Institut National Universitaire
Jean-François CHAMPOLLION


L3 Info

# Théorie et Algorithmique des Graphes

Thierry Montaut

Septembre 2021

# Table des matières

# Introduction

En 1736, le mathématicien suisse Leonhard Euler s'intéresse au problème des ponts de Königsberg "existe-t-il une promenade dans la ville prussienne de Königsberg passant une et une seule fois par les sept ponts de la ville ?" et le premier, propose de modéliser ce problèmes par un graphe (voir figure1).


FIGURE 1 – Les ponts de Königsberg. Graphe d'Euler

En 1856, le mathématicien irlandais William Hamilton utilise ce modèle pour chercher un chemin autour du monde passant une et une seule fois par 20 villes prestigieuses et en fait un jeu en plaçant les 20 villes aux sommets d'un dodécaèdre régulier (figure 2.)

FIGURE 2 – Le voyage autour du monde d'Hamilton

Depuis, le recours à un graphe pour modéliser un système réel est devenu un outil classique des mathématiques discrètes et les propriétés des graphes ont été largement étudiées notamment par le français Claude Berge, un des fondateurs de la théorie des graphes dans les années 50.

Les graphes permettent de modéliser et d'étudier les ensembles structurés complexes, les relations entre objets, l'évolution de systèmes dans le temps, les réseaux (informatiques ou de

distribution), notamment les problèmes de connexion, de cheminement, la décomposition de projets en tâches.
Peu de domaines d'études ont autant d'applications dans des disciplines aussi diverses que mathématiques, informatique, mais aussi sociologie, théorie des jeux, botanique, zoologie, économie, automatique, génie industriel et beaucoup d'autres.

A l'origine, les graphes sont des objets mathématiques étudiés par la **théorie des graphes**, une branche des mathématiques discrètes. Cette théorie s'applique à prouver des propriétés des graphes.
**L'algorithmique des graphes,** liée à l'informatique, complète cette théorie en s'intéressant non plus à l'étude des graphes mais à la réalisation d'algorithmes de traitement des graphes, à leur mise en oeuvre et à leur optimisation.
Les applications des graphes ayant pris une grande importance ces 20 dernières années en optimisation de projet, production, logistique, leur étude constitue aujourd'hui une partie importante de la **recherche opérationnelle**, discipline industrielle dont le but est de fournir des méthodes d'optimisation pour les problèmes de grande taille. La recherche opérationnelle s'intéressera donc entre autre à la mise au point de méthodes de résolution des problèmes de graphes "difficiles" c'est-à-dire de complexité non polynomiale.

Nous nous attacherons dans ce cours à définir les principaux concepts de la théorie des graphes en étudiant successivement les graphes non orientés, les graphes orientés, puis les graphes valués. Nous étudierons leurs propriétés mathématiques mais aussi et surtout les algorithmes permettant de manipuler ces graphes et leurs principales applications. Nous devrons toujours être sensibles à la complexité de ses algorithmes et la recherche de leur optimisation et présenterons en fin de cours quelques méthodes de résolution pour les problèmes "difficiles" (non polynomiaux). Les algorithmes seront décrits dans le cours en pseudo-code et devront être mis en oeuvre en Python lors des séances de TP.

# Chapitre 1

# Graphes orientés et non orientés

## 1 Définitions - Vocabulaire

**Définition 1** *Un graphe non orienté est un couple $(X, E)$ où $X$ est un ensemble fini de sommets et $E$ un ensemble de paires $\{x, y\}$ (non ordonnées) de deux éléments de $X$ appelées arêtes. On autorise parfois que $E$ contienne des parties de la forme $\{x, x\}$ (appelée boucle) ou qu'il contienne plusieurs arêtes égales, on parle alors de* **multigraphe**. *Si on veut insister sur leur absence on parle de* **graphe simple**.

**Définition 2** *Un graphe orienté est un couple $(X, E)$ où $X$ est un ensemble fini de sommets et $E$ un ensemble de couples $(x, y)$ (ordonnés) d'éléments de $X$ appelées arcs.*
*On autorise parfois que $E$ contienne des couples de la forme $(x, x)$ (appelée boucle) ou qu'il contienne plusieurs arcs égaux, on parle alors de* **multigraphe orienté**. *Si on veut insister sur leur absence on parle de* **graphe orienté simple**.

Afin de mieux voir les choses, on peut représenter les petits graphes par leur diagramme sagittal :
**Cas non orienté :**
Une représentation sagittale de $G$ est obtenue en représentant les sommets de $X$ dans le plan et en représentant l'arête $\{x, y\}$ par un segment joignant $x$ à $y$, et une arête $\{x, x\}$ par une boucle joignant $x$ à lui même.
Par exemple le graphe orienté

$$G_1 = (X = \{1, 2, 3, 4, 5\}, E = \{(1, 5), (2, 1), (2, 4), (3, 2), (4, 3), (5, 2), (5, 4)\}$$

sera représenté par le diagramme de la figure 1.1.

**Cas orienté :**
Une représentation sagittale de $G$ est obtenue en représentant les sommets de $X$ dans le plan et en représentant l'arc $(x, y)$ par une flèche joignant $x$ à $y$, et une boucle $\{x, x\}$ par une flèche joignant $x$ à lui même.
Par exemple le multigraphe non orienté

$$G_2 = (X = \{1, 2, 3, 4, 5\}, E = \{\{1, 2\}, \{1, 5\}, \{1, 5\}, \{2, 3\}, \{2, 4\}, \{2, 4\}, \{3, 3\}, \{3, 4\}, \{4, 5\}\}$$

sera représenté par le diagramme de la figure 1.2.

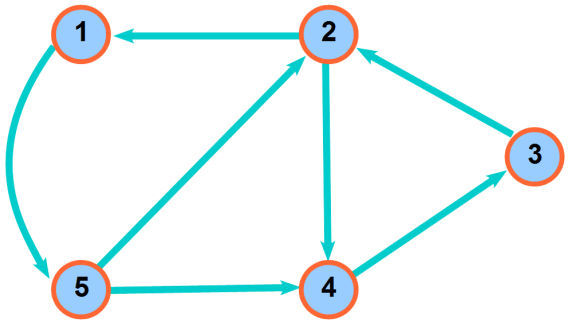



FIGURE 1.1 – représentation sagittale du graphe simple orienté $G_1$.

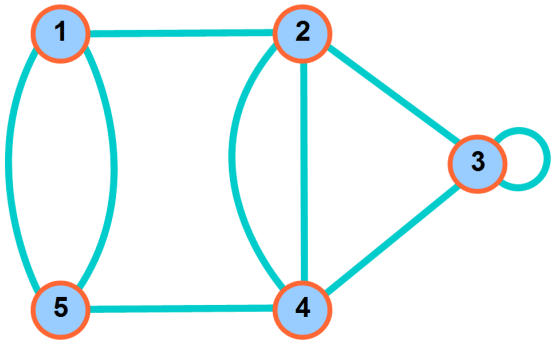



FIGURE 1.2 – représentation sagittale du multigraphe non orienté $G_2$.

Il y a souvent confusion entre un graphe et sa représentation sagittale mais il faut se méfier car il n'est pas toujours facile de reconnaître un même graphe à partir de deux représentations sagittales différentes.

**Remarques**

- Sauf cas exceptionnels (où cela sera précisé explicitement), nous ne nous intéresserons qu'aux **graphes simples.** Le graphe d'Euler et le graphe $G_2$ de la figure 1.2 sont des exemples de multigraphes.
- On notera souvent indifféremment $(x, y)$ les arêtes ou les arcs.

- Pour simplifier les choses nous considérerons dans tout le cours que $X = \{1, 2, \ldots, n\}$.

**Définition 3** *Soit $G = (X, E)$ un graphe .*
**L'ordre** *du graphe $G$ est le nombre de ses sommets. On notera dans tout ce cours $n = |X|$ l'ordre du graphe $G$. On notera également $m$ le nombre d'arêtes ou d'arcs de $G$.*

**Définition 4** *On dit que le graphe orienté $G_o$ est une orientation du graphe non orienté $G$ s'ils*

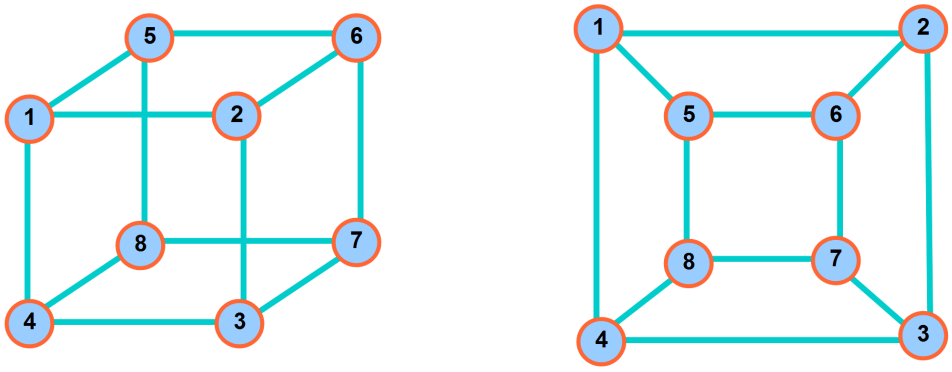



FIGURE 1.3 – Deux représentation sagittales différentes d'un même graphe

*ont le même ensemble de sommets $X$ que pour toute arête $\{x, y\}$ de $G$ soit $(x, y)$ soit $(y, x)$ est un arc de $G_o$ (et pas les deux) et que pour tout arc (x,y) de $G_o$, $\{x, y\}$ est bien une arête de $G$.*

La figure 1.10 montre un graphe non orienté et une de ses orientations possibles.

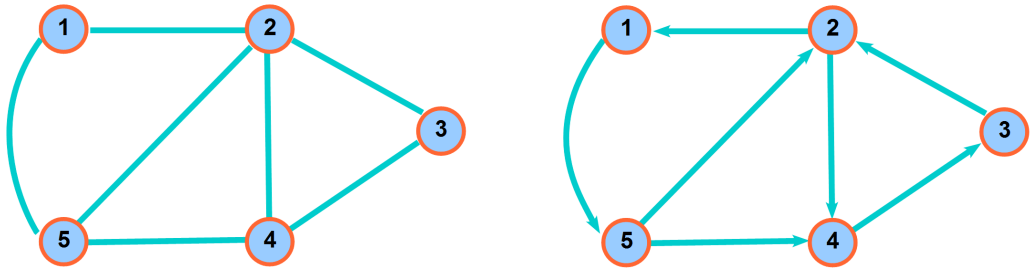



FIGURE 1.4 – Un graphe non orienté et une de ses orientations possibles

Exercice : Combien existe-t-il d'orientations d'un graphe à $n$ sommets et $m$ arêtes ?

**Définition 5** *Si $G = (X, E)$ est un graphe orienté, on appelle graphe inverse de $G$ le graphe $G' = (X, E')$ de même ensemble de sommets $X$ et tel que $G$ possède l'arc $(x, y)$ si et seulement si $G'$ possède l'arc $(y, x)$.*

**Définition 6** *Un graphe est dit* **planaire** *s'il admet une représentation sagittale sans que les arêtes se croisent. (Tout graphe a moins de 5 sommets est planaire).*

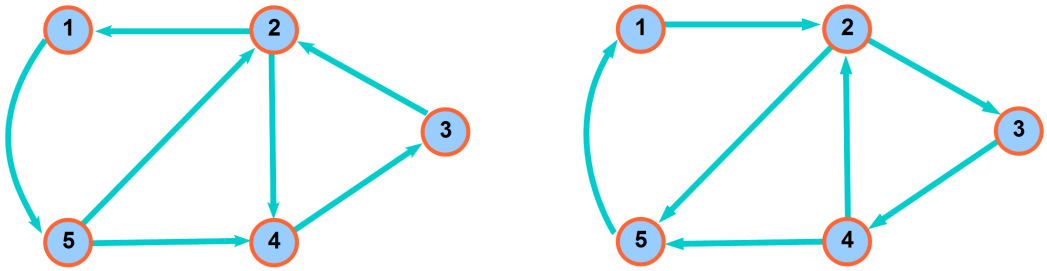



FIGURE 1.5 – Le graphe $G_1$ et son graphe inverse

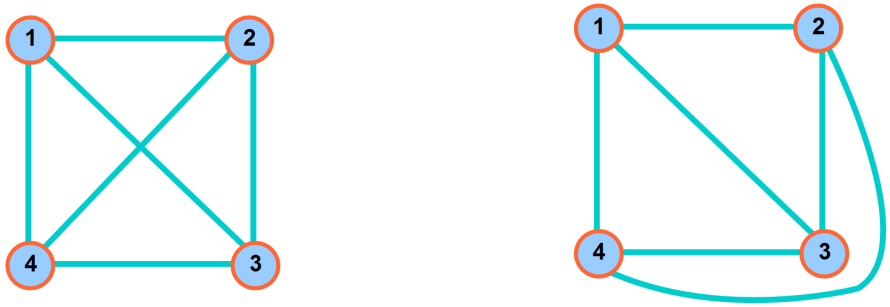



FIGURE 1.6 – Le graphe $K_4$ est planaire.

## 2 Voisinage, degré

La théorie des graphes manipule un vocabulaire important, généralement très naturel mais qui doit être bien maîtrisé.

On suppose ici que $G$ est un graphe non orienté :

**Définition 7** *Les deux sommets $x$ et $y$ sont dits* **adjacents ou voisins** *s'il existe une arête $\{x, y\}$ dans $G$. $x$ et $y$ sont appelés* **les extrémités** *de l'arête $\{x, y\}$.*

**Le voisinage** *du sommet $x$ est l'ensemble noté $V(x)$ de ses sommets adjacents (répétés avec leur multiplicité dans le cas d'un multigraphe)*
*On appelle* **degré** *de $x$ le nombre d'arêtes incidentes à $x$. C'est donc aussi le cardinale du voisinage de $x$. On note $d(x) = |V(x)|$ le degré de $x$.*
*Un sommet de degré nul est dit* **isolé**, *un sommet de degré 1 est dit sommet* **pendant**.

Exemple : Dans le multigraphe $G_2$, le sommet 2 est de degré 4 et de voisinage : $\{1, 3, 4, 4\}$.

**Définition 8** *Soit maintenant $G = (X, E)$ un graphe orienté.*
*Les deux sommets $x$ et $y$ sont dits adjacents s'il existe un arc $(x, y)$ dans $G$, $x$ est* **l'origine** *de*

*l'arc et y son extrémité. x est un* **prédécesseur** *de y et y un* **successeur** *de x.*
*Le* **voisinage sortant** *du sommet x est l'ensemble noté $V^+(x)$ de ses successeurs.*

*On appelle* **degré sortant** *de x le nombre de successeurs de x. On note $d^+(x) = |V^+(x)|$ le degré sortant de x.*

*Le* **voisinage entrant** *du sommet x est l'ensemble noté $V^-(x)$ de ses prédécesseurs.*
*On appelle* **degré entrant** *de x le nombre de prédécesseurs de x. On note $d^-(x) = |V^-(x)|$ le degré entrant de x.*
*Un sommet x de degré entrant égal à 1 et de degré sortant nul est dit pendant. Un sommet de degré entrant nul est une source, un sommet de degré sortant nul est un puits.*

Dans le graphe $G_1$, 4 est de degré entrant 2, de voisinage entrant $\{2, 5\}$, de degré sortant 1 et de voisinage sortant $\{3\}$.

