# Chapitre 10 : Théorie des graphes

## Table des matières

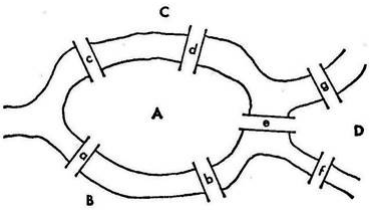
# 1 Introduction

## 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 ?

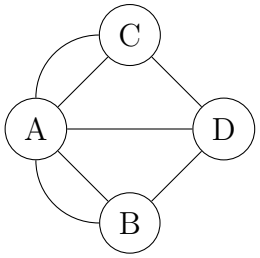



### 1.1.2 Une solution formelle

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.

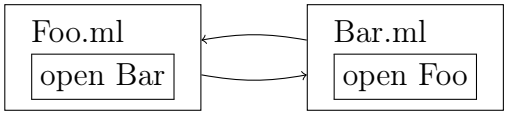
### 1.1.3 Remarque

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 câble 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 $s'$.

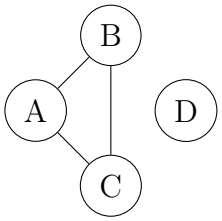
### 2.1.2 Représentation graphique

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

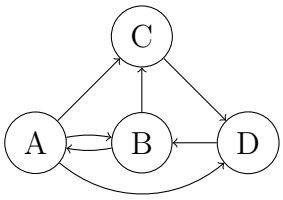
Exemple :

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

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 5) 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 en 1.1.1, page 4, 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 $s$ ■

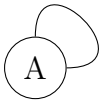
• Corollaire (*handshaking lemma*) : Tout GNO sans boucle possède un nombre pair de sommets de degré impair.

□ 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

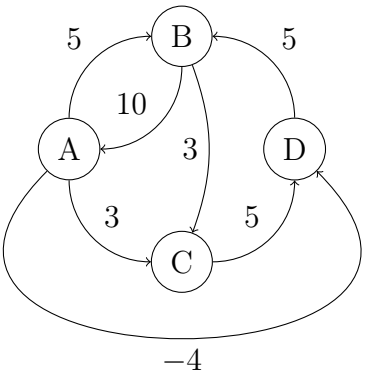
• Définition (*graphe étiqueté*) : Soit $G = (S, A)$ un graphe.

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 étiqueté par des nombres (entiers / réels) : on parle de *poids* plutôt que d'étiquette.

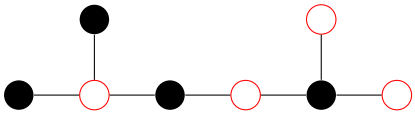
Exemple : 1.2.2 (page 5), 1.2.4 (page 5),



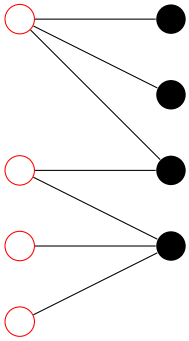

En MPI : les automates finis.

### 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 $(U, V)$ de $S$ 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 une fois 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* $\Leftrightarrow$

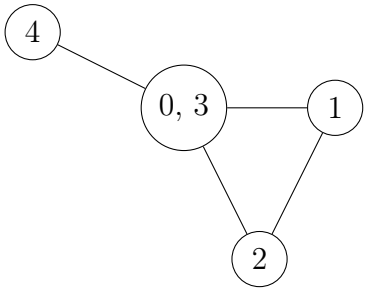
$$\forall i, j \in [\![0\ ;\ n]\!]\ ,\ i \neq j \Rightarrow s_i \neq s_j$$

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

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

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

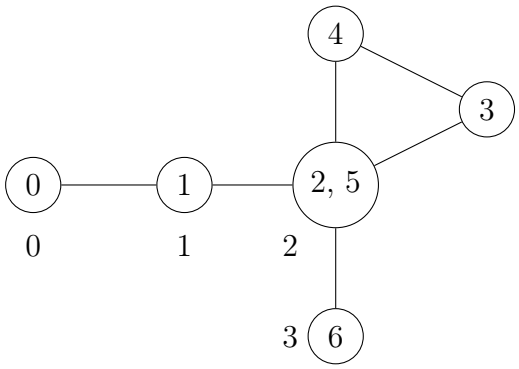
### 2.2.5 Remarques

• 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 4.1.8, page 22)

### 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 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 $\sim_G$.

