Chapitre 10 : Théorie des graphes

Table des matières

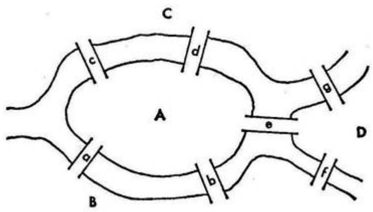
1 Introduction

1.1 Les origines

1.1.1 Le problème fondateur : les sept ponts de Königsberg

Énoncé : Peut-on effectuer une promenade à Königsberg passant exactement une fois par chacun des ponts ?

Variante : peut-on le faire en revenant à son point de départ ?

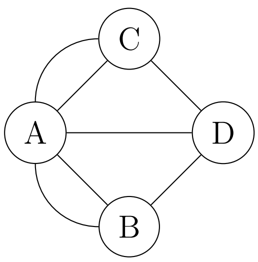


1.1.2 Une solution formelle

Présentée par Euler en 1735.

Trois étapes :

(1) Nommage des trois zones et représentation abstraite



(2) Formalisation de la notion de chemin

(3) Démonstration d'impossibilité : il n'existe pas de chemin / circuit eulérien dans ce graphe.

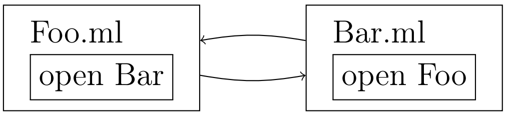
1.1.3 Remarque

Les points (1) et (2) sont les étapes fondatrices de la théorie des graphes vue comme une théorie mathématique. Cette théorie a pris une grande ampleur car elle permet de modéliser de nombreux phénomènes.

1.2 De nombreuses applications

1.2.1 Compilation

- Modélisation : on représente le graphe de dépendance entre fichiers.



• Problème : faisabilité : c'est un problème de détection de cycle.

Ordre de compilation : choisir un ordre, c'est effectuer un tri topologique du graphe.

1.2.2 Transports

• Modélisation : on représente un réseau de transports en commun en représentant les stations liées par les lignes qui y passent.

• Problème : recherche de chemin le plus court en terme de distance / de temps / nombre de stations.

1.2.3 Ordonnancement de tâches

• Problème : répartition d'un ensemble de tâches sur un nombre minimal d'unité de calcul.

• Modélisation : on utilise un graphe d'incompatibilité : on lie les tâches incompatibles entre elles. On veut attribuer une couleur (une unité de calcul) à chaque sommet de sorte qu'aucun sommet ne soit de la même couleur que l'un de ses voisins.

1.2.4 Construction d'un réseau électrique

• Problème : on veut raccorder un certain nombre de villes en utilisant le moins de cable possible.

• Modélisation : on utilise un graphe qui représente les villes liées par des axes annotés par leur longueur. On veut sélectionner des axes pour lier toutes les villes entre elles en utilisant le moins de longueur possible.

C'est la recherche d'un arbre couvrant de poids minimal.

2 Bases des graphes

2.1 Vocabulaire

2.1.1 Définition (*graphe*)

Un graphe est un couple $G = (S, A)$ où :

- S est un ensemble fini de *sommets* ou de *nœuds* ;
- A est un ensemble d'associations entre deux sommets, qui peut prendre plusieurs formes :
  – Si A est un ensemble de points de sommets, on dit que G est *non orienté* ;

– Si $a = \{s, s'\} \in A$, on dit que a est une *arête* d'extrémités s et s', que a est *incidente* à s et s', et que s et s' sont *adjacents* ou *voisins* ;

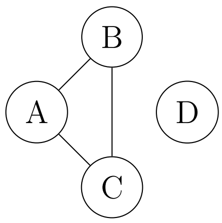
– Si A est un ensemble de couples de sommets, on dit que G est *orienté*.
Si $a = (s, s') \in A$, on dit que a est un *arc*, que s' est un *successeur* de s, que a est un *arc sortant* pour s et *entrant* pour s'.