# 3 Graphes non orientés classiques

Le graphe établi par Euler pour résoudre le problème des ponts de Königsberg est le célèbre graphe de la figure 1.7. Une fois le problème ainsi modélisé, la réponse à la question est clairement non car quand on traverse un sommet, on y arrive par une arête et on en ressort par une autre, donc pour qu'un tel chemin (dit eulérien) existe, il faut qu'autour de chaque sommet il y ait un nombre pair d'arêtes ce qui n'est pas le cas ici.

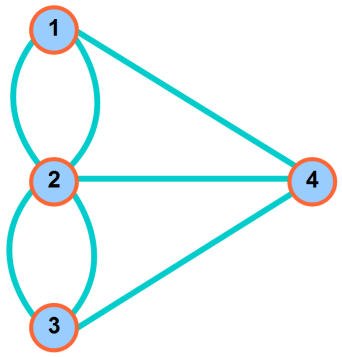



FIGURE 1.7 – Le graphe d'Euler est un multigraphe.

**Définition 9** *Un graphe non orienté est dit* **complet** *si toute paire de sommets est reliée par une arête.*
*On note $K_n$ tout graphe complet à n arêtes (En hommage à Kasimir Kuratowski qui donna en 1930 une caractérisation des graphes planaires, il est donc autorisé de penser qu'on les appelle K comme Kasimir !).*

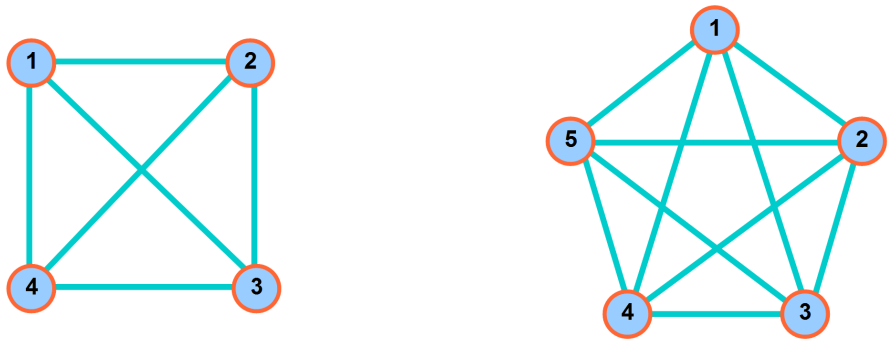



FIGURE 1.8 – Les graphes $K_4$ et $K_5$.

$K_5$ est le plus petit graphe complet non planaire.

**Définition 10** *Un graphe $G = (X, E)$ est dit k-parti s'il existe une partition de $X$ en $k$ ensembles non vides tel que toute arête de $G$ ait ses extrémités dans des ensembles différents. Si $k$ n'est pas précisé on parle de graphe multiparti, pour $k = 2$ et $k = 3$ on parle de graphe biparti, triparti.*

Précisons le cas (fondamental) où $k = 2$.

**Définition 11** *Un graphe $G$ est dit biparti s'il existe une partition de l'ensemble des sommets en $X = X_1 \cup X_2$ telle que toute arête $(x, y)$ de $G$ ait une extrémité dans $X_1$ et l'autre dans $X_2$.*

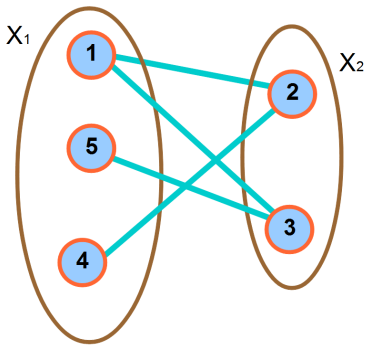



FIGURE 1.9 – Exemple de graphe biparti

**Définition 12** *On appelle graphe biparti complet de tailles $n$ et $p$ un graphe biparti tel que $|X_1| = n$, $|X_2| = p$, et pour tout $x, y \in X_1 \times X_2$ il existe une arête entre $x$ et $y$. Toujours en hommage à Kasimir Kuratowski, on note $K_{n,p}$ un tel graphe biparti complet.*

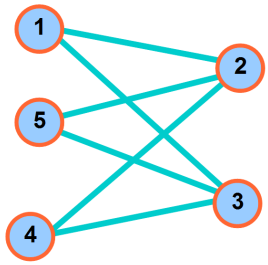



FIGURE 1.10 – Le graphe biparti $K_{32}$

exercice : Montrer que $K_{2,3}$ est planaire.

Nous utiliserons beaucoup en TD le graphe de Petersen (figure 1.11) à 10 sommets et 15 arêtes, du au mathématicien danois Julius Petersen qui l'étudia en 1898 et qui possède de très nombreuses propriétés.

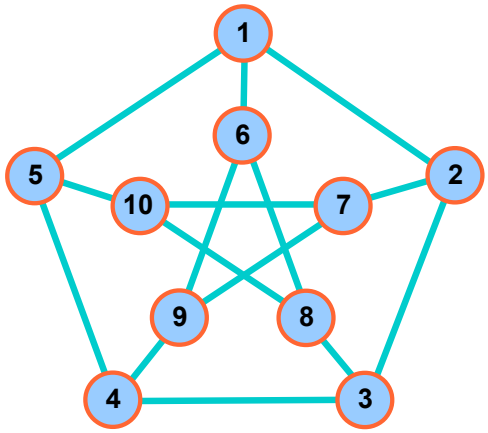



FIGURE 1.11 – Le graphe de Petersen

En particulier, Donald Ervin Knuth, informaticien et mathématicien américain de renom, pionniers de l'algorithmique et professeur émérite de l'art de programmer le cite comme "une configuration remarquable, qui sert de contre-exemple à de nombreuses prédictions optimistes sur ce qui pourrait être vrai pour tous les graphes." C'est donc un bon réflexe de commencer par tester une intuition ou un algorithme sur le graphe de Petersen.

# 4 Premières propriétés

**Proposition 1** *Soit $G$ un graphe non orienté simple à $n$ sommets et $m$ arêtes :*

*(1) Le graphe complet à $n$ sommets possède $m = \frac{n(n-1)}{2}$ arêtes.*

*(2) $G$ vérifie donc $m \leq \frac{1}{2}n(n-1)$. (On a donc toujours $m \leq O(n^2)$. )*

*(3) $\sum_{x \in X} d(x)$ est le nombre d'extrémités d'arêtes, c'est donc deux fois le nombre d'arêtes, c'est donc un nombre pair.*

*(4) Il y a un nombre pair de sommets de degré impair.*

**Démonstration**

■

Exercice :

- Montrer que tout graphe admet deux sommets de même degré.
- On suppose que l'amitié est réciproque (même si la littérature et la vie nous en donnent beaucoup de contre-exemple). Montrer que dans tout assemblée de personnes, il existe toujours deux personnes qui ont exactement le même nombre d'amis dans l'assemblée.

**Proposition 2** *Soit $G$ un graphe orienté simple à $n$ sommets et $m$ arcs :*

*(1) $G$ vérifie $m \leq n(n-1)$. (On a donc toujours $m \leq O(n^2)$. )*

*(2) $\sum_{x \in X} d^+(x)$ est le nombre d'origine ou d'extrémités d'arcs, c'est donc $s$ le nombre d'arcs.*

**Démonstration**

■

# 5 Graphes partiels et sous-graphes

**Définition 13** *Soit $G = (X, E)$ un graphe, $G' = (X, E')$ est un* **graphe partiel** *de $G$ si $E' \subset E$. Dit autrement ils ont mêmes sommets mais on a enlevé quelques arêtes ou arcs...*

à ne pas confondre avec la définition suivante

**Définition 14** *$G' = (X', E')$ est un* **sous-graphe** *de $G$ si $X' \subset X$ et que $E' = E \cap X \times X$. Dit autrement on a enlevé quelques sommets et les arêtes ou arcs dont ils étaient extrémités.*

Exercice : Combien un graphe à $n$ sommets et $m$ arêtes possède-t-il de graphes partielles ? de sous-graphes ?

**Définition 15** *Si $G$ est un graphe non orienté, on appelle* **clique** *de $G$ un sous graphe complet de $G$. On précise parfois $k$-clique pour une clique d'ordre $k$.*

Exercice : Bien identifier la forme d'une $k$-clique pour $k = 1, 2, 3, 4, 5$.

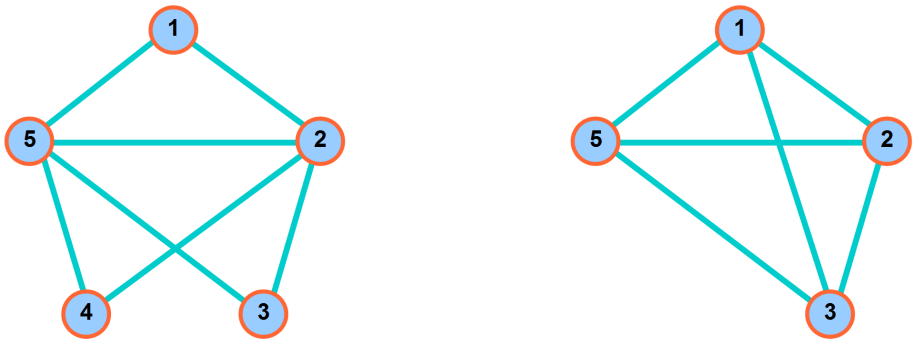



FIGURE 1.12 – A gauche un graphe partiel de $K_5$. A droite un de ses sous graphe

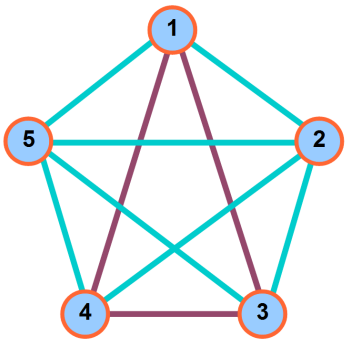



FIGURE 1.13 – Le graphe $K_5$ possède des 3-cliques

# 6 Chaines et cycles d'un graphe non orienté. Connexité

**Définition 16** *On appelle chaîne de $G$ toute suite alternée de sommets et d'arêtes de $G$ : $c = (x_0, a_1, x_1, ..., a_n, x_n)$ telle que $\forall i \in [1, n]$, $a_i = (x_{i-1}, x_i)$.*
*$c$ est une chaîne de longueur $n$ (le nombre d'arêtes) joignant $x_0$ à $x_n$.*
*Dans le cas d'un graphe simple, il est inutile de préciser l'arête et nous noterons donc simplement :*

$$c = (x_0, x_1, ..., x_n).$$

*ou*

$$c = x_0 \text{ — } x_1 \text{ — } x_2 \ \dots \ x_{n-1} \text{ — } x_n$$

**Définition 17** *1)Soit $x$ et $y$ deux sommets de $G$. On dit que $y$ est accessible à partir de $x$ dans $G$ s'il existe une chaîne de $G$ joignant $x$ à $y$. On considère $(x_0)$ comme une chaîne de longueur nulle joignant $x_0$ à lui-même.*
*2)$G = (X, E)$ est dit connexe si $\forall (x, y) \in X^2$, $y$ est accessible à partir de $x$.*
*3)L'accessibilité définit une relation d'équivalence entre les sommets. Les classes d'équivalence de cette relation sont appelées composantes connexes de $G$.*

**Définition 18** *Une chaîne est dite simple si elle ne passe pas deux fois par la même arête et est dite élémentaire si elle ne passe pas deux fois par le même sommet.Il est clair qu'une chaîne élémentaire est simple.*

**Définition 19** *On appelle cycle une chaîne simple dont l'origine est égale à l'extrémité, i.e. une chaîne simple de la forme* $c = (x_0, x_1, ..., x_{n-1}, x_0)$.

Remarque : On impose à une chaîne d'être simple (ne pas passer deux fois par la même arête, pour éviter de pouvoir considérer l'aller-retour $x - y - x$ comme un cycle.

**Théorème 1** *(1) Toute chaîne élémentaire a une longueur* $\leq n - 1$.

*(2) Tout cycle élémentaire a une longueur* $\leq n$.

*(3) De toute chaîne on peut extraire une chaîne élémentaire.*

# 7 Chemins et circuits d'un graphe orienté. Forte connexité

Le passage du cas non orienté au cas orienté n'entraine parfois qu'un changement de vocabulaire, mais le changement est parfois plus profond (c'est le cas par exemple de la notion de forte connexité).

**Définition 20** *On appelle chemin de* $G$ *toute suite alternée de sommets et d'arcs de* $G$ *:* $c = (x_0, a_1, x_1, ..., a_n, x_n)$ *telle que* $\forall i \in [1, n], \quad a_i = (x_{i-1}, x_i)$. $c$ *est un chemin de longueur* $n$ *(le nombre d'arcs) joignant* $x_0$ *à* $x_n$.
*Dans le cas d'un graphe simple, il est inutile de préciser l'arc et on notera :*

$$c = (x_0, x_1, ..., x_n).$$

*ou*

$$c = x_0 \quad \to \quad x_1 \quad \to \quad x_2 \quad \dots \quad x_{n-1} \quad \to \quad x_n$$

**Définition 21** *1)Soit* $x$ *et* $y$ *deux sommets de* $G$. *On dit que* $y$ *est accessible à partir de* $x$ *dans* $G$ *s'il existe un chemin de* $G$ *joignant* $x$ *à* $y$.
*2)*$G = (X, E)$ *est dit fortement connexe si* $\forall (x, y) \in X^2$, $y$ *est accessible à partir de* $x$.
*3)On construit une relation d'équivalence entre les sommets par* $x$ *est en relation avec* $y$ *ssi* $y$ *est accessible à partir de* $x$ *et* $x$ *à partir de* $y$. *Les classes d'équivalence de cette relation sont appelées composantes fortement connexes de* $G$.

**Définition 22** *Un chemin est dit élémentaire si il ne passe pas deux fois par le même sommet.*

**Définition 23** *On appelle circuit un chemin dont l'origine est égale à l'extrémité, i.e. un chemin de la forme* $c = (x_0, x_1, ..., x_{n-1}, x_0)$.

Cette fois-ci on autorise le circuit $(x, y, x)$ car les arcs $(x, y)$ et $(y, x)$ sont différents.
Par abus de langage on se laissera parfois à dire chemin et circuit que le graphe soit orienté ou non.

**Proposition 3** *De tout chemin on peut extraire un chemin élémentaire.*

**Démonstration** .

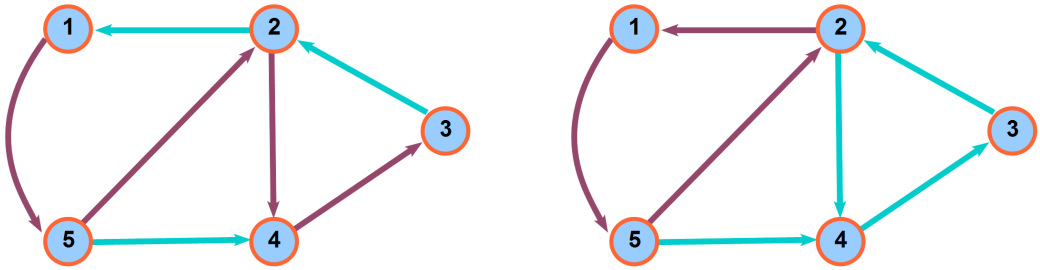



FIGURE 1.14 – Un chemin élémentaire et un circuit de $G_1$.