• $C_1 = C_2 \Leftrightarrow \{s_1, s_2\}$ 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 12), 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 12), 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 perd 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) ⇒ (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 13).

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

• (4) ou (5) ⇒ (1) :

D'après 2.2.12 (page 13), $G$ a au moins $n-m = 1$ composante connexe, exactement ⇔ $G$ est acyclique, donc $G$ est connexe ⇔ $G$ est acyclique.

Donc (4) ou (5) ⇒ (1).

• (1) ⇒ (2) : c'est 2.2.11 (page 12) dans le cas acyclique.

• (2) ⇒ (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 12).

• (1) ⇒ (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) ⇒ (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 12), donc $m$ n'est pas maximal : absurde. ■

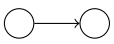
### 2.2.15 Définition (*forêt*)

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* ⇔ $\forall s, s' \in S,\ s \sim_{\vec{G}} s'$.

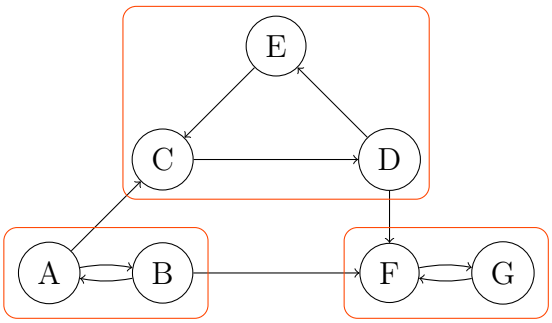
### 2.2.18 Proposition

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

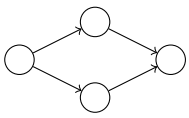
### 2.2.19 Définition (*composantes fortement connexes*)

Soit $G = (S, A)$ un GO, et $s \in S$.
La *composante fortement connexe* de $G$ contenant $s$ est la classe 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 $S = [\![0\ ;\ n-1]\!]$.

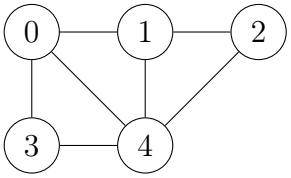
## 3.1 Matrice d'adjacence

### 3.1.1 Définition (*matrice d'adjacence*)

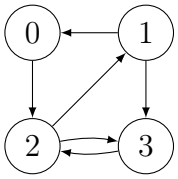
Soit $G = (\llbracket 0\ ;\ n-1 \rrbracket, A)$ un graphe.
La *matrice d'adjacence entière* (resp. *booléenne*) de $G$ est la matrice $A_G = (a_{i,j})_{i,j \in \llbracket 0\ ;\ n-1 \rrbracket}$ définie par

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

### 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

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

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 :

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

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

Exemple :

```
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 :

```
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).

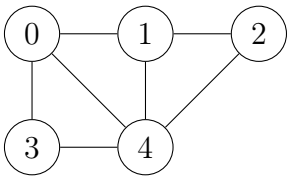
  Exo : code.