2.1.2 Représentation graphique

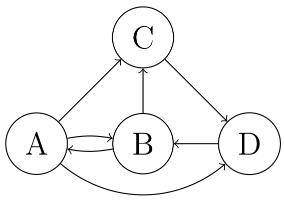
On place un point pour chaque sommet et on relie les extrémités d'une même arête (avec une flèche dans le cas orienté).
Exemple :

$$G = (\{A, B, C, D\}, \{\{A, B\}, \{B, C\}, \{C, A\}\})$$

est un graphe non orienté (GNO)



Autre exemple :



est la représentation du graphe orienté (GO) :

$$G = (\{A, B, C, D\}, \{(A, B), (B, A), (B, C), (D, B), (A, D), (A, C), (C, D)\})$$

2.1.3 Boucles

• Définition (*boucle*) : Une *boucle* dans un graphe est une arête / un arc dont les extrémités sont égales.

• Remarque : la définition 2.1.1 (page 4) empêche la présence de boucles dans les GNO. On peut les autoriser en considérant non pas des paires de sommets, mais des *multi-ensembles* de cardinal 2.

On pourrait aussi utiliser les *multi-ensembles* pour autoriser les multi-arêtes (plusieurs arêtes entre 2 sommets donnés, comme ent 1.1.1, page 3, mais c'est H.P : A sera toujours un ensemble).

2.1.4 Degré

• Définition (*degré*) : Soit $G = (S, A)$ un GNO et $s \in S$.
Le degré de s, noté $d(s)$ est le nombre de voisins de s :

$$d(s) = \Big| \{a \in A \mid s \in a\} \Big|$$

• Définition (*degré entrant / sortant*) : Soit $G = (S, A)$ un GO et $s \in S$.
Le *degré entrant* (resp. *sortant*) de s, noté $d_-(s)$ (resp. $d_+(s)$), est le nombre d'arcs entrants (resp. sortant) pour s :

$$d_-(s) = \Big| \{a \in A \mid \exists s' \in S \mid a = (s', s)\} \Big|$$

$$d_+(s) = \Big| \{a \in A \mid \exists s' \in S \mid a = (s, s')\} \Big|$$

• Proposition (formule de la somme des degrés) : Soit $G = (S, A)$ un graphe.
(1) Si G est un GNO sans boucle,

$$\sum_{s \in S} d(s) = 2\,|A|$$

(2) Si G est un GO,

$$\sum_{s \in S} d_-(s) = \sum_{s \in S} d_+(s) = |A|$$

□ Démonstration :
(1) On compte les extrémités d'arêtes :
– Une arête compte pour deux extrémités car ce n'est pas une boucle, donc il y en a $2\,|A|$;
– $\forall s \in S$, s est extrémité de $d(s)$ arêtes, donc il y en a

$$\sum_{s \in S} d(s)$$

(2) Par récurrence sur $|A|$:
– Si $|A| = 0,\ \forall s \in S,\ d_+(s) = d_-(s) = 0$ Ok.
– Héréditée : si $|A| > 0$, alors $\exists (s, s') \in A$.
On note $G' = (S,\ a \setminus \{(s, s')\})$.
Par hypothèse de récurrence :

$$|A| - 1 = |A \setminus \{(s, s')\}| = \sum_{v \in S \setminus \{s'\}} d_-(v) + \underbrace{d_-(s) - 1}_{\text{degré sortant de } s \text{ dans } G}$$

Donc $|A| = \sum_{v \in S} d_-(v)$
De même pour les degrés entrants, en considérant s' plutôt que s ■

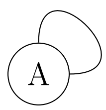
• Corollaire (*handshaking lemma*) : Tout GNO sans boucle possède un nombre pair de sommets de degré impair.

☐ Démonstration :

$$2\mathbb{N} \ni 2|A| = \sum_{s\in S} d(s) = \underbrace{\sum_{\substack{s\in S\\ d(s)\in 2\mathbb{N}}} d(s)}_{\in 2\mathbb{N}} + \underbrace{\sum_{\substack{s\in S\\ d(s)\in 2\mathbb{N}+1}} d(s)}_{\substack{\text{de la parité du nombre}\\ \text{de sommets de degré impair}}}$$

■

Contre-exemple en cas de boucle :



2.1.5 Graphes étiquetés

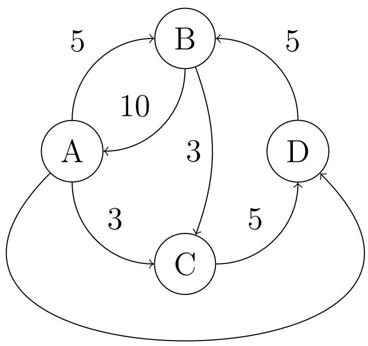
• Définition (*graphe étiqueté*) : Soit $G = (S, A)$ un graphe.