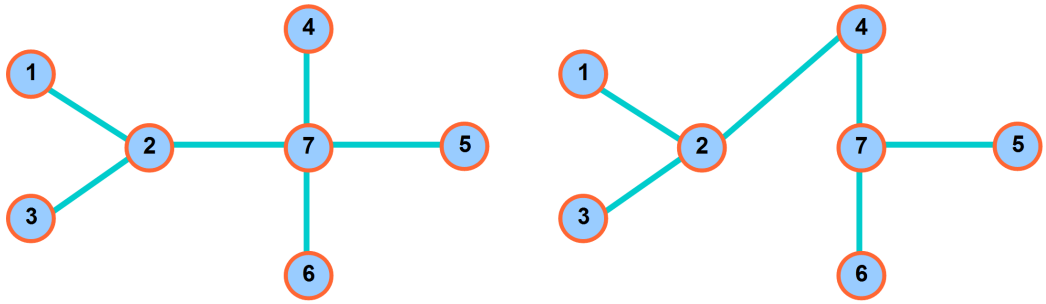



FIGURE 1.15 – Deux exemples d'arbres $A_1$ et $A_2$.

# 8 Arbres et arborescences

**Définition 24** *On appelle* **arbre** *un graphe non orienté connexe et sans cycle.*
*Une union d'arbre, donc un graphe non orienté et sans cycle est* **une forêt**.

**Théorème 2** *Il existe de nombreuses caractérisations équivalentes des arbres : Un graphe $T$ à $n$ sommets et $m$ arêtes est un arbre ssi :*

*(1) $T$ est connexe et $m = n - 1$*

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

*(3) $T$ est sans cycle et maximal en tant que tel (si on lui ajoute une arête entre deux sommets quelconques, cela crée toujours un cycle)*

*(4) $T$ est connexe et minimal en tant que tel (si on lui enlève une arête quelconque, le graphe n'est plus connexe)*

*(5) Deux sommets quelconques de $T$ sont reliés par un unique chemin.*

Nous montrerons ces équivalences en TD (Elles se montrent toutes par récurrence sur $n$, ce sera l'occasion de réviser la preuve par récurrence...)

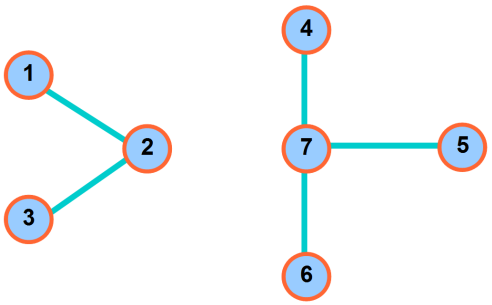



FIGURE 1.16 – Une forêt est une union d'arbres

En pratique, pour prouver qu'un graphe non orienté est un arbre, on aura donc le choix du critère à appliquer en fonction du contexte.
Il ne faut pas confondre les arbres qui sont des graphes non orienté avec les arborescences qui seront des graphes orientés.

**Définition 25** *Un arbre est dit enraciné en $r$ lorsqu'on a choisi un de ses sommets $r$ que l'on appelle alors la racine de l'arbre enraciné.*
*On appelle arborescence tout graphe obtenu à partir d'un arbre par enracinement en $r$ et orientation des arêtes à partir de $r$.*

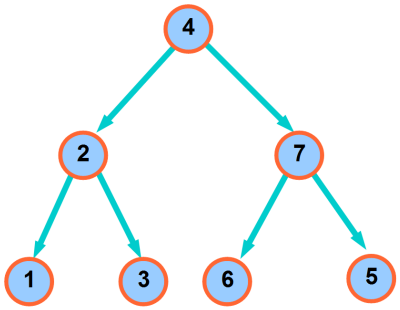



FIGURE 1.17 – Une arborescence obtenue par enracinement d'$A_2$ sur la racine 4

# Chapitre 2

# Représentations d'un graphe - Premiers algorithmes

## 1 Représentation par liste d'arêtes

**Définition 26** *La représentation d'un graphe $G$ par sa liste d'arête consiste à représenter $G$ par son nombre de sommets et la liste des arêtes ou des arcs de $G$.*

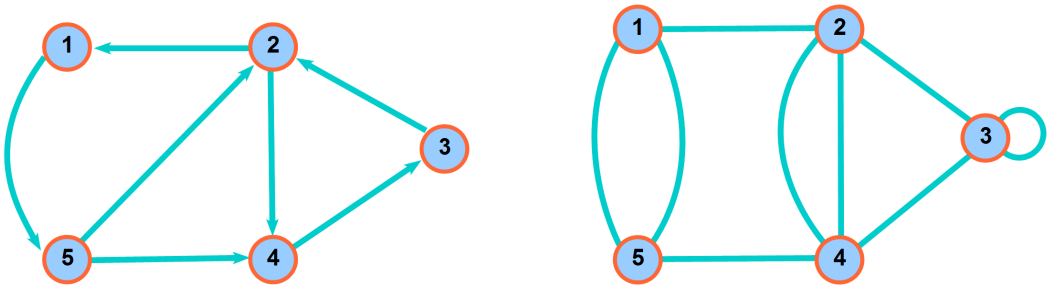



FIGURE 2.1 – Le graphe orienté $G_1$ et le multigraphe non orienté $G_2$.

Exemple :

- Le graphe orienté $G_1$ est représenté par la liste d'arcs

$$[[1,5],[2,1],[2,4],[3,2],[4,3],[5,2],[5,4]].$$

- Le multigraphe non orienté $G_2$ est représenté par la liste d'arêtes

$$[[1,2],[1,5],[1,5],[2,1],[2,4],[2,4],[2,3],[3,2],[3,3],[3,4],[4,2],[4,2],[4,3],[4,5],[5,1],[5,1],[5,4]].$$

Remarque : La représentation de $G$ par sa liste d'arête est "naturelle" d'après la définition d'un graphe, mais nous verrons qu'elle est peu efficace en pratique. Nous lui préférerons les deux représentations suivantes. En algorithmique elle est en général seulement utilisée pour la saisie du graphe et immédiatement transformée en représentation par matrice ou liste d'adjacence.

# 2 Représentation matricielle

**Définition 27** *On appelle matrice d'adjacence de $G$ la matrice $A = (a_{i,j})$ telle que $\forall (i,j) \in X^2$ :*

- *Dans le cas d'un graphe non orienté : $a_{i,j}$ est égal à 1 si $\{i,j\} \in E$ et 0 sinon. On a donc toujours $a_{i,j} = a_{j,i}$.*
- *Dans le cas d'un graphe orienté : $a_{i,j}$ est égal à 1 si $(i,j) \in E$ et 0 sinon. La matrice n'est donc plus nécessairement symétrique.*
- *Dans le cas d'un multigraphe : $a_{i,j}$ est le nombre d'arêtes ou d'arcs entre les sommets $i$ et $j$ et les coefficients diagonaux comptent alors les boucles.)*

### Cas d'un graphe non orienté

**Proposition 4** *(1) La matrice d'adjacence d'un graphe non orienté est toujours symétrique.*

*(2) La somme des coefficients de $A$ est égale à deux fois le nombre d'arêtes $m$.*

*(3) La somme des éléments de la ligne $i$ ou de la colonne $i$ est égale à $d(i)$.*

### Cas d'un graphe orienté

**Proposition 5** *(1) La somme des coefficients de $A$ est égale au nombre d'arcs $m$.*

*(2) La somme des éléments de la ligne $i$ est égale à $d^+(i)$.*

*(3) La somme des éléments de la colonne $i$ est égale à $d^-(i)$.*

Exemple :

- La matrice d'adjacence du graphe non orienté $G_1$ est la matrice (non symétrique)

$$M_1 = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \end{pmatrix}.$$

  On voit qu'il n'y a pas de boucle car les coefficients diagonaux sont tous non nuls.

- La matrice d'adjacence du multigraphe non orienté $G_2$ est la matrice symétrique

$$M_2 = \begin{pmatrix} 0 & 1 & 0 & 0 & 2 \\ 1 & 0 & 1 & 2 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 2 & 1 & 0 & 1 \\ 2 & 0 & 0 & 1 & 0 \end{pmatrix}.$$

  On voit qu'il y a une boucle et des arêtes multiples.

Cette représentation permet d'utiliser les opérations matricielles et les propriétés du calcul matriciel, elle est donc très intéressante théoriquement :

**Proposition 6** *On considère la matrice $A^k$ obtenue en multipliant $k-1$ fois la matrice $A$ par elle-même. Alors le coefficient d'indice $(x,y)$ de la matrice $A^k$ est le nombre de chemins de longueur $k$ entre les sommets $x$ et $y$.*

Algorithmiquement, cette représentation est pratique par exemple pour tester l'existence d'un arc entre deux sommets (Si A[i,j] alors ...), pour ajouter ou retirer un arc (mettre A[i,j] à 0 ou 1), ce qui se fait en temps constant $O(1)$. Il est également facile de parcourir ou de traiter tous les sommets adjacents d'un sommets (en $O(n)$.)

```
{traiter les voisins de i, i.e. parcourir la ligne i}
Pour j de 1 à n faire
   si A[i,j] alors traiter j;
```

Exemple : Calculer le degré de tous les sommets .

```
Pour i de 1 à n faire {pour chaque sommet}
Début
D[x]:=0;
Pour j de 1 à n faire
    si A[i,j] alors D[x]:=D[x]+1;
  Fin;
```

On voit sur cet exemple l'inconvénient de cette représentation, les algorithmes de parcours sont en $O(n)$ quel que soit le nombre de prédécesseurs ou de successeurs de $x$. Une consultation de tous les arcs nécessite de visiter toutes la matrice donc un algorithme en $O(n^2)$. L'espace mémoire nécessaire est aussi en $O(n^2)$ quel que soit le nombre d'arêtes. La représentation suivantes sera plus économe.

# 3 Représentation par les listes d'adjacence

**Définition 28** *La représentation de $G$ par listes d'adjacence consiste à représenter $G$ par un vecteur $L$ indexé par $X$ tel que pour tout sommet $x$, $L[x]$ soit la liste de ses successeurs dans $G$.*

**Remarque :**

- Ici, le terme vecteur est à prendre au sens mathématique. En pratique, on a tout intérêt à utiliser un dictionnaire en python. En utilisant un dictionnaire, nous bénéficions à la fois d'un accès direct à tous les successeurs d'un sommet, et aucune restriction n'est imposée sur les étiquettes des sommets (nombre non consécutifs, caractères, chaînes, etc). C'est donc la représentation que nous adopterons en TP.
- Cette définition s'adapte au cas des graphes orientés ou non. Dans le cas d'un graphe non orienté, il faut bien vérifier que si $y$ appartient à $L[x]$, $x$ doit appartenir à $L[y]$.

Exemple :

- Le graphe orienté $G_1$ sera donc représenté par

$$G1 = \{1 : [5], 2 : [1, 4], 3 : [2], 4 : [3], 5 : [2, 4]\}.$$

- Le multigraphe non orienté $G_2$ sera représenté par la liste d'arêtes

$$G2 = \{1 : [2, 5, 5], 2 : [1, 3, 4, 4], 3 : [2, 3, 4], 4 : [2, 2, 3, 5], 5 : [1, 1, 4]\}.$$

**Proposition 7** *Pour un graphe à n sommets et m arcs, l'espace mémoire utilisé est un* $O(n+m) = O(max(n,m))$.

Le parcours des sommets du graphe se résume alors à :

```
for sommet in G:
```

Le parcours des voisins du sommet est cette fois bien en $O(d(x))$ ou $O(d^+(x))$.

```
for succ in G[sommet]:
   {traiter succ}
```

Ces performances sont telles que nous n'utiliserons en TP que la représentation par listes d'adjacence.

Tous les algorithmes présentés dans ce cours sont donnés en pseudo-code. Leur adaptation en python à l'aide des structures de données adaptées sera effectuée en TP. C'est néanmoins une bonne idée de rafraîchir dès à présents vos connaissances sur la manipulation des listes, des dictionnaires et des ensembles qui seront nos structures de données les plus usuelles.

# 4 Représentation des chemins et des arborescences.

- Le chemin $(x_0, x_1, ...x_n)$ peut être représenté par la liste $[x_0, x_1, ..., x_n]$ ou par un tableau des pères donnant pour chaque sommet du chemin le nom de son prédécesseur dans le chemin. Plus précisément :
  - Si $x$ n'appartient pas au chemin alors $Pere[x] = 0$
  - $Pere[x_0] = x_0$ (l'origine est son propre père)
  - Pour les autres sommets du chemin, $Pere[x_{k+1}] = x_k$.

La propriété d'unicité du père dans une arborescence permet également de représenter une arborescence à l'aide d'un tableau des pères tel que :

- $Pere[r] = x_0$ (la racine est son propre père)
- Pour les autres sommets $x$ de l'arborescence, $Pere[x]$ est le numéro de son père dans l'arborescence.
- Ce tableau pourra être représenté en python à l'aide d'un dictionnaire.

L'arborescence de la figure 1.17, enracinée en 4, peut être représenté par son tableau des pères :

$$\{1:2, 2:4, 3:2, 4:4, 5:7, 6:7, 7:4\}$$

(Le père de 1 est 2, le père de 2 est 4, les père de 3 est 2, 4 est son propre père etc.)

# Chapitre 3

# Parcours d'un graphe

Beaucoup de problèmes sur les graphes nécessitent un examen exhaustif des sommets ou des arcs du graphe, on en verra des exemples dans la suite : recherche de chemin, tri topologique, étude de la connexité, etc.
Nous distinguerons deux types de parcours, le parcours en profondeur et le parcours en largeur.
Terminologie : On parle indifféremment de parcours, d'exploration ou de visite de graphe, le terme anglo-saxon est "Graph Search".
Comme nous choisissons une représentation par liste d'adjacence, le même algorithme de parcours pourra être utilisé avec les graphes orientés ou non orientés. Nous verrons que les différences apparaîtront au moment des applications du parcours (à la recherche de cycles ou à l'étude de la connexité notamment).

## 1 Parcours en profondeur (Depth First Search ou DFS)

Le parcours en profondeur consiste, à partir d'un sommet donné à suivre un chemin le plus loin possible, sans passer deux fois par le même sommet. Quand on rencontre un sommet déjà visité, on fait marche arrière pour revenir au sommet précédent et explorer les chemins ignorés précédemment.

Cet algorithme se conçoit naturellement de manière récursive.
Pour savoir si un sommet a déjà été visité, on utilise un ensemble $Visite$ contenant les sommets déjà visités.
Le parcours en profondeur peut alors être décrit récursivement de la manière suivante :

```
Visite est initialisé à l'ensemble vide;
Profond (G,x) :  {parcours en profondeur du graphe G à partir du sommet x}
Début
x est visité
{Traiter x en première visite}
Pour chaque voisin y de x faire
      Si y n'est pas visité alors
         profond(G,y)
```

```
      {Sinon on détecte une revisite de y};
{Traiter x en dernière visite}
fin.
```

Vous trouverez sur Moodle une illustration complète du parcours en profondeur des graphe $G_1$ et $G_3$ suivant.

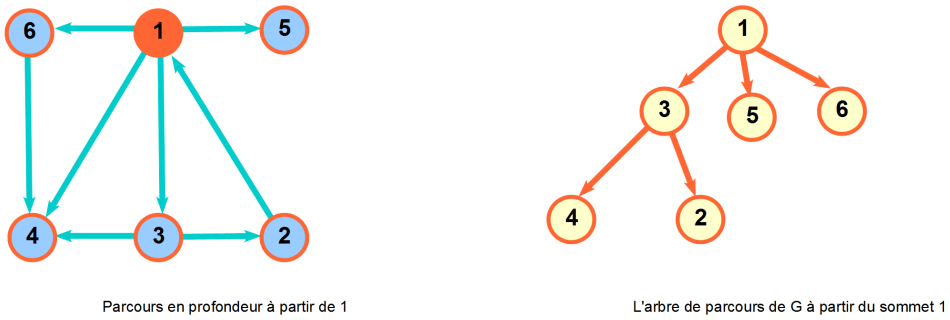



FIGURE 3.1 – Le graphe $G_3$ et son arborescence de parcours en profondeur

## 1.1 Ordre de parcours en première et dernière visite

Dans les applications on traitera les sommets $x$ rencontrés soit en première visite (quand on commence le parcours en profondeur de $x$) soit en dernière visite (quand ce parcours est terminé et qu'on remonte au sommet qui a appelé récursivement $profond(G, x)$).