## 3.2 Listes d'adjacences

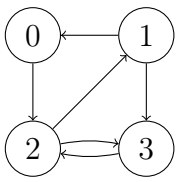
### 3.2.1 Définition (*listes d'adjacence*)

Soit $G = (\llbracket 0\ ;\ n-1 \rrbracket, A)$ 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 | □ | → | 1 | → | 3 | → | 4 | → | | |
| 1 | □ | → | 0 | → | 2 | → | 4 | → | | |
| 2 | □ | → | 1 | → | 4 | → | | | | |
| 3 | □ | → | 0 | → | 4 | → | | | | |
| 4 | □ | → | 0 | → | 1 | → | 2 | → | 3 | → |




| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | □ | → | 2 | → | | |
| 1 | □ | → | 0 | → | 3 | → |
| 2 | □ | → | 1 | → | 3 | → |
| 3 | □ | → | 2 | → | | |

### 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 :

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

- En 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 nombre 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).

# 4 Parcours de graphes

## 4.1 Généralités

### 4.1.1 Définition (parcours d'un GNO connexe)

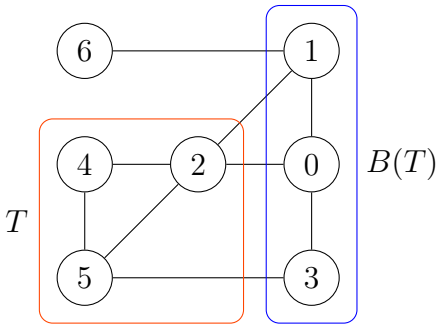
Soit $G = (S, A)$ un GNO connexe.
Un parcours de $G$ partant d'un sommet $s \in S$ est une suite de sommets $s_0 \dots s_{n-1}$ tq
(1) $s_0 = s$
(2) $n = |S|$ et $\{s_i \mid i \in [\![0\ ;\ n-1]\!]\} = S$
(3) $\forall i \in [\![1\ ;\ n-1]\!]\ ,\ \exists j < i \mid \{s_i, s_j\} \in A$

### 4.1.2 Algorithme générique

• Définition (*bordure*) : Soit $G = (S, A)$ un GNO, et $T \subseteq S$.
On appelle *bordure* de $T$ l'ensemble

$$B(T) = \{s \in S \setminus T \mid \exists t \in T \mid \{s, t\} \in A\}$$

Exemple :




Algorithme : entrée : $G = (S, A)$ un GNO connexe, $s \in S$

> $s_0 \leftarrow s$
> Pour $i$ de 1 à $|S| - 1$ :
> $\quad s_i \leftarrow$ un élément de $B(\{s_0 \dots s_{i-1}\})$
> Renvoyer $s_0 \dots s_{|S|-1}$

### 4.1.3 Proposition

Soit $G = (S, A)$ un GNO connexe.
Les parcours de $G$ sont exactement les suites produites par l'algorithme générique.

□ Démonstration :
$\boxed{\subseteq}$ Soit $s_0 \dots s_{n-1}$ un parcours de $G$.
Soit $i \in [\![1\ ;\ n-1]\!]$.

$s_i \in B(\{s_0 \dots s_{i-1}\})$ car $s_i \notin \{s_0 \dots s_{i-1}\}$ (sinon le point (2) de 4.1.1 (page 20) est faux), et d'après 4.1.1 (3), $\exists j < i \mid \{s_i, s_j\} \in A$.
Donc l'algorithme produit $s_0 \dots s_{n-1}$ en choisissant $s_i$ à l'itération $i$, $\forall i \in [\![1 \ ; \ |S| - 1]\!]$.

$\boxed{\supseteq}$ Soit $s_0 \dots s_{n-1}$ une séquence générique produite par l'algorithme générique.
$\forall i \in [\![1 \ ; \ n-1]\!], \exists j < i \mid \{s_i, s_j\} \in A$ car $s_i \in B(\{s_0 \dots s_{n-1}\})$
Les $(s_i)_{i \in [\![0 \ ; \ n-1]\!]}$ sont 2 à 2 distinctes par définition de la bordure.
Il reste à montrer que $n = |S|$, *i.e* que $\forall i \in [\![1 \ ; \ n-1]\!], \ B(\{s_0 \dots s_{i-1}\}) \neq \varnothing$.
Or si $B(\{s_0 \dots s_{i-1}\}) = \varnothing$ avec $i < |A|$, alors il existe un sommet qui n'est pas connecté à $s_0 \dots s_{i-1}$ donc le graphe n'est pas connexe. ■