On dit que G est :

− *étiqueté* s'il est muni d'une fonction $f : A \longrightarrow V$ où V est un ensemble de valeurs appelées les *étiquettes*.

− *pondéré* s'il est étiquetté par des nombres (entiers / réels) : on parle de *poids* plutôt que d'étiquette.

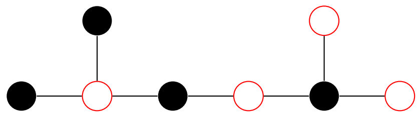
Exemple : 1.2.2 (page 4), 1.2.4 (page 4),



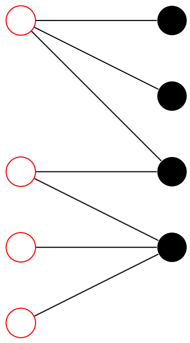
En MPI : les automates finis.

2.1.6 Graphes bipartis

• Définition (*graphes bipartis*) : Soit $G = (S, A)$ un graphe.
On dit que G est *biparti* s'il existe une partition de S (U, V) telle que pour toute arête a, une extrémité de a appartienne à U et l'autre à V.
Exemple (U, V) :



Peut aussi se représenter :



Remarque : un graphe biparti est 2-colorable.

2.2 Connexité

2.2.1 Définition (*chemin*)

Soit $G = (S, A)$ un graphe.
Un chemin dans G est une suite finie de sommets

$$(s_k)_{k \in [\![1\ ;\ n]\!]} \subset S \mid \forall i \in [\![0\ ;\ n-1]\!],\ \{s_i,\ s_{i+1}\} \in A$$

(resp. $(s_i,\ s_{i+1}) \in A$ dans le cas orienté).

La longueur du chemin est le nombre d'arêtes / d'arcs parcourus, ici n.

2.2.2 Définition (cas particuliers de chemins)

Soit $G = (S, A)$ un graphe et $p = s_0 \dots s_n$ un chemin dans G.
• p est *fermé* $\Leftrightarrow$ $s_0 = s_n$
• p est *élémentaire* si p passe au plus par chaque arête / arc, *i.e*

$$\forall i, j \in [\![0\ ;\ n-1]\!],\ i \neq j \Rightarrow \{s_i,\ s_{i+1}\} \neq \{s_j,\ s_{j+1}\}$$

(resp. $(s_i,\ s_{i+1}) \neq (s_j,\ s_{j+1})$)

- p est *simple* $\Leftrightarrow$

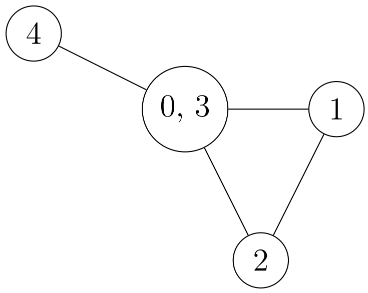
$$\forall i, j \in [\![0 \ ; \ n]\!] \,, \ i \neq j \Rightarrow s_i \neq s_j$$

sauf éventuellement $s_0 = s_n$ uniquement si $n \neq 2$ dans le cas non orienté.

Attention : vocabulaire différent selon les auteurs.

2.2.3 Vrai / Faux

(1) p élémentaire $\Rightarrow$ p simple ?
Faux :



(2) p simple $\Rightarrow$ p élémentaire ?
Vrai
(3) $\exists$ des chemins simples / fermés de longueur (non) nulle ?

2.2.4 Définition (*circuits / cycles / chemins eulériens*)

Soit $G = (S, A)$ un graphe, $p = s_0 \dots s_n$ un chemin de G.
- p est un *circuit* $\Leftrightarrow$ p est un chemin fermé de longueur non nulle ;
- p est un *cycle* $\Leftrightarrow$ p est un circuit élémentaire ;
- G est *acyclique* s'il ne contient aucun cycle ;
- p est un *chemin eulérien* $\Leftrightarrow$ p passe exactement une fois par chaque arête / arc, *i.e* si

$$\begin{cases} \{\{s_i, \ s_{i+1}\} \ | \ i \in [\![0 \ ; \ n-1]\!]\} = A \\ |A| = n \end{cases}$$

(resp. $(s_i, \ s_{i+1})$ pour les graphes orientés) ;
- G est eulérien $\Leftrightarrow$ G contient un chemin fermé eulérien.