L'ordre dans lesquels les sommets sont traités est alors respectivement appelé ordre de parcours en première ou en dernière visite.
Le parcours en profondeur du graphe $G_1$ donne en première visite l'ordre $[1, 5, 2, 4, 3]$ et en dernière visite l'ordre $[3, 4, 2, 5, 1]$.
Le parcours en profondeur du graphe $G_3$ donne en première visite l'ordre $[1, 3, 2, 4, 5, 6]$ et en dernière visite l'ordre $[2, 4, 3, 5, 6, 1]$.
On constate sur cet exemple que l'ordre de parcours en dernière visite n'est pas l'ordre inverse de l'ordre de parcours en première visite.

## 1.2 Parcours en profondeur généralisé à tout le graphe

Pour parcourir tous les sommets du graphe $G$, on itère des parcours en profondeur à partir de $x$ tant qu'il existe un sommet $x$ non visité :

```
initialiser Visite à l'ensemble vide;
Pour tout sommet i de G faire
  Si i n'est pas visité alors
     profond(G,i);
```

Vous trouverez sur Moodle une illustration complète du parcours en profondeur généralisé du graphe $G_4$ suivant.

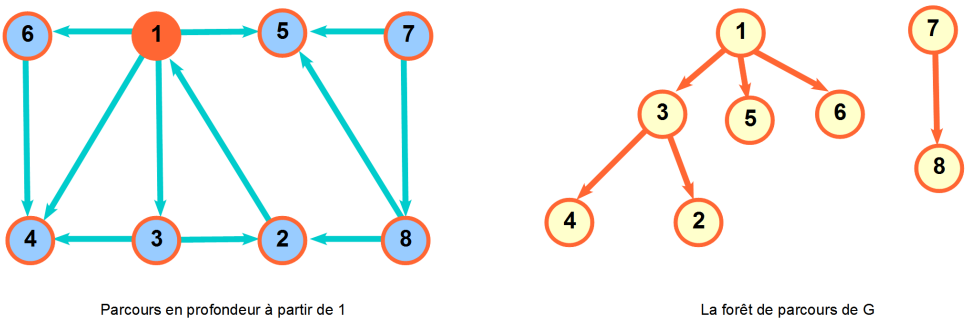



FIGURE 3.2 – Le graphe $G_4$ et sa forêt de parcours en profondeur

### 1.3 Arborescence de parcours en profondeur

Le parcours en profondeur du graphe $G$ à partir d'un sommet $x$ permet de définir une arborescence de visite $A$.

**Définition 29** *L'arborescence de parcours en profondeur de $G$ à partir du sommet $x$ est un enraciné en $x$, orienté et tel qu'il existe un arc $(x, y)$ dans $A$ ssi l'appel à $profond(G, x)$ a engendré récursivement un appel à $profond(G, y)$.*
*Les sommets de l'arborescence de parcours sont tous les sommets de $G$ accessibles à partir de $x$.*

On associe alors au parcours complet du graphe $G$ une forêt de visite contenant tous les sommets de $G$.

## 2 Parcours en largeur (Breadth First Search ou BFS)

Le parcours en largeur est un parcours plus "prudent". Au lieu de s'aventurer le plus loin possible comme dans le parcours en profondeur, on va commencer par visiter d'abord tous les voisins de $x$, c'est-à-dire les sommets à une distance 1 de x, puis les voisins des voisins i.e. les sommets à une distance 2 etc. Pour cela, au fur et à mesure qu'on rencontre de nouveaux sommets (non encore visités,) on mémorise leurs voisins dans une file d'attente $F$ pour une visite prochaine.
$Visite$ est toujours l'ensemble des sommets visités.
$F$ est la liste des sommets qu'on devra prochainement visiter.

```
initialiser Visite à l'ensemble vide;
largeur (G,x) : {parcours en largeur du graphe G à partir du sommet x}
F:=[x];Visite[x]:=vrai
Tant que  F n'est pas vide  faire
Début
  considérer y la tête de F (et l'enlever de F)
  {Traiter y}
```

```
pour chaque successeur z de y Faire
si z n'est pas visité alors
Debut
z est maintenant visité
ajouter z à la fin de la liste F
Fin
Fin
```

Vous trouverez sur Moodle une illustration complète du parcours en largeur des graphe $G_1$ et $G_3$ à partir du sommet 1.

## 2.1 Ordre de parcours en première et dernière visite

L'ordre dans lesquels les sommets sont traités est alors appelé ordre de parcours en largeur. Lors du parcours en largeur du graphe $G_1$ à partir du sommet 1, les sommets sont visités dans l'ordre :

$$[1, 5, 2, 4, 3].$$

Lors du parcours en largeur du graphe $G_3$ à partir du sommet 1, les sommets sont visités dans l'ordre :

$$[1, 3, 4, 5, 6, 2].$$

## 2.2 Parcours en profondeur généralisé à tout le graphe

Pour parcourir tout les sommets du graphe $G$, on itère là encore des parcours en largeur à partir de $x$ tant qu'il existe un sommet $x$ non visités.

## 2.3 Arborescence de parcours en largeur

Le parcours en largeur du graphe $G$ à partir d'un sommet $x$ permet de définir une arborescence de visite $A$.

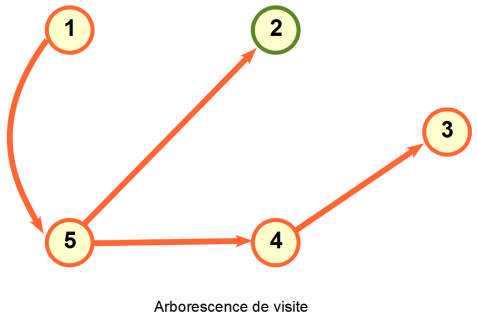



FIGURE 3.3 – Arborescence de parcours du graphe $G_1$ en largeur à partir de 1

**Définition 30** *L'arborescence de parcours en largeur de $G$ à partir du sommet $x$ est un arbre enraciné en $x$, orienté et tel qu'il existe un arc $(y, z)$ dans $A$ ssi le traitement de $y$ ajoute le sommet $z$ dans la liste d'attente $F$.*
*Les sommets de l'arborescence de parcours sont tous les sommets de $G$ accessibles à partir de $x$.*

On associe alors au parcours complet du graphe $G$ une forêt de visite contenant tous les sommets de $G$.

# 3 Complexité

Le tableau $Visite$ est initialisé en $O(n)$,
Le parcours est appelé exactement une fois pour chaque sommet $x$ de $G$ et a pour complexité $O(1)$ pour le traitement de $x$ et $O(d(x))$ pour l'exploration des successeurs de $x$.
La complexité d'une visite complète de $G$ est donc pour les deux parcours de

$$\begin{aligned} C(n) &= O(n) + \sum_{x=1}^{n}(d(x) + O(1)) \\ &= O(n) + \sum_{x=1}^{n} d(x) \\ &= O(n) + O(m) \\ &= O(max(n, m)). \end{aligned}$$

# 4 Applications des parcours

## 4.1 Classification des arcs

- Considérons le cas du parcours en profondeur d'un graphe $G$ orienté. Si on ajoute à la forêt de visite tous les arcs du graphe $G$, on en déduit une classification des arcs :

  (1) Un arc $(x, y)$ de $G$ est dit arc **couvrant** si $(x, y)$ est encore un arc de la forêt de visite de $G$.
  (2) Un arc $(x, y)$ de $G$ est dit arc **en avant** si il existe un chemin de $x$ à $y$ dans la forêt de visite de $G$.
  (3) Un arc $(x, y)$ de $G$ est dit arc **en arrière** si il existe un chemin de $y$ à $x$ dans la forêt de visite de $G$.
  (4) Tous les autres arcs de $G$ sont dits arcs croisés ou **arcs traversiers**.
- Dans le cas du parcours en largeur d'un graphe orienté, il n'y a plus d'arcs en avant donc il ne reste que trois types d'arcs. En effet si l'arc $(x, y)$ existe dans $G$ et que lors de la visite de $x$, $y$ est un voisin de $x$ non visité alors $x$ placera $y$ dans $F$ donc l'arc sera couvrant.
- Dans le cas du parcours en profondeur ou largeur d'un graphe non orienté, il n'y a plus d'arcs transverses et on ne distingue plus en avant ou en arrière, il ne reste donc plus que deux types d'arcs appelés couvrant et en arrière.
- L'existence d'un cycle est caractérisé par la présence d'un arc en arrière.

Il existe de nombreuses applications des parcours en profondeur. Tous les algorithmes qui suivent seront mis en oeuvre en TP en copiant un des parcours et en l'adaptant (en général il suffit d'ajouter une ou deux structures de données et d'écrire leur initialisation et leur mise à jour...)

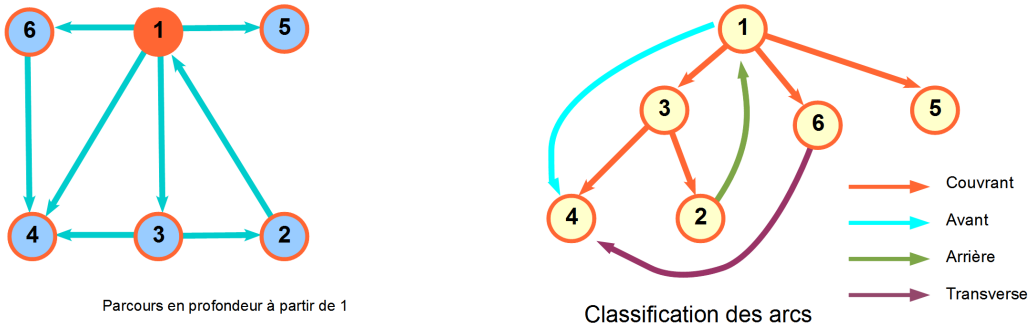



FIGURE 3.4 – Le graphe $G_3$ et la classification de ses arcs dans un parcours en profondeur à partir de 1.

## 4.2 Existence de chemin et connexité

(1) On a vu que les deux parcours d'un graphe à partir du sommet $x$ permettent de visiter tous les sommets $y$ accessibles à partir de $x$. Il est donc facile d'adapter ces parcours pour déterminer la liste de tous les sommets accessibles ou pour **tester l'existence d'un chemin entre deux sommets $x$ et $y$.**

(2) Dans le cas d'un graphe non orienté, le graphe est **connexe** ssi le parcours à partir du sommet 1 visite les $n$ sommets. Si le graphe n'est pas connexe, un parcours à partir du sommet $x$ permet de déterminer la composante connexe de $x$.

## 4.3 Plus courts chemins

Il s'agit ici de déterminer le plus court chemin en "nombre de sauts" entre deux sommets $x$ et $y$ et non du plus court chemin dans un graphe valué que nous étudierons plus loin.

Le parcours en profondeur, trop "aventurier" n'assure pas la minimalité du chemin obtenu. L'avantage d'un parcours en largeur est que du fait de l'ordre de visite des sommets, (les voisins de $x$, les sommets à une distance 2, puis 3 etc.) la première fois que l'on rencontrera $y$ ce sera par un chemin de longueur minimale.
On va en un seul parcours calculer le plus court chemin de $x$ vers tous les sommets $y$ accessibles à partir de $x$. On utilise un dictionnaire $Dist$ tel que $Dist[y]$ est la longueur du plus court chemin de $x$ vers $y$ (et il n'y a pas de clé y si $y$ n'est pas accessibles). Afin de retrouver le chemin, on utilise un dictionnaire des pères tel que Pere[y] est le prédécesseur de $y$ dans le plus court chemin de $x$ vers $y$.

```
  trouvé:=faux;
initialiser Visite au singleton x;
initialiser Dist et Pere à des dictionnaires vides;
F:=[x];
Dist[x]=0;
Pere[x]=x;
```

```
tant que F<>[]  faire
Début
y:=enleveTete(F);
Pour chaque successeur z de y Faire
si z n'est pas visité alors
Debut
z est maintenant visité;
ajouterFin(z,F);
Dist[z]=Dist[y]+1;
Pere[z]=y
Fin
Fin
```

## 4.4 Existence de cycles

On a vu que l'existence de cycles est caractérisée par la présence d'arcs en arrière.

- Dans un graphe non orienté, on détecte un arc en arrière (et donc un cycle) en cas de revisite d'un sommet déjà visité (et différent de son père).
- Dans un graphe orienté, les arcs en avant et les arcs transverses provoquent également des revisites. On détecte un arc en arrière (et donc un circuit) en cas de revisite d'un sommet dont le parcours en profondeur n'est pas encore terminé (on a fait la première visite mais pas la dernière). On peut pour cela utiliser le vecteur de visite et considérer que $Visite[x]$ vaut 0 s'il n'a pas été visité, 1 si on a effectué la première visite et 2 quand on a effectué la dernière visite.

## 4.5 Test d'arbre

On sait qu'un arbre est un graphe non orienté, connexe et sans cycle, les méthodes vues précédemment pour tester l'existence d'un cycle et la connexité d'un graphe non orienté permettent donc de vérifier si un graphe est bien un arbre.

## 4.6 Test de bipartisme

Il existe un critère simple pour vérifier qu'un graphe est biparti :

**Propriété 1** *Un graphe non orienté est biparti ssi il n'admet pas de cycle de longueur impaire.*

**Démonstration** .
L'algorithme de recherche de cycle peut être adapté en test de bipartisme. Il suffit pour cela de numéroter alternativement à l'aide de 0 et 1 les sommets visités. Un cycle de longueur impaire est détecté lorsque $x$ revisite un sommet $y$ de même numéro que lui.
Dans la suite, nous distinguerons les algorithmes de traitement des graphes non orientés, des graphes orientés et des graphes valués.

# Chapitre 4

# Problèmes modélisables par des graphes non orientés

Dans tout ce paragraphe $G = (X, E)$ note un graphe non orienté.

## 1 Chemins et circuits eulériens

### 1.1 Le problème d'Euler

Comme dans le cas des ponts de Königsberg, les problèmes eulériens sont les problèmes relatifs aux chemins et circuits passant une et une seule fois par chaque **arête** du graphe. De nombreux problèmes se modélisent par la recherche d'un parcours eulérien :

- les figures traçables sans lever le crayon.
- le problème du facteur chinois (Mai-Ko-Kwan 1962) qui veut faire sa tournée de distribution de courrier en passant une et une seule fois par chaque rue et en revenant à son point de départ.