### 4.1.4 Définition (*sous-graphe induit / couvrant*)

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

• Le *sous-graphe induit* par $T \subseteq S$ est le graphe $(T, A_T)$, où

$$A_T = \{a \in A \mid \text{les deux extrémités de } a \text{ sont dans } T\}$$

• Le *sous-graphe induit* par $A' \subseteq A$ est le graphe $(S', A')$ où

$$S' = \{s \in S \mid s \text{ est une extrémité d'un élément de } A'\}$$

• Un *sous-graphe* $(S', A')$ de $G$ est dit *couvrant* $\Leftrightarrow$ $S' = S$.

### 4.1.5 Proposition

Soit $G = (S, A)$ un GNO connexe, et $s_0 \dots s_{n-1}$ un parcours de $G$.
$\forall i \in [\![1 \ ; \ n-1]\!]$, on choisit $t_i \in \{s_0, \dots, s_{i-1}\} \mid \{s_i, t_i\} \in A$ (possible d'après 4.1.1 (3) (page 20)).

Alors le sous-graphe induit par $\{\{s_i, t_i\} \mid i \in [\![1 \ ; \ n-1]\!]\}$ est un arbre couvrant de $G$.

☐ Démonstration :

$\forall k \in [\![1 \ ; \ n-1]\!]$, on note $G_k$ le sous-graphe induit par $\{\{s_i, t_i\} \mid i \in [\![1 \ ; \ k]\!]\}$.
Il suffit de montrer que $G_{n-1}$ est un arbre à $n$ sommets.
En effet, $n = |S|$ d'après 4.1.1 (2) (page 20), donc $G_{n-1}$ serait couvrant.
On montre par récurrence finie que $\forall k \in [\![1 \ ; \ n-1]\!], \ G_k$ est un arbre à $k+1$ sommets.

• Initialisation : $k = 1$. $G_1$ est le sous-graphe induit par $\{\{s_1, t_1\}\}$.
$t_1 = s_0 \neq s_1$ donc $G_1$ est le graphe


C'est bien un arbre à $2 = 1 + 1$ sommets.

• Héréditée : Soit $k \in [\![1\ ;\ n-2]\!]$ tel que $G_k$ est un arbre à $k+1$ sommets.

Montrons que $G_{k+1}$ est un arbre à $k+2$ sommets.

$G_{k+1}$ est construit à partir de $G_k$ en ajoutant l'arête $\{s_{k+1}, t_{k+1}\}$.

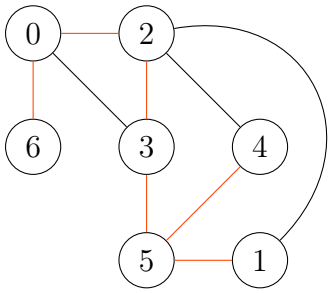
$t_{k+1}$ est un sommet de $G_k$ est pas $s_{k+1}$, donc $G_{k+1}$ a exactement 1 sommet et une arête de plus que $G_k$.

Or $G_k$ est connexe avec $k+1$ sommets et $k$ arêtes (*cf* 2.2.14, page 13).

Donc $G_{k+1}$ est connexe (le seul sommet ajouté au graphe connexe $G_k$ est relié à un sommet de $G_k$ dans $G_{k+1}$) avec $k+2$ sommets et $k+1$ arêtes.

Donc $G_{k+1}$ est un arbre (*cf* 2.2.14, page 13) à $k+2$ sommets. ■

### 4.1.6 Exemple




| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Parcours : | 0, | 2, | 3, | 5, | 6, | 1, | 4. |
| Antécédent : | -, | 0, | 2, | 3, | 0, | 5, | 5. |

Arbre couvrant.