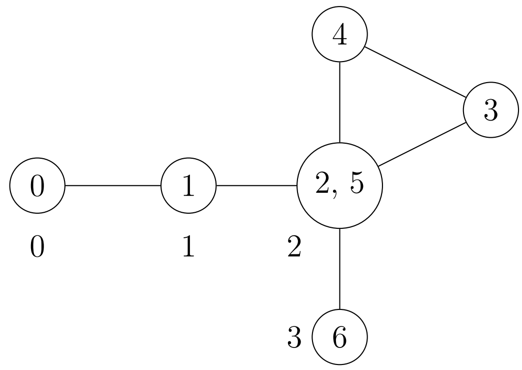
2.2.5 Remarques

- Un chemin eulérien est élémentaire ;

• On parle souvent de circuit / cycle eulérien dans la définition de graphe eulérien même si c'est un abus de langage pour le graphe sans arête ;

• une boucle est un cycle simple de longueur 1 ;

• On peut toujours rendre simple un chemin / cycle en coupant les circuits intermédiaires



Cela n'est pas toujours possible pour les circuits : le circuit s_0, s_1, s_2 dans un graphe non-orienté contenant une arête $\{s_0, s_1\}$ ne peut pas être rendu simple (couper le circuit le rend de longueur nulle : ce n'est plus un circuit).

2.2.6 Proposition

Soit $G = (S, A)$ un graphe.

Si G est biparti, alors G ne contient aucun cycle de longueur impaire.

☐ Démonstration

On note (U, V) une partition de S convenable.

Soit $c = s_0 \dots s_n$ un cycle dans G. On suppose sans perte de généralité que $s_0 \in U$.

On montre alors par récurrence finie que

$$\begin{cases} \forall i \in 2\mathbb{N} \cap [\![0\ ;\ n]\!]\,,\ s_i \in U \\ \forall j \in 2\mathbb{N}+1 \cap [\![0\ ;\ n]\!]\,,\ s_j \in V \end{cases}$$

Alors $s_n = s_0 \in U$, donc $n \in 2\mathbb{N}$ ■

2.2.7 Remarque

C'est une caractérisation des graphes bipartis (réciproque en **??**, page **??**)

2.2.8 Définition (*connexité*)

Soit $G = (S, A)$ un GNO et $s, s' \in S$.

• On dit que s et s' sont *connectés* dans G, noté $s \sim s'$, s'il existe un chemin reliant s et s' dans G ;

• G est *connexe* $\Leftrightarrow$ $\forall s, s' \in S,\ s \sim s'$.

2.2.9 Proposition

Soit $G = (S, A)$ un GNO. Alors $\sim_G$ est une relation d'équivalence.

Démonstration en $\boxed{\text{Exo}}$.

2.2.10 Définition (*composantes connexes*)

Soit $G = (S, A)$ un GNO et $s \in S$.

La *composante connexe* de G contenant s est la classe d'équivalence de s par $\sim_G$.

2.2.11 Lemme

Soit $G = (S, A)$ un GNO et $\{s_1, s_2\} \in A$.

On note $G' = (S, A \setminus \{\{s_1, s_2\}\})$, C la composante connexe de G contenant s_1 et s_2, et $\forall i \in [\![1\ ;\ 2]\!]$, C_i la composante connexe de G' contenant s_i.

Alors, si il existe un cycle dans G passant par $\{s_1, s_2\}$, alors $C_1 = C_2 = C$.

Sinon, $C_1 \cap C_2 = \varnothing$ et $C_1 \cup C_2 = C$.

☐ Démonstration

- $C_1 \cup C_2 = C$:

  $\boxed{\subseteq}$: vrai car un chemin dans G' est un chemin dans G.

  $\boxed{\supseteq}$: Soit $s \in C$.

  Il existe un chemin $n_0 \dots n_k$ de s à s_1 dans G (avec $n_0 = s$ et $n_k = s_1$)

  On considère $i = \min\{i \in [\![0\ ;\ k]\!] \mid n_i = s_1 \text{ ou } n_i = s_2\}$ (existe car $n_k = s_1$).