**Définition 31** *Soit $G = (X, E)$ un graphe non orienté. Un chemin (circuit) eulérien est un chemin (circuit) passant exactement une fois par chaque arête du graphe.*
*Un graphe est dit eulérien ssi il possède un circuit eulérien.*

### 1.2 Théorème d'Euler

**Théorème 3** *Soit $G = (X, E)$ un graphe non orienté connexe.*

- *$G$ admet un circuit eulérien ssi tous ses sommets ont un degré pair.*
- *$G$ admet un chemin eulérien ssi tous ses sommets ont un degré pair sauf deux sommets $a$ et $b$. Alors tous les chemins eulériens de $G$ admettent $a$ et $b$ comme extrémités.*

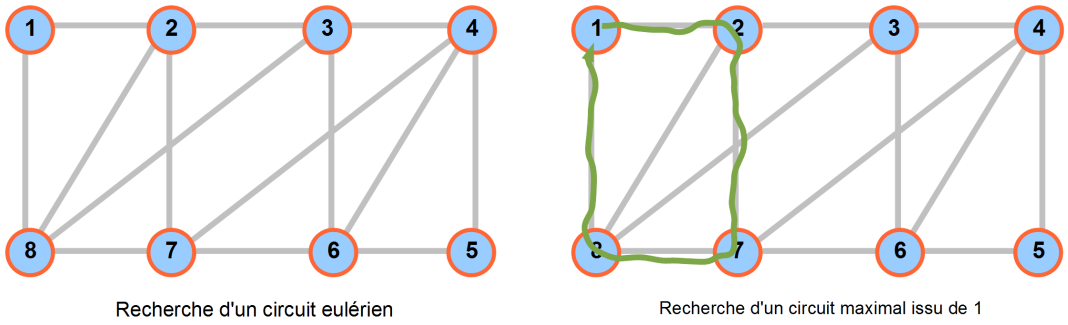
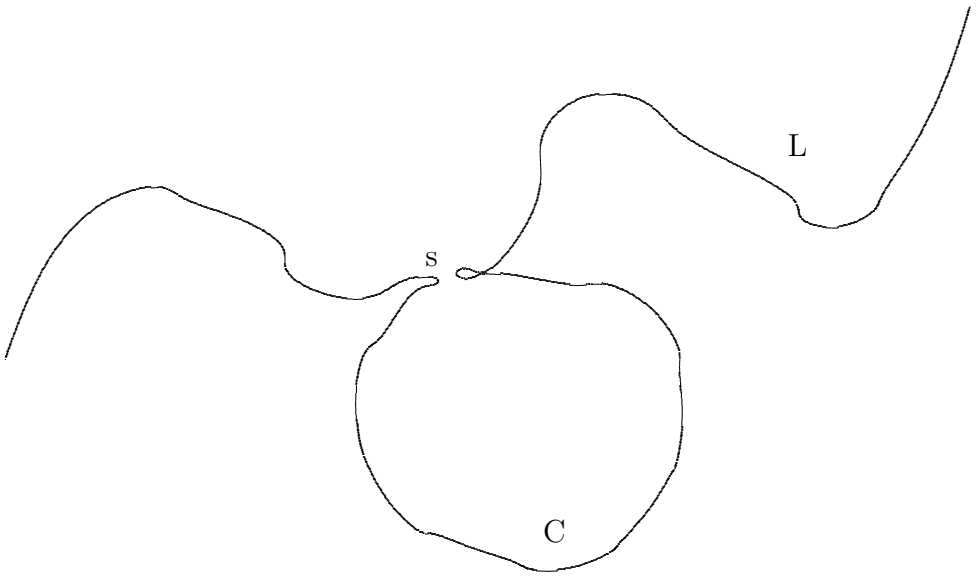



FIGURE 4.1 – Le graphe $G_4$ et un circuit eulérien.

## 1.3 Algorithme de construction d'un chemin eulérien

La démonstration du théorème précédent est constructive, elle fournit donc un algorithme de construction d'un chemin ou circuit eulérien :

- S'il existe deux sommets $a$ et $b$ de degré impair on part de l'un d'eux (disons $a$), sinon on part d'un sommet quelconque, en générale le sommet 1.
- On construit à partir de ce point de départ un chemin simple maximal $L$ (ne passant qu'une fois par une même arête). Dans le premier cas, ce chemin se termine nécessairement en $b$. Dans le second, il est revenu à son point de départ.
- On considère alors le graphe partiel $G_1$ obtenu en enlevant de $G$ les arêtes de $L$.
  $G_1$ n'est plus nécessairement connexe, mais tous ses sommets ont des degrés pairs et par connexité de $G$ chacune des composantes connexes rencontre le chemin $L$.
- Tant que $G_k$ possède une arête, soit $s$ un sommet de $L$ dont le degré dans $G_k$ est strictement positif.
  — Comme précédemment, on construit à partir de $s$ un chemin simple maximal $C$ issu de $s$. On sait que ce chemin est un circuit revenant en $s$.
  — On insère le circuit $C$ dans le chemin $L$ au point d'insertion $s$.




Si $L = (l_0, l_1, ..., l_k = s, ..., l_p)$, ($s$ est le $k$ième sommet de $L$) et que le circuit $C$ est de la forme $(c_0 = s, c_1, ..., c_r = s)$, on obtient alors après insertion le nouveau chemin :

$$L = (l_0, l_1, ..., l_{k-1}, s, c_1, ..., c_{r-1}, s, l_{k+1}, ...l_p).$$

— On considère alors le graphe partiel $G_{k+1}$ obtenu en enlevant de $G_k$ les arêtes de $C$.

- Lorsque $G_k$ n'a plus d'arêtes, $L$ est un parcours eulérien de $G$.

Vous trouverez sur Moodle une illustration complète de la construction d'un circuit eulérien pour le graphe $G_4$.

# 2 Problèmes de coloration

Les algorithmes présentés dans ce paragraphe trouvent leur utilité pratique en modélisant et en permettant de résoudre les problèmes d'exclusion mutuelle et...en théorie des jeux.

Dans tout ce paragraphe, $G = (X, E)$ est un **graphe non orienté.**

## 2.1 Coloration des sommets d'un graphe

## 2.2 Définition. Nombre chromatique

**Définition 32** *Une k coloration des sommets d'un graphe est une coloration des sommets du graphe à l'aide de k couleurs, telle que pour toute arête $(x, y)$ de $G$, $c(x) \neq c(y)$. (i.e. telle que deux sommets adjacents du graphe n'ont jamais la même couleur).*

**Proposition 8**
*1) Si $G = (X, E)$ est un graphe k-parti, alors en associant une couleur différente à chaque $X_i$ on obtient une k-coloration de G.*
*2) Si G est un graphe complet à n sommets, les seules colorations de G sont des n-colorations*

*associant une couleur différente à chaque sommet.*
*3) Si $G$ contient une clique de taille $p$ (sous-graphe complet) alors il ne peut être colorié avec moins de $p$ couleurs.*

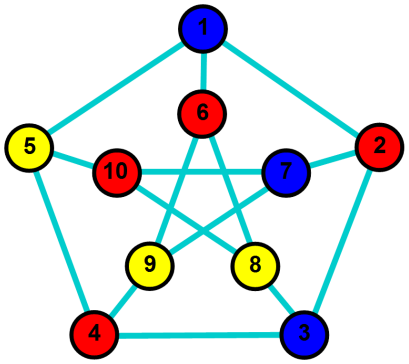



FIGURE 4.2 – Une 3 coloration des sommets du graphe de Petersen.

**Définition 33** *On appelle nombre chromatique d'un graphe $G$, le plus petit entier $k$ tel que $G$ admette une $k$ coloration. On le note $\gamma(G)$.*

On peut alors résumer les définitions et propositions précédentes :

**Proposition 9**
*1) Pour un graphe à $n$ sommets on a bien sûr*

$$1 \leq \gamma(G) \leq n.$$

*2) Si $G$ est $k$ parti alors $\gamma(G) \leq k$.*
*3) Si $G$ contient une clique de taille $p$ alors $\gamma(G) \geq p$.*

**Exercice :** Montrer que si $c_n$ est un cycle à $n$ sommets $n \geq 2$, alors $\gamma(c_n) = 2$ si $n$ est pair et 3 si $n$ est impair.

## 2.3 Exemples d'applications

(1) **Le problème des quatre couleurs :**
Le célèbre problème des quatre couleurs consiste à essayer de montrer que toute carte de géographie peut être coloriée à l'aide de seulement 4 couleurs de sorte que deux pays voisins n'aient pas la même couleur. Etudié depuis 1880 par Guthrie, il a été résolu par l'affirmative par Appel et Haken en 1976 en se ramenant à l'étude de 1509 cas particuliers explorés par ordinateur. L'impossibilité pratique d'une vérification humaine a fait couler beaucoup d'encre...

**Remarque :** Il est facile de voir que quatre couleurs sont nécessaires lorsqu'on a une 4-clique (l'effet Luxembourg).

(2) **Stockage de produits chimiques (où l'organisation de soirées)**
On crée le graphe d'incompatibilité où les sommets sont les invités et où il y a une arête entre deux sommets si les invités ne doivent pas se rencontrer. Le nombre chromatique est le nombre minimum de soirées à organiser pour inviter tout le monde sans commettre d'impair (ou le nombre minimum d'entrepôts différents pour stocker des produits chimiques incompatibles).

(3) **Un problème d'emploi du temps**
La vie est ainsi faite qu'il faut bien que des enseignants fassent cours à des étudiants. Mais les cours sont incompatibles (ne peuvent avoir lieu en même temps) s'il possèdent un même enseignant ou un même étudiant. Une $k$ coloration du graphe de contrainte donne un emploi du temps réalisable et le nombre minimum de salles nécessaires pour sa mise en place.

## 2.4 Un premier algorithme naïf de coloration

Cet algorithme est simple à comprendre et facile à mettre en oeuvre mais...gourmand en couleurs !
L'idée est élémentaire. On attribue à chaque sommet la plus petite couleur non encore attribuée à un de ses voisins.

La coloration est représenté par un tableau (un dictionnaire) tel que $color[x]$ soit la couleur attribuée au sommet $x$ s'il est coloré et tel qu'il n'y pas de clé $x$ dans le cas contraire.

```
DEBUT
{On initialise color au dictionnaire vide}
{On détermine la couleur de chaque sommet}
Pour tout sommet x Faire
  Début
  {Chercher la plus petite couleur non attribuée à un voisin de x,
pour cela on construit l'ensemble S des couleurs de ses voisins}
  et on calcule le plus petit entier k non présent dans S}
on attribue cette couleur au sommet x
  Fin
FIN
```

On a un algorithme en $\sum_x d(x)$ donc en $O(m)$, c'est à dire peu coûteux. Mais il peut engendrer un nombre important de couleur en fonction de l'ordre dans lequel les sommets sont écrits.
Exercice : Trouver une coloration naïve du graphe de Petersen.

## 2.5 Un algorithme glouton de coloration

Lorsqu'on demande à un enfant de colorier un graphe en respectant la contrainte de ne pas colorier deux sommets adjacents de la même couleur il utilise souvent l'algorithme suivant. On le qualifie de glouton car il consiste à prendre les couleurs une par une en coloriant à chaque

fois un ensemble maximal de sommets.

**Définition 34** *On appelle noyau de G un ensemble maximal de sommets non adjacents deux à deux. Cette algorithme consiste donc à attribuer à chaque couleur un noyau de sommets.*

Le calcul d'un noyau peut être effectué de la manière suivante :

```
DEBUT
(* Soit L la liste des sommets restants à colorier *)
N est initialisé à l'ensemble vide;
Tant que L n'est pas vide faire
  x:=tete(L);
  ajouter(x,N);
  Enlever tous les voisins de x de L;
FIN
```

L'algorithme de coloration a alors la forme suivante :

```
DEBUT
c:=1; (* La première couleur *)
S est initialisé à l'ensemble {1..n}; (* les sommets  à colorier *)
Tant que S n'est pas vide faire
  Calculer un noyau à partir de S;
  Enlever les sommets correspondants de S;
  Colorier tous les sommets du noyau
      à l'aide de la couleur c;
  c++;
FIN
```

## 2.6 Un algorithme optimal mais non polynomial

- Si on souhaite une coloration optimale (avec le moins de couleurs possibles), une idée consiste à utiliser le principe suivant :
- On part du nombre de couleurs $c$ obtenu par la coloration naïve par exemple, et on cherche, (tant qu'on trouve des solutions), à obtenir une coloration avec une couleur de moins.
- Pour cela il faut explorer toutes les combinaisons de coloration (on peut tout de même imposer la couleur du premier sommet qui ne change rien à l'existence d'une $k$-coloration). Pour les parcourir toutes, on utilise un algorithme de backtracking : Pour tous les sommets (sauf le premier) on va essayer une à une les $k$ couleurs possibles et chercher à colorer tous les sommets suivants. Si toutes les couleurs ont été essayées sans succès, on revient au sommet précédent (retour en arrière ou backtracking) et on passe à la couleur suivante.

**Propriété 2** *Comme chaque sommet peut prendre n couleurs, la recherche d'une k coloration par exploration complète est donc dans le pire cas en $n^n$ donc surexponentielle ! La croissance d'une telle complexité limite ce type d'algorithme à de très petits graphes (quelques sommets).*

Il semble donc que nous n'ayons le choix qu'entre des méthodes non optimales ou trop complexes. C'est la dure réalité ! Le problème consistant à "Trouver une coloration de $G$ à l'aide du nombre minimal de couleurs $\gamma(G),$" est ce qu'on appelle en algorithmique un problème "difficile" cela signifie qu'on n'en connaît pas actuellement d'algorithme efficace et on conjecture qu'il n'en existe pas !

En pratique, les meilleurs résultats pour les "gros" graphes sont actuellement obtenus à l'aide de **méthodes heuristiques.** Notre première rencontre avec un problème difficile (il en existe beaucoup d'autres) va nous donner l'occasion (en complément de cours) de présenter ces algorithmes.

## 2.7 Méthodes heuristiques

*Aucune connaissance de ce paragraphe n'est exigible à l'examen. Il peut donc être ignoré en première lecture.*

Les méthodes heuristiques donnent des solutions réalisables, prenant en compte la recherche d'une "bonne" solution mais n'assurant pas l'optimalité (on veut se rapprocher de l'intuition d'un bon joueur d'échec).

Le génie industriel a beaucoup développé les méthodes heuristiques car il a à la fois besoin d'améliorer des performances (sans que l'optimalité soit une nécessité), qu'il traite des problèmes de grande taille, et ne dispose pas toujours du temps nécessaire à une longue recherche préalable à la décision. Pour dire les choses simplement, un industriel n'a pas besoin de "la meilleure solution" mais il veut une solution "meilleure que les autres" et surtout "avant les autres".
Le principe général d'une heuristique est le suivant :

- On part d'une solution $s$ donnée de coût $c$ et on cherche "localement" à modifier cette solution pour en diminuer le coût. Il reste à préciser ce que signifie "localement".
- On définit **un voisinage** d'une solution $s$ comme l'ensemble des solutions obtenues à partir de $s$ à l'aide de certaines "transformations" des données.
- Si on autorise toutes les transformations, on obtient le voisinage complet c'est à dire l'ensemble de toutes les solutions, mais on sait qu'il n'est pas explorable polynomialement.
- On se restreint donc en pratique à certaines transformations définissant un voisinage explorable en $n^2$ ou $n^3$.

Dans une recherche locale simple on répète le calcul du voisinage de $s$ et la recherche de la solution $s'$ de coût minimum dans le voisinage de $s$ tant que le coût de $s'$ est inférieur à celui de $s$. On construit donc ainsi une suite de solutions de coût strictement décroissant et on s'arrête lorsqu'on a atteint un minimum sur le voisinage. Cette méthode est meilleure en moyenne que les méthodes naïves mais a le même comportement dans le pire cas. Son inconvénient est bien sûr que cette méthode se fait piéger dans les minimums locaux.

La méthode suivante cherche à échapper à ces minimums. Pour éviter le piège d'un minimum local, on doit parfois s'autoriser à augmenter temporairement le coût. En général une balle de

tennis, grâce a ses rebonds descend plus bas qu'un caillou si on la lance dans un terrain de creux et de bosses. C'est une bonne leçon : **Il faut donc laisser rebondir !**
**Le recuit simulé :** Le principe général du recuit simulé est le suivant : Partant d'une solution initiale $s$ on va passer à une solution voisine $s'$ suivant le principe suivant :

- On calcule la variation de coût $d$ entre les deux solutions.
- Si $d < 0$ on effectue le changement, qui améliore le coût (la balle descend).
- si $d > 0$ va-t-on autoriser ce rebond ? peut-être mais d'autant moins que la température est basse (notre solution est bonne) et que le rebond est grand. On définit un taux d'acceptation, en général $a = e^{-\frac{d}{T}}$ puis on tire au sort p dans $[0, 1]$ pour une loi uniforme. le rebond $s'$ est accepté si $p < a$.