### 4.1.7 Remarques

• Un problème classique est le calcul d'un arbre couvrant de poids minimal dans un graphe pondéré (où le poids d'un arbre est la somme des poids des arêtes), *cf* MPI.

• En pratique, on implémente souvent les parcours en déterminant un voisin non visité d'un sommet "courant". Le sommet courant est donc naturellement choisi comme antécédent de ce voisin, ce qui produit un arbre particulier associé au parcours.

### 4.1.8 Proposition (réciproque de 2.2.6)

Soit $G = (S, A)$ un GNO

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

□ Démonstration :

On suppose sans perte de généralité que $G$ est connexe (sinon on travaille séparément sur ses composantes connexes).

Alors $G$ possède un arbre couvrant $T$ dont on distingue un sommet $r$.

Comme $T$ est un arbre, il existe un unique chemin simple dans $T$ entre toute paire de sommets.


Exo

On peut alors partitionner $S$ en $(U, V)$, où :

– $U$ est l'ensemble des sommets $s$ tels que la longueur du chemin de $r$ à $s$ dans $T$ est paire ;

– $V$ est l'ensemble des sommets $s$ tels que la longueur du chemin de $r$ à $s$ dans $T$ est impaire.

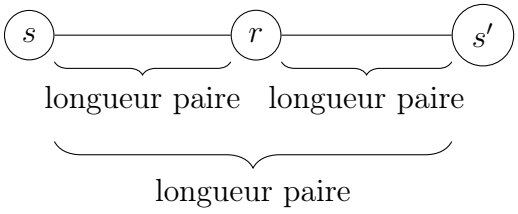
$U \cap V = \varnothing$, et comme $T$ est couvrant, $U \cup V = S$.

Il reste à montrer que toute arête a une extrémité dans $U$ et l'autre dans $V$.

Supposons qu'il existe une arête $\{s, s'\} \in A$ avec $s \in U$ et $s' \in U$ (le cas $V$ est similaire).

Il existe un unique chemin simple de $s$ à $s'$ dans $T$.

– Si il passe par $r$ :




L'ajout de $\{s, s'\}$ à ce chemin donne un cycle de longueur impaire.

– Sinon, on procède de même avec le premier sommet $r'$ commun aux chemins de $s$ à $r$ et de $s'$ à $r$.

La somme des longueurs des chemins de $s$ à $r'$ et de $s'$ à $r'$ est bien paire car elles sont de même parité (par disjonction de cas selon que $r' \in U$ ou $r' \in V$).

Ajouter l'arête $\{s, s'\}$ donne un cycle de longueur impaire : absurde. ■

### 4.1.9 Généralisation aux graphes non connexes

• Définition (parcours d'un GNO) : Soit $G = (S, A)$ un GNO.

Un parcours de $G$ à partir de $s \in S$ est une suite finie $s_0 \dots s_{n-1} \in S$ tq :

(1) $s_0 = s$

(2) $|S| = n$ et $\{s_i \mid i \in [\![0\ ;\ n-1]\!]\} = S$

(3) $\forall i \in [\![1\ ;\ n-1]\!],\ B(\{s_0 \dots s_{i-1}\}) \neq \varnothing \Rightarrow s_i \in B(\{s_0 \dots s_{i-1}\})$.

• Conséquences :

– Un parcours est la caractérisation de parcours des composantes connexes ;

– On peut tirer d'un tel parcours une forêt couvrante du graphe.

### 4.1.10 Généralisation aux GO

On adopte simplement la notion de bordure : si $G = (S, A)$ est un GO, et $T \subseteq S$,

$$B(T) = \{s \in S \setminus T \mid \exists t \in T \mid (t, s) \in A\}$$

On peut également sélectionner un antécédent pour chaque sommet du parcours.

Si tous les sommets sont accessibles à partir du sommet de départ (*a priori* si $G$ est fortement connexe), alors on obtient une arborescence couvrante de $G$.

• Définition (*arborescence*) : Soit $G = (S, A)$ un GO.