  Alors $n_0 \dots n_i$ est un chemin dans G' de s à s_1 ou s_2 donc $s \in C_1$ ou $s \in C_2$.

- Soit $C_1 = C_2$, soit $C_1 \cap C_2 = \varnothing$ car C_1 et C_2 sont des classes d'équivalences pour $\sim_G$.

- $C_1 = C_2 \iff \{s_1, s_2\}$ appartient à un cycle de G.

  $\boxed{\Rightarrow}$ $s_1 \in C_2$ donc il existe un chemin $s_2 s_3 \dots s_n s_1$ dans G' que l'on suppose élémentaire (on peut toujours rendre simple un chemin).

  Alors en ajoutant l'arête $\{s_1, s_2\}$ à ce chemin, on obtient $s_1 s_2 \dots s_n s_1$, qui est un chemin fermé, de longueur non nulle et élémentaire car le chemin initial ne contenait pas $\{s_1, s_2\}$ et était élémentaire. C'est donc un cycle contenant $\{s_1, s_2\}$.

  $\boxed{\Leftarrow}$ Quitte à réordonner les sommets, on peut toujours supposer qu'il existe un cycle $s_1 s_2 \cdots s_n s_1$.

  Comme il est élémentaire, le chemin $s_2 s_3 \cdots s_n s_1$ est un chemin de s_2 à s_1 dans G', donc $s_2 \sim_{G'} s_1$, donc $C_1 = C_2$ ■

2.2.12 Proposition

Soit $G = (S, A)$ un GNO avec $|S| = n$, et $|A| = m$.

(1) G a au moins $n - m$ composantes connexes ;

(2) G a exactement $n - m$ composantes connexes $\Leftrightarrow$ G est acyclique.

□ Démonstration

Par l'absurde, considérons un contre-exemple avec m minimal.

• Si $m = 0$: G n'a pas d'arête donc G est acyclique et a $n = n - 0 = n - m$ composantes connexes. Donc G n'est pas un contre-exemple : absurde.

• Donc $m > 0$: on peut essayer de supprimer une arête de G.

S'il existe un cycle dans G, on choisit une arête de ce cycle.

D'après 2.2.11 (page 11), on obtient G' avec les mêmes composantes connexes que G.

Par minimalité de m, G' n'est pas un contre-exemple, donc a au moins $n-(m-1)$ composantes connexes, donc G a au moins $n-m+1 > n-m$ composantes connexes.

Donc G n'est pas un contre-exemple : absurde.

• Donc G est acyclique.

Donc d'après 2.2.11 (page 11), supprimer une arête de G donne G' avec exactement une composante connexe de plus que G.

G' est acyclique et n'est pas un contre-exemple par minimalité de m, donc G' a exactement $n - (m - 1)$ composantes connexes.

Donc G a exactement $n - m + 1 - 1 = n - m$ composantes connexes.

Donc G n'est pas un contre-exemple : absurde. ■

2.2.13 Définition (*arbre*)

Un *arbre* est un graphe non orienté connexe et acyclique.

2.2.14 Proposition

Soit $G = (S, A)$ un GNO avec $|S| = n$ et $|A| = m$.

Les assertions suivantes sont équivalentes :

(1) G est un arbre ;

(2) G est connexe avec m minimal, *i.e* si on retire une arête de G, on pert la connexité ;

(3) G est acyclique avec m maximal, *i.e* si on ajoute une arête à G, on perd l'acyclicité ;

(4) G est connexe avec $m = n - 1$;

(5) G est acyclique avec $m = n - 1$.

□ Démonstration :

• (1) $\Rightarrow$ (4), (5) : G est connexe donc a exactement une composante connexe.

G est acyclique donc a exactement $n - m$ composantes connexes d'après 2.2.12 (page 12).

Donc $1 = n - m$, $i.e$ $m = n - 1$.

• (4) ou (5) $\Rightarrow$ (1) :

D'après 2.2.12 (page 12), G a au moins $n - m = 1$ composante connexe, exactement $\Leftrightarrow$ G est acyclique, donc G est connexe $\Leftrightarrow$ G est acyclique.

Donc (4) ou (5) $\Rightarrow$ (1).

• (1) $\Rightarrow$ (2) : c'est 2.2.11 (page 11) dans le cas acyclique.