- On assure la convergence en diminuant $T$ à chaque itération. On arrête lorsque la température atteint un seuil $\epsilon$ fixé.

Cette méthode peut être appliquée au problème de la coloration de graphe en utilisant un recuit simulé sur la méthode naïve. On a vu que l'efficacité de cette méthode dépend de l'ordre dans lequel les sommets sont fournis. On cherche par recuit simulé un ordre des sommets donnant un bon nombre de couleurs.

On va donc appliquer l'algorithme glouton en modifiant l'ordre des sommets d'une solution à l'autre. Le voisinage d'une solution est constitué des ordres des sommets obtenus en permutant deux des sommets $i$ et $j$.
La fonction de coût à minimiser est le nombre de couleurs obtenues à partir de cet ordre.
Cette méthode, très simple à programmer et à mettre en oeuvre est réputée pour ses résultats impressionnants sur des graphes jusqu'à 1000 sommets.

## 2.8 Coloration des arêtes d'un graphe

On définit de même le problème de la coloration des arêtes d'un graphe :

**Définition 35** *Une k coloration des arêtes d'un graphe simple est une coloration des arêtes du graphe à l'aide de k couleurs, telle que deux arêtes adjacentes du graphe n'ont jamais la même couleur).*

**Définition 36** *L'indice chromatique $\chi(G)$ est le plus petit nombre k tel qu'il existe une k coloration des arêtes de G. (A ne pas confondre avec le nombre chromatique $\gamma(G)$...)*

Ce problème n'est pas traité ici mais sera étudié partiellement en TD.

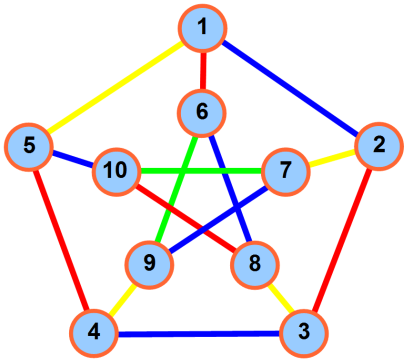



FIGURE 4.3 – Une 4 coloration des arêtes du graphe de Petersen.

# Chapitre 5

# Problèmes modélisables par des graphes orientés

## 1 Ordonnancement et graphes orientés sans circuits

### 1.1 Ordonnancement et graphe de dépendance

Tant d'un point de vue théorique que pratique, les graphes sans circuit jouent un rôle important dans l'étude des ordonnancements de tâches et de l'optimisation. Les applications sont très nombreuses en système, compilation, et en génie industriel.
Le problème de l'ordonnancement (séquentiel ou parallèle) d'un ensemble de tâches est le suivant :
La réalisation globale d'un projet à été décomposée en un certain nombre de tâches élémentaires à réaliser : $t_1, t_2, ...t_n$.
Ces tâches ne peuvent être exécutées dans n'importe quel ordre et doivent respecter un certain nombre de dépendances.

Ce problème peut être modélisé par un graphe orienté appelé graphe de dépendances dont les sommets sont les tâches $X = (t_1, t_2, ...t_n)$ à réaliser et tel qu'il existe un arc entre $t$ et $t'$ si la tâche $t$ doit être terminée avant de pouvoir commencer la tâche $t'$.
En fonction des ressources (des personnes par exemples) dont on dispose on va alors chercher à classer (ordonnancer) ces tâches pour permettre leur exécution.

- Si on ne dispose que d'une personne, elle va exécuter les tâches l'une après l'autre en respectant les dépendances. On parle d'ordonnancement séquentiel.
- Si on dispose d'un nombre de personnes aussi grand que nécessaire on va chercher à exécuter les tâches en parallèle tout en respectant les dépendances. On parle d'ordonnancement parallèle.
- Si on dispose d'un nombre borné de personnes, on produira un ordonnancement parallèle contraint.

Ce problème est bien sûr lié à l'existence de cycles dans le graphe $G$ car si $G$ possède un cycle $(c_1, c_2, ..c_p, c_1)$ la tâche $c_1$ ne peut être réalisée avant $c_2$ qui ne peut être réalisée avant $c_p$ qui ne peut être réalisée avant $c_1$. Il est bien clair que le projet est alors irréalisable.
Nous allons donc commencer par établir quelques propriétés essentielles des graphes sans cir-

cuits.

## 1.2 Propriétés des graphes sans circuit

**Proposition 10** *(1) Si $G$ est sans circuit, tous ses sous-graphes sont sans circuit.*

*(2) Si $G$ est sans circuit, le graphe inverse de $G$, obtenu en inversant l'orientation de tous les arcs de $G$ est encore sans circuit.*

*(3) Un graphe est sans circuit ssi tous ses chemins sont élémentaires.*

Un graphe sans circuit possède des sommets remarquables, on rappelle que dans un graphe orienté

**Définition 37**

*Une source est un sommet dont le degré entrant est non nul.*

*Un puits est un sommet dont le degré sortant est nul.*

**Théorème 4** *Tout graphe sans circuit possède une source et un puits.*

## 1.3 Tri topologique et ordonnancement séquentiel

**Définition 38** *Mathématiquement, on appelle tri topologique d'un graphe orienté $G = (X, E)$ toute numérotation des sommets respectant l'ordre des arcs :*
*En informatique, on considère souvent un tri topologique comme une fonction qui à un graphe associe une liste de ses sommets triés de manière à ce que si $G$ contient un arc $(x, y)$ alors $x$ soit avant $y$ dans la liste.*

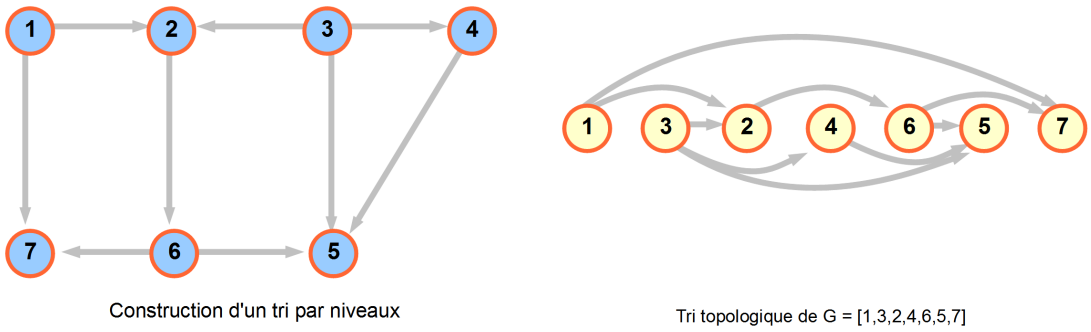



FIGURE 5.1 – Le graphe $G_5$ et un tri topologique.

**Théorème 5** *Un graphe orienté admet un tri topologique ssi il est sans circuit.*

Si $G$ n'a pas de circuit, c'est aussi le cas de tous ses sous-graphes. Tant qu'on n'a pas traité tous les sommets de $G$, on sait que $G$ possède une source $x$ (donc un sommet sans prédécesseur). $x$ peut donc être traité. On remplace alors $G$ par le sous-graphe obtenu en enlevant $x$ et ses

arêtes incidentes. Il n'y a plus qu'à itérer.

Réciproquement si on ne trouve pas de source alors que l'on n'a pas traité tous les sommets de $G$ c'est que $G$ possède un cycle.

La mise en oeuvre de cet algorithme ne nécessite que de déterminer une source à chaque étape. Pour cela on calcule le vecteur $Degre$ tel que $Degre[x]$ soit le degré entrant de $x$ dans $G$. et on définit une liste $S$ des sources trouvées. L'initialisation ne pose pas de problème.
La mise à jour est la suivante : après avoir traité une source $x$, on décrémente le degré entrant de tous ses successeurs dans $G$. si un de ces degré devient nul, c'est qu'on a trouvé une nouvelle source que l'on peut ajouter à $S$. On itère tant qu'on a des sources. Si on n'a plus de sources avant d'avoir traiter tous les sommets c'est que $G$ possède un cycle, sinon l'ordre de traitement des sommets donne un tri topologique donc un ordonnancement séquentiel.

On obtient l'algorithme suivant :

```
Fonction tri topologique G:graphe -> T liste des sommets triés
Début
S:={};T:=[];
Pour tout sommet x faire
Degre[x]:=d^-(x) dans G;
Si Degre[x]=0 alors ajouter x à S;
{S contient alors toutes les sources de G}

tant que S non vide faire
x:=enleverTête(S);
ajouterFin(x,T)
{Mettre à jour les degrés}
Pour chaque successeur y de x dans G faire
Degre[y]:=Degre[y]-1;
si Degre[y] = 0 alors ajouterFin(y,S)
fin
fin.
```

**Propriété 3**

*A chaque étape on traite le premier élément de la liste des sources mais on pourrait traiter n'importe lequel des éléments de S. Il n'existe donc pas un unique tri topologique.*

*Le tri topologique du graphe de dépendance fournit un ordonnancement séquentiel du problème initial.*

## 1.4 Tri par niveaux et ordonnancement parallèle

On souhaite cette fois-ci paralléliser au maximum l'exécution des tâches dans le but de réduire la durée du projet.

**Théorème 6** *"Partition en niveaux" :*
*$G = (X, E)$ est un graphe sans circuit ssi $X$ admet une partition en niveaux i.e. une partition de l'ensemble des sommets $X$ en $X = N_0 \cup ... \cup N_p$ telle que $\forall i \in [1, p]$, les sommets du niveau $N_i$ peuvent être exécutés en parallèle à l'étape $i$ (n'ont plus de prédécesseurs non réalisés à l'étape $i$.)*

**Démonstration**
Il suffit pour cela d'adapter l'algorithme de tri topologique en traitant simultanément toutes les sources de $G$. On définit deux listes de sources $N1$ et $N2$ correspondant à deux niveaux successifs. Le traitement des sources de $N1$ permettant de déterminer les sources du niveau suivant $N2$.

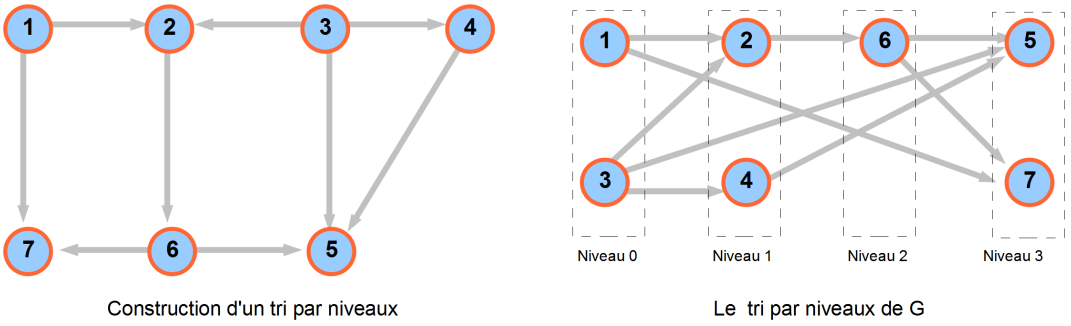



FIGURE 5.2 – Le graphe $G_5$ et son tri par niveaux.

Bien réfléchir aux structures de données adaptées pour représenter les niveaux et le tri !

```
Fonction tri par niveau G:graphe -> T liste des niveaux T=[N0, N1,..., Np]
Début
N1:={};T:={};
Pour tout sommet x faire
Degre[x]:=d^-(x) dans G;
Si Degre[x]=0 alors ajouter x à N1;
{S contient alors toutes les sources de G}

tant que N1 n'est pas vide faire
ajouterFin(N1,T);
N2:={};
pour tout x dans N1 faire
{Mettre à jour les degrés des successeurs de x et calculer les nouvelles sources}
Pour chaque successeur y de x dans G faire
Degre[y]:=Degre[y]-1;
si Degre[y] = 0 alors ajouterFin(y,N2)
N1:=N2;
```

**Propriété 4**

*Si on traite les sommets dans l'ordre des niveaux, on obtient un tri topologique. Mais la réciproque n'est pas nécessaire, il existe des tris topologiques ne respectant pas les niveaux.*

*Le tri par niveau du graphe de dépendance fournit un ordonnancement parallèle optimal du problème initial. La taille des niveaux fournit le nombre de ressources nécessaires à chaque étape.*

# Chapitre 6

# Problèmes modélisables par des graphes valués

Nous allons pour terminer ce cours, étudier deux problèmes nécessitant de compléter la définition d'un graphe en définissant une valuation (une fonction de poids) sur les arêtes ou les arcs du graphe. Nous verrons un exemple modélisable par un graphe non orienté valué (la recherche d'arbre couvrant de poids minimum) et un exemple fondamental modélisable par un graphe orienté valué (la recherche de plus court chemin).

## 1 Graphes valués

**Définition 39** *On appelle valuation (ou fonction de poids ou fonction de coût) définie sur le graphe $G = (X, E)$ une application*

$$p : E \to \mathbb{R} \textit{(ce sont les arêtes ou les arcs qui ont des poids et non les sommets).}$$

*Un graphe (orienté ou non) muni d'une valuation est appelé graphe valué. On note donc $p(i, j)$ le poids de l'arête $\{i, j\}$ ou de l'arc $(i, j)$.*

**Définition 40** *Si $p$ est une valuation définie sur $G = (X, E)$ on peut calculer le poids (ou coût) total*

- *du graphe $G$. C'est la somme des poids de ses arêtes :*

$$p(G) = \sum_{l \in E} p(l).$$

- *d'un graphe partiel $G' = (X, E')$ :*

$$p(G') = \sum_{l \in E'} p(l).$$

- *d'un chemin $C = (x_1, x_2, ..., x_n)$.*

$$p(C) = \sum_{i=1}^{n-1} p((x_i, x_{i+1})).$$

## 1.1 Représentation d'un graphe valué

Comme pour un graphe classique, nous allons retrouver les trois représentations d'un graphe par liste d'arêtes, matrice ou listes d'adjacence, qu'il faut modifier un peu pour y ajouter une représentation des valuations des arêtes ou arcs. Nous ne distinguons pas ici le cas d'un graphe orienté ou non et nous parlerons indifféremment d'arête ou d'arc.

Les différentes représentations seront illustrées à l'aide du graphe $G_6$ suivant :

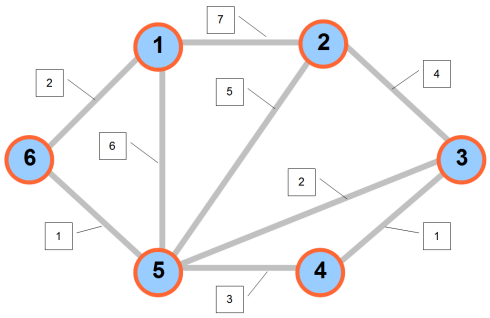



FIGURE 6.1 – Le graphe $G_6$.

### Représentation par liste d'arêtes

**Définition 41** *Une arête ou un arc $(i, j)$ de poids $p(i, j)$ est représentée par un triplet $(i, j, p(i, j))$. La représentation d'un graphe $G$ par sa liste d'arête est donc une liste de triplets.*

Exemple : Le graphe $G_6$ est représenté par sa liste d'arêtes :

$$[[1, 2, 7], [1, 5, 6], [1, 6, 2], [2, 3, 4], [2, 5, 5], [3, 4, 1], [3, 5, 2], [4, 5, 3], [5, 6, 1]].$$

Nous utiliserons cette représentation "naturelle" pour la saisie du graphe mais nous la transformerons aussitôt en matrice ou liste d'adjacence.

