

   Pour chaque successeur j de i faire :
   si $t_j < t_i + duree(i)$ alors $t_j \leftarrow t_i + duree(i)$

L'algorithme consiste donc à relâcher chaque arc (correspondant à une contrainte de précédence) une et une seule fois selon un ordre topologique sur les sommets du graphe. Ainsi, si le projet comporte n tâches et p contraintes d'antériorité, le calcul des dates au plus tôt se fera en $\mathcal{O}(p)$.

Sur l'exemple précédent, on pourra considérer l'ordre topologique suivant :

```
E, G, D, B, A, J, C, I, H, F, Fin
```

On relâchera alors les arcs du graphe dans l'ordre suivant :

Arc relâché	Date modifiée	Arc relâché	Date modifiée
1 : E/G	$t_G \leftarrow 8$	8 : B/C	$t_C \leftarrow 22$
2 : E/B	$t_B \leftarrow 8$	9 : A/F	$t_F \leftarrow 30$
3 : G/D	$t_D \leftarrow 15$	10 : J/I	$t_I \leftarrow 24$
4 : G/J	$t_J \leftarrow 15$	11 : C/Fin	$t_{Fin} \leftarrow 34$
5 : G/A	$t_A \leftarrow 15$	12 : I/F	-
6 : D/H	$t_H \leftarrow 29$	13 : H/Fin	$t_{Fin} \leftarrow 38$
7 : B/I	$t_I \leftarrow 22$	14 : F/Fin	$t_{Fin} \leftarrow 40$

A la fin, on obtiendra les dates au plus tôt suivantes :

tâche	E	G	D	B	A	J	C	I	H	F	Fin
date au plus tôt	0	8	15	8	15	15	22	24	29	30	40

12.5 Date au plus tard

Il s'agit de déterminer, pour chaque tâche i de combien de temps on peut reculer sa date de début (par rapport à sa date au plus tôt) sans retarder la date de fin du projet. Pour cela, on va calculer, pour chaque tâche i sa **date au plus tard** T_i, c'est-à-dire le temps maximum qui peut séparer le début de l'exécution de la tâche i du début du projet, sans augmenter la durée totale du projet, si toutes les tâches ont la durée prévue.

Pour calculer les dates au plus tard, il suffit de relâcher chaque arc une et une seule fois, dans l'ordre inverse de celui utilisé pour le calcul des dates au plus tôt :

Algorithme de calcul de la date au plus tard T_i d'une tâche i :

1. Reprendre l'ordre topologique défini sur les sommets pour le calcul des dates au plus tôt, et l'inverser.
2. Pour chaque tâche i, initialiser T_i à t_{Fin}.
3. Considérer chaque tâche i selon l'ordre topologique inversé et faire :

   Pour chaque prédécesseur j de i faire
   si $T_j > T_i - duree(j)$ alors $T_j \leftarrow T_i - duree(j)$

Si le projet comporte n tâches et p contraintes d'antériorité, le calcul des dates au plus tard se fera donc en $\mathcal{O}(p)$.

Sur l'exemple précédent, on pourra considérer l'ordre topologique inversé suivant :

```
Fin, F, H, I, C, J, A, B, D, G, E
```

On relâchera alors les arcs du graphe dans l'ordre suivant :

Arc relâché	Date modifiée	Arc relâché	Date modifiée
1 : F/Fin	$T_F \leftarrow 30$	8 : B/I	$T_B \leftarrow 11$
2 : H/Fin	$T_H \leftarrow 31$	9 : D/H	$T_D \leftarrow 17$
3 : I/F	$T_I \leftarrow 25$	10 : G/A	$T_G \leftarrow 8$
4 : C/Fin	$T_C \leftarrow 28$	11 : G/J	-
5 : J/I	$T_J \leftarrow 16$	12 : G/D	-
6 : A/F	$T_A \leftarrow 15$	13 : E/B	$T_E \leftarrow 3$
7 : B/C	$T_B \leftarrow 14$	14 : E/G	$T_E \leftarrow 0$

A la fin, on obtiendra les dates au plus tard suivantes :

tâche	E	G	D	B	A	J	C	I	H	F	Fin
date au plus tard	0	8	17	11	15	16	28	25	31	30	40

12.6 Marge totale

A partir de la date au plus tot et de la date au plus tard d'une tâche, on peut calculer la marge totale associée à cette tâche, c'est à dire le battement maximum dont on dispose, en plus de la durée propre de la tâche, pour fixer sa réalisation, sans pour autant perturber la date finale de fin du projet. Etant donnée une tâche i, la marge totale associée à cette tâche, notée δ_i, est définie par

$$\delta_i = T_i - t_i$$

Sur l'exemple précédent, on calcule les marges totales suivantes :

tâche	E	G	D	B	A	J	C	I	H	F	Fin
Marge totale	0	0	2	3	0	1	6	1	2	0	0

Ainsi, la tâche H a une marge totale de 2, ce qui signifie que l'on peut retarder le début de l'exécution de cette tâche de 2 unités de temps après avoir fini l'exécution de la tâche précédente D (ou encore, que l'on peut augmenter la durée de réalisation de H de 2 unités de temps), sans pour autant retarder le projet.

Notons que la marge totale suppose que tout ce qui a précédé la tâche i se soit toujours accompli le plus tôt possible, tandis que tout ce qui suit sera accompli le plus tard possible. Une conséquence de cela est que les "marges totales se partagent", ce qui signifie qu'on ne peut les utiliser qu'une seule fois sur toutes les tâches d'un chemin non ramifié. Sur notre exemple, les tâches D et H ont toutes les deux une marge totale de 2. Néanmoins, si on retarde de 2 unités de temps le début de l'exécution de la tâche D, alors on n'aura plus aucune marge pour l'exécution de la tâche suivante H.

12.7 Chemins et tâches critiques

Certaines tâches ne disposent d'aucune marge pour leur réalisation. Le moindre retard dans la réalisation d'une telle tâche entrainera nécessairement un retard global du projet. C'est pourquoi de telles tâches sont dites **tâches critiques**.

Sur l'exemple précédent, les tâches E, G, A et F sont critiques. La réalisation de ces tâches devra être particulièrement surveillée car le moindre retard se transmettra à l'ensemble du projet.

D'une façon plus générale, on appelle **chemin critique** tout chemin dans le graphe allant du sommet de début au sommet de fin du projet et ne passant que par des tâches critiques. Tout projet possèdera au moins un chemin critique, correspondant au plus long chemin entre les sommets de début et de fin du projet. Certains projets peuvent avoir plusieurs chemins critiques, s'il y a plusieurs plus longs chemins.

13 Pour en savoir plus

– La théorie des graphes
  Aimé Sache
  Presses universitaires de France, série "Que sais-je ?"
– Introduction à l'algorithmique
  T. Cormen, C. Leiserson, R. Rivest
  Editions Dunond - 1997

– Exercices et problèmes résolus de recherche opérationnelle
ROSEAUX
Editions Masson - 1986