• (2) $\Rightarrow$ (1) : si G n'était pas acyclique, on pourrait supprimer une arête d'un cycle, ce qui contredit la minimalité de m d'après 2.2.11 (page 11).

• (1) $\Rightarrow$ (3) : Si m n'était pas minimal, on pourrait ajouter une arête à G et obtenir G' acyclique et toujours connexe.

G' serait un arbre, donc par (2), retirer l'arête que l'on vient d'ajouter donnerait un graphe non connexe. Or c'est G qui est un arbre : absurde.

• (3) $\Rightarrow$ (1) : Si G n'est pas connexe, on peut ajouter une arête entre deux sommets de deux composantes connexes distinctes sans créer de cycle (d'après le lemme 2.2.11, page 11), donc m n'est pas maximal : absurde. ■

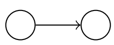
2.2.15 Définition (*forêt*)

Une *forêt* est un GNO acyclique.

2.2.16 Remarques

• Les composantes connexes d'une forêt sont des arbres ;

• La relation de connexité n'est pas une relation d'équivalence dans les graphes orientés : on perd la symétrie :



2.2.17 Définition (*connexité forte*)

Soit $G = (S, A)$ un GO, et $s, s' \in S$.

• On dit de s et s' sont *fortement connectés*, noté $s \sim_{\vec{G}} s$ si il existe un chemin de s à s' et un chemin de s' à s dans G.

• On dit que G est *fortement connexe* $\Leftrightarrow$ $\forall s, s'$ $s \sim_{\vec{G}} s'$.

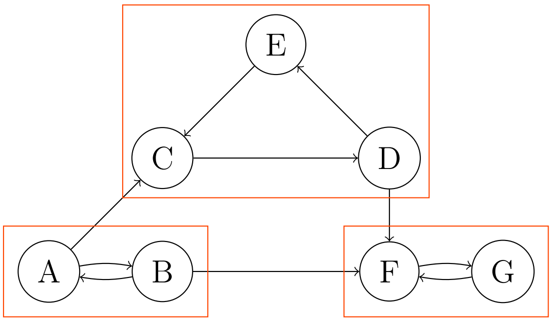
2.2.18 Proposition

Soit $G = (S, A)$ un GO.
Alors $\sim_{\vec{G}}$ est une relation d'équivalence.
Remarque : si G est un GNO, $\sim_G = \sim_{\vec{G}}$.

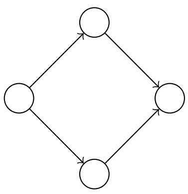
2.2.19 Définition (*composantes fortement connexes*)

Soit $G = (S, A)$ un GO, et $s \in S$.
La *composante fortement connexe* de G contenant s est la clause d'équivalence de s pour $\sim_{\vec{G}}$.

2.2.20 Exemple



Remarque : le graphe des composantes fortement connexes est un graphe orienté acyclique. Dans un tel graphe, on définit naturellement une relation d'ordre entre sommets : $s \leqslant s' \Leftrightarrow$ il existe un chemin de s à s' (on dit que s' est accessible à partir de s).
Dans le cas général, cet ordre n'est pas total :



On peut choisir un ordre total compatible avec cet ordre partiel en effectuant un tri topologique du graphe (*cf* TD_{30}).

3 Représentation des graphes

3.0 Remarque

Il s'agit d'étudier des implémentations effectives des graphes. On suppose dans la suite qu'une numérotation des sommets a été choisie, donc que $S = [\![0 ~;~ n-1]\!]$.

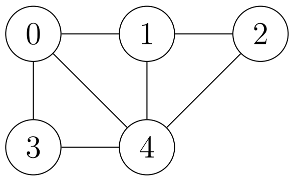
3.1 Matrice d'adjacence

3.1.1 Définition (*matrice d'adjacence*)

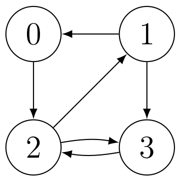
Soit $G = (\llbracket 0\ ;\ n-1 \rrbracket, A)$ un graphe.
La *matrice d'adjacence entière* (resp. *booléenne*) de G est la matrice $A_G = (a_{i,j})_{i,j \in \llbracket 0\ ;\ n-1 \rrbracket}$ définie par

$$\forall i, j \in \llbracket 0\ ;\ n-1 \rrbracket,\ a_{i,j} = \begin{cases} 1 \text{ (resp. V) si } \{i,j\} \in A \text{ (resp. } (i,j) \in A \text{ dans le cas orienté)} \\ 0 \text{ (resp. F) sinon} \end{cases}$$

3.1.2 Exemples



$$\longrightarrow \begin{pmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{pmatrix}$$



$$\longrightarrow \begin{pmatrix} F & F & V & F \\ V & F & F & V \\ F & V & F & V \\ F & F & V & F \end{pmatrix}$$

3.1.3 Proposition

Soit $G = (S, A)$ un GNO.
Alors A_G est symétrique.