### Représentation matricielle

**Définition 42** *La matrice d'adjacence (aussi appelée matrice de poids) de $G$ est la matrice $A$ dont le coefficient d'indice $(i, j)$ est*

- *$p(i, j)$ si l'arc ou l'arête $(i, j)$ appartient à $G$.*

*égal à une valeur conventionnelle (par exemple $+\infty$) si cet arc n'existe pas.*

Exemple : La matrice de poids du graphe $G_6$ est la matrice (symétrique puisque le graphe est

non orienté)

$$P = \begin{pmatrix} +\infty & 7 & +\infty & +\infty & 6 & 2 \\ 7 & +\infty & 4 & +\infty & 5 & +\infty \\ +\infty & 4 & +\infty & 1 & 2 & +\infty \\ +\infty & +\infty & 1 & +\infty & 3 & +\infty \\ 6 & 5 & 2 & 3 & +\infty & 1 \\ 2 & +\infty & +\infty & +\infty & 1 & +\infty \end{pmatrix}.$$

Algorithmiquement, cette représentation permet d'accéder en temps constant au poids d'une arête mais on sait qu'elle est moins performante que la représentation par liste d'adjacence dans de nombreux algorithmes.

Remarque : En python, pour que le poids de l'arête $(i, j)$ soit effectivement à l'indice $(i, j)$ de la matrice, nous laisserons à 0 les lignes et colonnes d'indice 0.

**Représentation par les listes d'adjacence**

On pourrait imaginer une représentation par listes d'adjacence dans laquelle $G[i]$ serait la liste des couples $(j, p(i, j))$ pour tous les successeurs $j$ de $i$. En pratique on préfère utiliser la liste d'adjacence classique (sans poids) et l'associer à la matrice de poids.

Nous ferrons donc le choix de la représentation suivante pour les graphes valués :

**Définition 43** *Nous représenterons un graphe valué par sa liste d'adjacence (classique) associée à sa matrice de poids.*

Finalement $G_6$ sera représenté par

- Sa liste d'adjacence (un dictionnaire) $\{1 : [2, 5, 6], 2 : [1, 3, 5], 3 : [2, 4, 5], 4 : [3, 5], 5 : [1, 2, 3, 4, 6], 6 : [1, 5]\}$
- Sa matrice de poids :

$$P = \begin{pmatrix} +\infty & 7 & +\infty & +\infty & 6 & 2 \\ 7 & +\infty & 4 & +\infty & 5 & +\infty \\ +\infty & 4 & +\infty & 1 & 2 & +\infty \\ +\infty & +\infty & 1 & +\infty & 3 & +\infty \\ 6 & 5 & 2 & 3 & +\infty & 1 \\ 2 & +\infty & +\infty & +\infty & 1 & +\infty \end{pmatrix}.$$

# 2 Arbre couvrant de poids minimum

Dans ce paragraphe, $G = (X, E, p)$ est un graphe non orienté valué.

**Définition 44** *On appelle arbre couvrant de $G$ un graphe partiel de $G$, $T = (X, E')$ qui soit un arbre.*
*Rq : bien noter que tout sommet $x \in X$ doit être "couvert" par une arête de $T$. On appelle "arbre couvrant de poids minimal" un arbre couvrant de poids total minimal.*

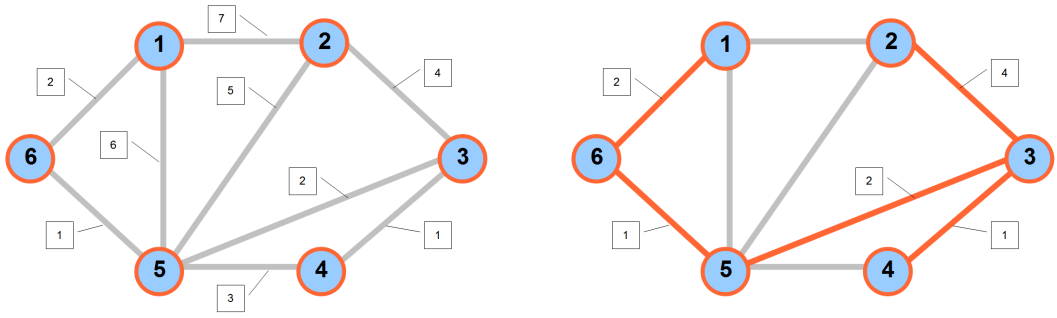



FIGURE 6.2 – Le graphe $G_6$ et un arbre couvrant de poids total 10 (minimal).

Le problème de la recherche d'un arbre couvrant de poids minimal possède de nombreuses applications en optimisation : un arbre couvrant constitue une manière d'assurer la connexité entre les sommets en minimisant le coût de réalisation ou de fonctionnement des liaisons (optimisation de réseaux de distribution (gaz, électricité...) ou de réseaux routiers).
L'existence d'une solution est assurée par le théorème suivant :

**Théorème 7** *Un graphe $G = (X, E)$ admet un arbre couvrant ssi il est connexe.*

Dans la suite on suppose que $G = (X, E, p)$ est connexe.

## 2.1 Algorithme de Kruskal

L'idée de l'algorithme de Kruskal (1956) est la suivante :

(1) Trier les arêtes par poids croissants et les stocker dans une liste $L$.

(2) $T$ est le dictionnaire initialement vide des arêtes de l'arbre couvrant cherché.

(3) A chaque étape on ajoute à $T$ l'arête de poids minimal dont l'ajout n'engendre pas de cycle. (Au cours de la construction, $T$ n'est donc a priori pas connexe).

(4) Si on parvient à la $m = n-1$ ème étape à un graphe à $n$ sommets et $m = n-1$ arêtes on a un graphe sans cycle de $n-1$ arêtes, i.e. un arbre. Comme il est élémentaire, il passe par $n$ sommets et est donc couvrant. Tout sous-graphe de poids strictement inférieur contient des cycles et n'est donc pas un arbre. Finalement $T$ obtenu est bien un arbre couvrant de poids minimal.
Si avant la $m$ième étape il n'y a plus d'arête qui convient c'est que le graphe initial n'était pas connexe.

Exemple :
A chaque étape il est nécessaire de tester si l'ajout d'une arête engendre ou non un cycle, on a vu au paragraphe précédent que cela pouvait être réalisé à l'aide d'un parcours (en largeur par exemple).
A la ième étape $T$ possède $i$ arêtes donc un parcours en largeur de complexité $O(i)$. Il y a $m-i$ arêtes à tester, le pire cas est quand c'est systématiquement la dernière la bonne ce qui donne une étape $i$ de coût $(m-i)*i$ et une complexité totale de $\sum_{i=1}^{m}(m-i)*i = O(m^2)$ (ce qui peut faire jusqu'à $O(n^4)$ pour un graphe complet). On a intéret à ordonner les arêtes par poids.

## 2.2 Algorithme de Prim

Cet algorithme date de 1957. Au contraire de l'algorithme de Kruskal qui traite les arêtes de $G$ en cherchant à ne pas créer de cycle, on va ici chercher à conserver la connexité à moindre coût. Le principe est le suivant :

(1) On choisit un sommet $x$ de $G$ (en général le sommet 1). On initialise $T$ à l'arbre $x$ à 1 sommet et 0 arête.

(2) Soit $C$ l'ensemble (toujours connexe) des sommets couverts par les arêtes de $T$ et on appelle $M$ le complémentaire de $C$ dans $X$ (qu'il est inutile de calculer). À chaque étape on ajoute à $T$ l'arête de coût minimal joignant un sommet de $C$ à un sommet de $M$.

(3) L'arête ajoutée reliant deux composantes connexes différentes ne peut créer de cycle. Au bout de $m = n-1$ étapes $T$ est donc un graphe connexe vérifiant $m = n-1$. C'est donc un arbre couvrant de $G$. Il est de poids minimal.

Si on ne trouve pas de telle arête avant la $m$ième étape c'est que $G$ n'est pas connexe.
Raffinons un peu l'étape importante de cet algorithme :

```
T:={};m:=0;
C:={1};
Tant que  m<n-1 faire
Début
{(1) chercher l'arête u=(y,z) de poids minimal telle que y soit dans C et z dans M}
Ajouterfin(u,T);
Ajouterfin(z,C)
m:=m+1;
Fin
```

Il faut bien sûr raffiner l'étape fondamentale (1). Pour cela on va tenir à jour les listes $C$ et $M$ et calculer deux tableaux $MIN$ et $DIST$, indexés par $M$ et tels que pour tout $y$ dans $M$, si $y$ est accessible depuis C par une arête, alors DIST[y] est le poids minimal d'une arête reliant y à un sommet de C et MIN[y] est le sommet de C réalisant ce minimum. Si $y$ est accessible depuis C par une arête on donne à DIST[y] et MIN[y] des valeurs conventionnelles.
Complexité :
L'initialisation se fait en $O(n)$. A chaque étape il faut :
Calculer le min de DIST en $O(n)$ et mettre à jour tous les successeurs de $z$ en $O(D^+(z))$. Cela conduit donc à un algorithme en $O(n^2)$.

# 3 Recherches de plus courts chemins dans un graphe valué

Dans tout ce paragraphe, $G = (X, E, p)$ est un graphe orienté valué. Néanmoins les algorithmes présentés peuvent être très facilement adaptés au cas d'un graphe non orienté.

## 3.1 Présentation du problème

La recherche d'un chemin de coût minimum entre deux sommets d'un graphe orienté valué est un des problèmes les plus classiques des graphes et les plus riche d'application à l'optimisation en général (c'est ainsi que Mappy trouve l'itinéraire minimisant la consommation d'essence ou le prix des péages ou la distance parcourue) et en particulier en informatique (en particulier le routage dynamique des paquets dans une transmission réseaux).
On rappelle que si $C = (a_1, a_2, ...a_n)$ est un chemin, on appelle coût de ce chemin, la somme des coûts des arcs le constituant.

$$cout(C) = \sum_{i=1}^{n} p(a_i).$$

Le but de ce paragraphe est alors de chercher à construire (s'il en existe) un chemin joignant deux sommets $x$ et $y$ de $G$ et minimisant le coût total du chemin. On parle également de "plus court chemin" entre $x$ et $y$.

## 3.2 Conditions d'existence

Commençons par remarquer qu'il n'existe pas nécessairement un chemin de coût minimum, même s'il existe des chemins joignant $x$ à $y$.
En effet si les poids des arêtes peuvent être négatifs, on appelle circuit absorbant un circuit de coût total négatif. Si un graphe contient un circuit absorbant, alors tout passage par ce circuit fait diminuer le coût du chemin. Il est alors clair qu'il n'existe pas de chemin de coût minimum.

**Théorème 8** *Le problème du plus court chemin admet une solution si y est accessible à partir de x et qu'il n'existe pas de circuit absorbant. S'il n'y a pas de circuit de coût nul, les plus courts chemins sont des chemins élémentaires. S'il y en a, nous nous restreindrons à la recherche de chemins élémentaires.*

Nous allons voir trois algorithmes permettant de construire des plus courts chemins, il en existe quelques autres, ils diffèrent par les hypothèses qu'ils nécessitent et le but recherché :

| Auteurs | Hypothèses | But | Complexité |
|---|---|---|---|
| Dijkstra | poids positifs | chemins entre $s$ et tous ses sommets accessibles | $O(n^2)$ |
| Bellman | graphes sans circuit | chemins entre $s$ et tous ses sommets accessibles | $O(m)$ |
| Floyd | | chemins entre tous les sommets | $O(n^3)$ |
| Dantzig | graphes quelconques | chemin entre deux sommets | $O(n^3)$ |
| Ford | graphes quelconques | chemin entre deux sommets | $O(n.m)$ |

## 3.3 Algorithmes de Dijkstra

Nous allons voir deux versions de l'algorithme de Dijkstra. La première (Dijkstra1) s'applique à tout graphe sans circuit absorbant, sans hypothèse supplémentaire mais n'a pas une très bonne complexité dans le pire cas. Le deuxième algorithme (DijkstraOpt) se limite au cas (fréquent

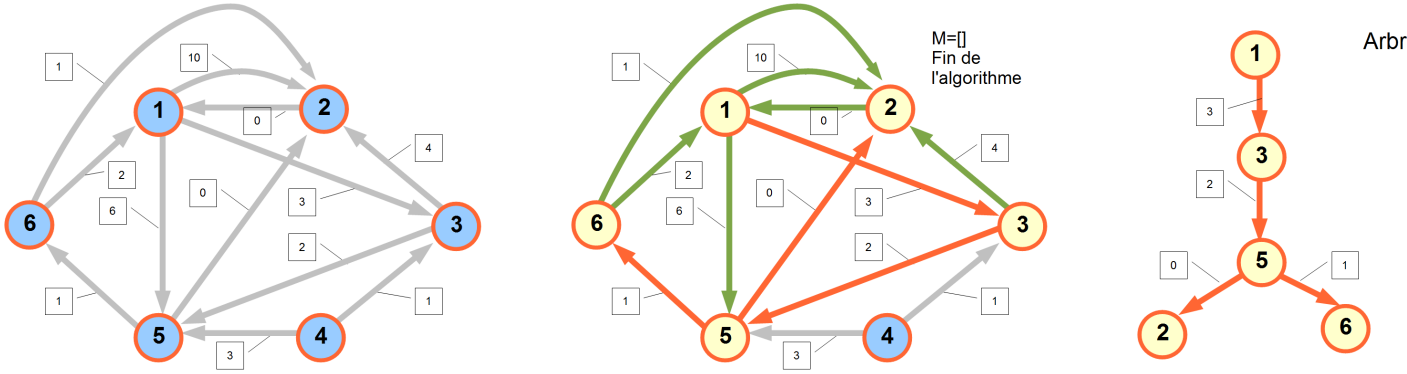



FIGURE 6.3 – Le graphe $G_7$ (à poids positifs) et l'arbre collecteur des plus courts chemins issus du sommet 1 (obtenu par l'algorithme de Dijkstra)

quand la fonction de poids est un côut ou un temps) où toutes les arêtes ont un poids positif. On est alors sûr qu'il n'y a pas de circuit absorbant. cette hypothèse supplémentaire permet de limiter la complexité de l'algorithme.
Dans les deux cas, l'algorithme permet de calculer à partir d'un sommet $s$ les plus courts chemins entre $s$ et tous les sommets accessibles à partir de $s$. (C'est ce qu'on appelle son arbre collecteur).
Version 1 :

```
Algorithme Dijkstra;
DEBUT
{initialisations}
M:={s};
Dist[s]=0
Pere[s]=s
Tant que M<>{} faire
  Début
  x:=enleveTete(M);
  Pour tout successeur  y de x faire
  Début
  d:=D[x]+cout(x,y);
  si d<D[y] alors
  Début
  D[y]:=d;
  P[y]:=x;
  ajouter(y,M)
FIN.
```

Version 2 :

```
Algorithme Dijkstra;
DEBUT
```

```
{initialisations}
M:={};
M:={s};
Dist[s]=0
Pere[s]=s

Tant que M<>{} faire
  Début
  x:=choisir_min(M,d);
  enlever(x,M);
  Pour tout successeur  y de x faire
  Si y est dans M alors
  Début
  d:=D[x]+cout(x,y);
  si d<D[y] alors
  Début
  D[y]:=d;
  P[y]:=x;
  ajouter(y,M)
  Fin
  Fin
  Fin
FIN.
```

**Calcul de la complexité** : L'initialisation est en $O(n)$.
La boucle de traitement va parcourir une et une seule fois chaque sommet accessible à partir de $s$. La boucle "tant que" sera donc parcouru au plus $n$ fois. A la ième étape $M$ contient $n-i$ sommets et donc la complexité de ChoisirMin est en $O(n-i)$. le parcours des successeurs de $x$ est en $O(d+(x))$. La boucle de traitement a donc une complexité de $\sum_{i=1}^{n}(n-i)+\sum_{x\in X} d^{+}x$. (on considère la boucle "tant que" en disant d'une part qu'elle parcourra tous les sommets dans un certain ordre et d'autre part qu'elle traitera des ensembles $M$ de cardinal $n-i$ pour toute valeur de $i$). D'où une complexité de $O(n^2)+O(m)=O(n^2)$.

## 3.4 Algorithme de Bellman

Cet algorithme est utilisé dans les graphes sans circuit dont les arcs ont des poids quelconques. On est ainsi assuré qu'il n'existe pas de circuit absorbant.
Pour la partie théorique, quitte à se restreindre à la composante connexe de $s$ (i.e. à l'ensemble des sommets accessibles à partir de $s$,) on peut supposer que tout sommet est accessible à partir de $s$. L'idée (récursive) de l'algorithme est la suivante : "un plus court chemin de $s$ à $y$ doit passer par un des prédécesseurs $x$ de $s$ et suivra alors le plus court chemin de $s$ à $x$ puis l'arc $(x,y)$." On note $M$ l'ensemble des sommets $x$ pour lesquels on connaît un plus court chemin de $s$ à $x$ et $S=X\backslash M$. On va construire l'ensemble $M$ suivant le principe suivant :

(1) Tant que $S$ est non vide, il existe toujours un sommet $y$ de $S$ dont tous les prédécesseurs sont dans $M$.

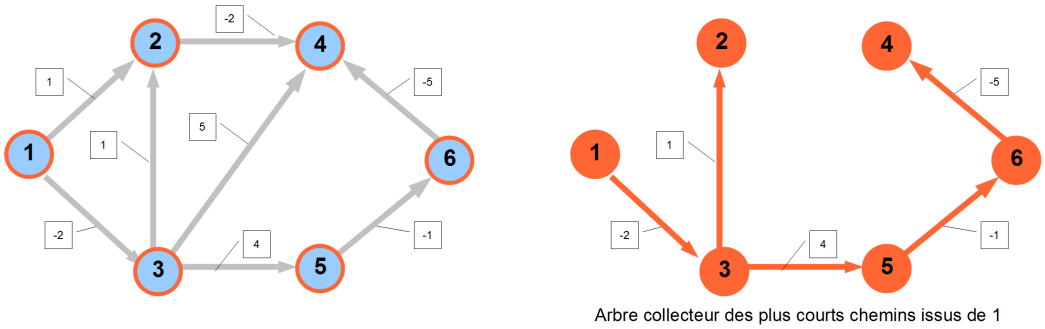



FIGURE 6.4 – Le graphe $G_8$ (sans cycles) et l'arbre collecteur des plus courts chemins issus du sommet 1 (obtenu par l'algorithme de Bellman)

**Démonstration** Par hypothèse $S$ est non vide et chacun de ses éléments admet au moins un prédécesseur puisqu'il est accessible à partir de $s$. Si tout sommet $y$ de $S$ admet un prédécesseur dans $S$ alors on peut construire une suite infinie $(y_n)$ d'éléments de $S$ tels que $y_{i+1}$ est prédécesseur de $y_i$. Comme $S$ est fini cette suite contient nécessairement un circuit ce qui est exclus par hypothèse.
En pratique si un tel sommet n'existe pas c'est qu'on a traité tous les sommets accessibles à partir de $s$.

(2) Pour ce sommet $y$ Soit $P_y$ l'ensemble de ses prédécesseurs (tous dans $M$). Le plus court chemin de $s$ à $y$ peut être calculé par

$$min_{x \in P_y}(dist(s,x) + p(x,y)).$$

$y$ passe alors dans $M$.

(3) L'algorithme s'arrête lorsque $S$ est vide ou qu'on a traité tous les sommets accessibles à partir de $s$.

Exemple
On peut raffiner l'algorithme en tenant à jour deux tableaux indexés par $X$ : $DIST$ tel que $Dist[x]$ est la longueur du plus court chemin entre $s$ et $x$ une fois celle-ci déterminée et $P$ tel que $P[x]$ soit le prédécesseur de $x$ dans ce plus court chemin. Le tableau $P$ servira a expliciter les chemins une fois qu'il seront déterminés. On obtient l'algorithme suivant :

```
{Dist  est initialisé à une valeur conventionnelle infinie et P à 0
S est initialisé à X\S }
Dist[s]:=0;Pere[s]:=s; fini:= faux ;

tant que  non fini faire
Début
fini:=vrai;
  pour tout élément y de S faire
 {On traite tous les y de S dont tous les prédécesseurs sont dans M}
      Oky:=vrai; d:=infini;
```

```
      {Oky vrai ssi tous les prédécesseurs de y sont dans M}
      pour tout prédecesseur x de y faire
         si Dist[x] <> infini
         alors si  Dist[x]+p(x,y)< d alors
         d:= Dist[x]+p(x,y)
         Pere[y]:=x;
         sinon Oky:=false;
         {on peut sortir de la boucle (pour tout prédécesseur..) dès que OKy est faux}
         Si OKy alors {On traite y}
         fini:=false;
         enlever(y,S);Si S=[] alors fini:=vrai;
         Dist[y]:=d
Fin
```

La version présentée ci-dessus est simple a comprendre mais a une complexité en $O(n^3)$ :
A la $i$ème étape, $S$ contient $n-i$ sommets. Dans le pire cas c'est le dernier élément de $S$ qui est le bon ce qui nécessite de parcourir tous les prédécesseurs de tous les éléments de $S$. L'algorithme a donc une complexité de $C(n) = \sum_{i=1}^{n}(\sum_{y=1}^{n-i} d^-(y))$. On arrête le parcours des prédécesseurs dès qu'on en trouve un dans $S$ donc on en parcourt toujours moins de $|M| + 1 = i + 1$. D'où

$$\begin{aligned} C(n) &\leq \textstyle\sum_{i=1}^{n}(\sum_{y=1}^{n-i} i + 1) \\ &= O(\textstyle\sum_{i=1}^{n} O(n^2) = O(n^3) \end{aligned}$$

Cette complexité peut être ramené à $O(m)$ en évitant la boucle de parcours de recherche d'un élément $y$ de $S$ satisfaisant. Il suffit pour cela d'initialiser et de tenir à jour un vecteur PS donnant pour chaque élément de $S$ le nombre de ses prédécesseurs dans $S$. pour ne pas parcourir ce tableau, on tient à jour une liste L des éléments de valeur nulle du tableau PS. Choisir un $y$ qui convient consiste alors a prendre l'élément de tête de $L$ ce qui se fait en $O(1)$. On obtient l'algorithme suivant :

```
{Dist  est initialisé à une valeur conventionnelle infinie et P à 0
Dist[s]:=0;Pere[s]:=s;

{construction du tableau PS et de la liste L}
L:=[];
pour y de 1 à n faire
  Début
   PS[y]:=0
   si y=s alors PS[y]=-1 {on exclus s en lui donnant une valeur <0} sinon
   pour chaque prédécesseur z de y faire
   si z<>s alors PS[y]:=PS[y]+1;
   Si PS[y]=0 alors ajouter(y,L)
  Fin;

tant que  L<>[] faire
```

```
Début
  y:=enleveTete(L);
      pour tout prédécesseur x de y faire
          si  Dist[x]+p(x,y)< Dist[y] alors
         Début
         Dist[y]:= Dist[x]+p(x,y)
         Pere[y]:=x;
         Fin
         {Mise à jour de PS et L}
         {tout successeur de y a un prédécesseur de moins (y) dans S}
         Pour chaque successeur z de y faire
          Debut
          PS[z]:=PS[z]-1;
          Si PS[z]=0 alors ajouter(z,L)
          Fin
Fin
```

La complexité de l'algorithme est alors de $\sum_{y=1}^{n} d^-(y) = O(m)$ pour l'initialisation de PS et L. La partie traitement ayant une complexité de $\sum_{y=1}^{n} d^-(y) + d^+(y) = O(m)$. On a donc une compléxité totale en $O(m)$. donc toujours meilleure que $O(n^2)$.

# Lexique

Entre parenthèse le terme anglais.

**Acyclique (acyclic)** : un graphe orienté est acyclique s'il ne possède pas de cycle. Graphe orienté sans cycle se dit (Direct acyclic graph), souvent contracté en DAG). On utilise aussi acyclique pour un graphe non orienté sans circuit.

**Adjacent (adjacent)** : Deux sommets sont adjacents s'ils sont reliés par un arc ou une arête. On utilise souvent voisin au lieu d'adjacent. Dans un graphe orienté on parle de prédécesseur et de successeurs.

**Arborescence (rooted tree)** : graphe orienté obtenu à partir d'un arbre par le choix d'une racine et une orientation des arêtes à partir de cette racine.

**Arbre (tree)** : Graphe non orienté, connexe et sans cycle.

**Arbre couvrant ( spanning tree )** : graphe partiel d'un graphe qui soit un arbre. On rappelle qu'un graphe partiel contient les mêmes sommets mais moins d'arêtes.

**Arc (arc)** : Cas orienté. Dans l'arc $(x, y)$ $x$ est l'origine de l'arc et $y$ l'extrémité. $x$ est dit prédécesseur de $y$ et $y$ successeur de $x$. On dit également que $x$ est un père de $y$.

**Arête (edge)** : Cas non orienté. $x$ et $y$ sont les extrémités de l'arête $\{x, y\}$.

**Biparti ( bipartite )** : Un graphe $G$ est dit biparti s'il existe une partition de l'ensemble des sommets en $X = X_1 \cup X_2$ telle que toute arête $(x, y)$ de $G$ ait une extrémité dans $X_1$ et l'autre dans $X_2$.

**Boucle (loop)** : une boucle est un arc ou une arête d'un sommet vers lui-même. Un graphe simple n'a pas de boucle ni deux fois la même arête ou le même arc.

**Chaîne (chain)** : Cas non orienté : Une chaîne est une suite de sommets reliés par des arêtes. La longueur de la chaîne est le nombre d'arêtes. Une chaîne simple ne passe pas deux fois par la même arête, une chaîne élémentaire ne passe pas deux fois par le même sommet.

**Chemin (path)** : Cas orienté : Un chemin est une suite de sommets reliés par des arcs. La longueur d'un chemin est le nombre d'arcs. Un chemin est dit simple si il ne passe pas deux fois par le même arc. Un chemin est dit élémentaire si il ne passe pas deux fois par le même sommet.

**Circuit (circuit)** : Cas orienté : Un circuit est un chemin simple fermé.

**Clique (clique)** : Une clique est un sous-graphe complet.

**Complet (complete)** : Un graphe non orienté est complet si pour tout couple $(x, y)$ de sommets, il existe une arête entre $x$ et $y$.

**Composente connexe (connected component)** : Cas non orienté : Deux sommets $x$ et $y$ sont dans la même composante connexe s'il existe une chaîne reliant $x$ à $y$ dans $G$.

**Connexe (connected)** : Cas non orienté. Un graphe est connexe ssi pour tout couple $(x, y)$ de sommets, il existe une chaîne reliant $x$ à $y$ dans $G$. $g$ ne possède donc qu'une

seule composante connexe.

**Cycle (cycle)** : Cas non orienté. Un cycle est une chaîne simple fermée.

**Degré (degree)** : Cas non orienté : Le degré d'un sommet est son nombre de voisin.

**Degré entrant** : Cas orienté : Le degré entrant d'un sommet est son nombre de prédécesseurs.

**Degré sortant** : Cas orienté : Le degré sortant d'un sommet est son nombre de successeurs.

**Eulérien (eulerian)** : Cas non orienté. Un circuit est eulérien s'il passe une et une seule fois par chaque arête du graphe. Un graphe est eulérien s'il admet un circuit eulérien.

**Forêt (forest)** : Graphe non orienté sans cycle. Une forêt est donc une union d'arbres.

**Fortement connexe (strongly connected)** : Cas orienté. Un graphe est fortement connexe ssi pour tout couple $(x, y)$ de sommets, il existe un chemin reliant $x$ à $y$ dans $G$.

**Graphe non orienté (graph)**

**Graphe orienté (directed graph ou digraph)**

**Graphe partiel (spanning graph)** Graphe ayant les mêmes sommets que $G$ et une partie de ses arêtes. Utiles dans les raisonnements par récurrence sur le nombre d'arêtes.

**Hamiltonien (hamiltonian)** Cas non orienté : Une chaîne ou un cycle est hamiltonien ssi elle passe une et une seule fois par chaque sommet du graphe. Un graphe est hamiltonien s'il admet un cycle hamiltonien.

**Incident (incident)** Une arête est incidente à un sommet $x$ si $x$ est l'une de ses extrémités.

**Indice chromatique** : L'indice chromatique $\chi(G)$ est le plus petit nombre $k$ tel qu'il existe une $k$ coloration des arêtes de $G$. (A ne pas confondre avec le nombre chromatique $\gamma(G)$...)

**Liste d'adjacence (adjacency list)** : La liste d'adjacence d'un sommet $x$ est la liste de ses voisins (cas non orienté) ou la liste de ses successeurs (cas orienté). Dans le cas d'un multigraphe, les voisins sont répétés avec leur multiplicité.

**Liste d'arc ou d'arêtes (arc list, edge list)** : Représentation d'un graphe par une liste de couples $(x, y)$ représentant les arêtes ou les arcs du graphe.

**Matrice d'adjacence (adjacency matrix)** : Représentation matricielle d'un graphe. $M[x, y] = 1$ s'il existe un arc ou une arête de $x$ vers $y$ et 0 sinon. Dans le cas d'un multigraphe $M[x, y]$ est le nombre d'arcs ou d'arêtes de $x$ vers $y$.

**Nombre chromatique (chromatic number)** : Le nombre chromatique $\gamma(G)$ est le plus petit nombre $k$ tel qu'il existe une $k$ coloration des sommets de $G$. (A ne pas confondre avec l'indice chromatique $\chi(G)$...)

**Ordre (order)** : L'ordre d'un graphe est le nombre de ses sommets.

**Planaire (planar)** : Un graphe est planaire s'il en existe une représentation sagittale sans que des arêtes se croisent.

**Simple (simple)** : Un graphe est simple s'il n'a ni boucle ni arêtes (ou arcs) multiples.

**Sous-graphe (induced graph)** Graphe obtenu en enlevant à $G$ des sommets et toutes leurs arêtes (arcs) incidentes. Utiles dans les raisonnements par récurrence sur le nombre de sommets d'un graphe.

**Voisinage (neighborhood)** Cas non orienté. Ensemble des voisins d'un sommet.

**Voisinage entrant** Cas orienté. Ensemble des prédécesseurs d'un sommet.

**Voisinage sortant** Cas orienté. Ensemble des successeurs d'un sommet.

# Bibliographie

[1] Mathématiques discrètes et informatique. N.H. Xuong. Masson.
[2] Mathématiques pour l'informatique. Arnold et Guessarian. Masson.
[3] Algorithmes de graphes. Lacomme, Prins et Sevaux. Eyrolles.
[4] Types de données et algorithmes. Froidevaux, Gaudel et Soria. Edisciences.
[5] Graphes et algorithmes des graphes. V. Bouchitté. Cours de 3ème année de l'ENS Lyon.
[6] Eléments de mathématiques discrètes. L. Frécon. Presses polytechniques romandes.
[7] Méthodes mathématiques pour l'informatique. J. Vélu. Dunod.
[8] A logical approch to discrete math. D.Gries,F.B. Schneider. Springer.