☐ Démonstration :
$\forall i, j \in \llbracket 0\ ;\ n-1 \rrbracket$,

$$\begin{aligned} a_{i,j} = 1 \text{ (resp. V)} &\Leftrightarrow \{i,j\} \in A \\ &\Leftrightarrow \{j,i\} \in A \\ &\Leftrightarrow a_{j,i} = 1 \text{ (resp. V)} \ \blacksquare \end{aligned}$$

3.1.4 Proposition

Soit $G = (\llbracket 0\ ;\ n-1 \rrbracket, A)$ un graphe, et A_G sa matrice d'adjacence entière.
Alors, $\forall (i,j) \in \llbracket 0\ ;\ n-1 \rrbracket^2$, en notant $\forall k \in \mathbb{N}$, $a_{i,j}^{(k)}$ le coefficient (i,j) de $A_G^{(k)}$, $a_{i,j}^{(k)}$ est le nombre de chemins de longueur k de i à j.

☐ Démonstration

Par récurrence sur k

• $k = 0$: $A_G^k = I_n$

$\forall i, j \in [\![0 \ ; \ n-1]\!]$, il existe un chemin de longueur nulle de i à j $\Leftrightarrow$ $i = j$: ok.

• Hérédité : $A_G^{k+1} = A_G^k A_G$

Donc

$$\forall i, j \in [\![0 \ ; \ n-1]\!] \, , \ a_{i,j}^{(k+1)} = \sum_{l=0}^{n-1} a_{i,l}^{(k)} a_{l,j}^{(k)} = \sum_{\substack{l \in [\![0 \ ; \ n-1]\!] \\ \{l,i\} \in A}} a_{i,l}^{(k)}$$

Or tout chemin de longueur $k+1$ de i à j se décompose de manière unique en un chemin de i à un sommet l de longueur k suivit de l'arc / arête de l à j.

Donc l'hypothèse de récurrence conclut. ■

3.1.5 Définition (*matrice d'adjacence pondérée*)

Soit $G = (S, A, w)$ un graphe pondéré.

La *matrice d'adjacence pondérée* de G est la matrice $A_G = (a_{i,j})_{i,j \in [\![0 \ ; \ n-1]\!]}$ définie par

$$\forall i, j \in [\![0 \ ; \ n-1]\!] \, , \ \begin{cases} w(\{i,j\}) & \text{si } \{i,j\} \in A \\ +\infty & \text{sinon} \end{cases}$$

3.1.6 Implémentation

On utilise un tableau à deux dimensions.

• En OCaml :

```ocaml
type graphe = int array array
```

• En C : d'après le programme, on n'utilise que des tableaux de taille statiquement connue.

Exemple :

```c
typedef int graphe[10][20];
graphe g;
g[0][0] = 1;
```

Dans le cas général, on devrait utiliser des pointeurs (`typedef int** graphe;`), mais cela nécessiterait d'utiliser $n+1$ fois la fonction `malloc`.

On préférera linéariser le tableau :

```c
typedef int* graphe;
graphe g = (graphe) malloc(n*n*sizeof(int));
//la case i, j est g[i*n + j]
free(g);
```

Avantage : 1 `malloc`, 1 `free`.

Inconvénient : risque de se tromper dans les accès.

3.1.7 Complexité

- Complexité spatiale : $\mathcal{O}(n^2)$
- Complexité temporelle des opérations usuelles :
  - Création du graphe : $\mathcal{O}(n^2)$;
  - Test de l'existence d'une arête / d'un arc de i à j : $\mathcal{O}(1)$ (un accès dans la matrice) ;
  - Calcul du nombre d'arêtes / d'arcs : $\mathcal{O}(n^2)$;
  - Calcul de la liste des voisins / successeurs d'un sommet : $\mathcal{O}(n)$ (parcours de la ligne) ;
  - Ajout / suppression d'une arête / d'un arc : $\mathcal{O}(1)$ (Attention au cas non orienté) ;
  - Ajout / suppression de sommet : $\mathcal{O}(n^2)$ (reconstruire la matrice).

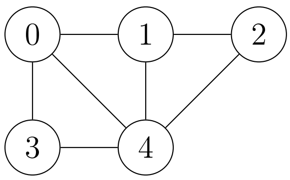
  $\boxed{\text{Exo}}$: code.

3.2 Listes d'adjacences

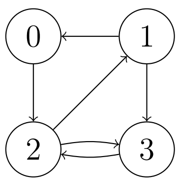
3.2.1 Définition (*listes d'adjacence*)

Soit $G = (\llbracket 0\ ;\ n-1 \rrbracket)$ un graphe.
On peut représenter G à l'aide d'un tableau de listes d'adjacence : $\forall i \in \llbracket 0\ ;\ n-1 \rrbracket$, la case d'indice i contient la liste des voisins / successeurs de i.

3.2.2 Exemple



0	$\square$	$\to$	1	$\to$	3	$\to$	4	$\to$			
1	$\square$	$\to$	0	$\to$	2	$\to$	4	$\to$			
2	$\square$	$\to$	1	$\to$	4	$\to$					
3	$\square$	$\to$	0	$\to$	4	$\to$					
4	$\square$	$\to$	0	$\to$	1	$\to$	2	$\to$	3	$\to$	



0	$\square$	$\to$	2	$\to$		
1	$\square$	$\to$	0	$\to$	3	$\to$
2	$\square$	$\to$	1	$\to$	3	$\to$
3	$\square$	$\to$	2	$\to$		

3.2.3 Cas des graphes pondérés

On peut utiliser des listes d'adjacence pondérées : chaque liste contient des couples (voisin, poids).

3.2.4 Implémentation

On utilise un tableau de listes chaînées.

- En OCaml :

```ocaml
type graphe = int list array
```

- En C :

```c
struct elem {
    int val;
    struct elem* next;
};
typedef struct elem* liste;

typedef liste* graphe;
```

Remarque : on peut se passer des listes en utilisant des tableaux : on peut par exemple utiliser une matrice dont les lignes ne sont pas toutes de même longueur en plaçant dans la première case de chaque ligne ne nombre de voisins (`g[i][0]` est le nombre de voisins de i et les voisins sont `g[i][1]`, ..., `g[i[g[i][0]]]`)

Problème : la linéarisation de cette matrice n'est pas pratique à manipuler.

Solution : on utilise un tableau `voisins` qui contient dans l'ordre les voisins des différents sommets et deux tableaux `debut` et `fin` tel que les voisins de i sont stockés entre les indices `debut[i]` (inclus) et `fin[i]` (exclu).

3.2.5 Complexité

On utilise un tableau de listes chaînées.

On note $n = |S|$, $m = |A|$.

- Complexité spatiale : $\mathcal{O}(n+m)$;
- Complexité temporelle des opérations usuelles :
  - Création de graphe (sans arêtes) : $\mathcal{O}(n)$;
  - Test d'existence de l'arête $\{i, j\}$ / de l'arc (i, j) : $\mathcal{O}\big(d_{(+)}(i)\big)$ (parcours de la liste d'adjacence de i) ;
  - Calcul du calcul d'arêtes / d'arcs : $\mathcal{O}(n+m)$ (calcul de la somme des longueurs des listes, divisée par 2 dans le cas non orienté) ;
  - Calcul de la liste des voisins d'un sommet : $\mathcal{O}(1)$ (accès à la case du sommet) ;
  - Ajout d'une arête / d'un arc : $\mathcal{O}(1)$ (ajout en tête de liste) ;
  - Suppression d'une arête / d'un arc entre i et j : $\mathcal{O}(d_+(i))$ dans le cas orienté, $\mathcal{O}(d(i)+d(j))$ dans le cas non orienté ;
  - Ajout d'un nœud : $\mathcal{O}(n)$ si tableau statique, $\mathcal{O}(1)$ amorti si tableau dynamique ;
  - Suppression d'un nœud : $\mathcal{O}(n+m)$ (création d'un nouveau tableau + renumérotation des nœuds).