$G$ est une *arborescence* s'il peut être obtenu à partir d'un arbre $(S, A')$ en distinguant un sommet $r \in S$, et en orientant chaque $\{s, s'\} \in A$ en $(s, s')$ si l'unique chemin simple de $r$ à $s'$ passe par $s$, en $(s', s)$ sinon.

On parle également *d'arbre enraciné*.

On parle aussi de forêt pour un ensemble d'arborescences disjointes et un parcours d'un GO permet de définir une forêt couvrante en sélectionnant des arcs comme dans le cas non orienté.

On dit que ce sont des *arcs de liaison* pour les distinguer des autres catégories d'arcs.

### 4.1.11 Catégorisation des arcs

Soit $G = (S, A)$ un GO, $s_0 \dots s_{n-1}$ un parcours de $G$.

On note $I = \{i \in [\![1\ ;\ n-1]\!] \mid B(\{s_0 \dots s_{i-1}\}) \neq \varnothing\}$.

$\forall i \in I$, on choisit $t_i \in \{s_0 \dots s_{i-1}\} \mid (t_i, s_i) \in A$ (on choisit les arcs de liaison).

Soit $(u, v) \in A$.

• $(u, v)$ est un *arc de liaison* si $\exists i \in I \left| \begin{array}{l} u = t_i \\ v = s_i \end{array} \right.$

• $(u, v)$ est un *arc arrière* si $v$ est un ancêtre de $u$ dans une arborescence de la forêt définie par les arcs de liaison, *i.e*

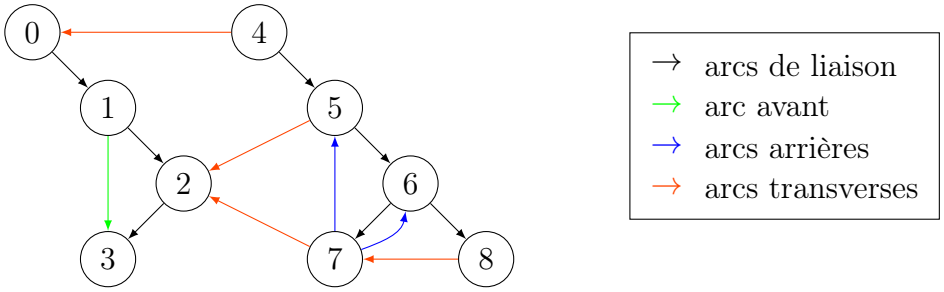
$$\exists i < j \left| \begin{array}{l} s_i = v \\ s_j = u \end{array} \right. , \ \exists k_1 < k_2 < \dots < k_l,\ l \geqslant 2, \left| \begin{array}{l} k_1 = i \\ k_l = j \\ \forall r \in [\![2\ ;\ l]\!],\ s_{k_{r-1}} = t_{k_r} \end{array} \right.$$

• $u, v$ est un *arc avant* si $v$ est un descendant non direct de $u$ dans une arborescence de la forêt définie par les arcs de liaison, *i.e*

$$\exists i < j \left| \begin{array}{l} v = s_j \\ u = s_i \end{array} \right. , \ \exists k_1 < k_2 < \dots < k_l,\ l > 2, \left| \begin{array}{l} k_1 = i \\ k_l = j \\ \forall r \in [\![2\ ;\ l]\!],\ s_{k_{r-1}} = t_{k_r} \end{array} \right.$$

• $(u, v)$ est un *arc transverse* s'il n'entre pas dans les catégories précédentes.

Exemple :




## 4.2 Parcours en profondeur

### 4.2.1 Retour sur l'algorithme générique

Cet algorithme nécessite de déterminer la bordure d'un ensemble de sommets construit *incrémentalement*. Cela implique que l'on peut déterminer cette bordure sans la recalculer de zéro : il suffit d'y ajouter les voisins non encore visités du dernier sommet extrait, à chaque itération.

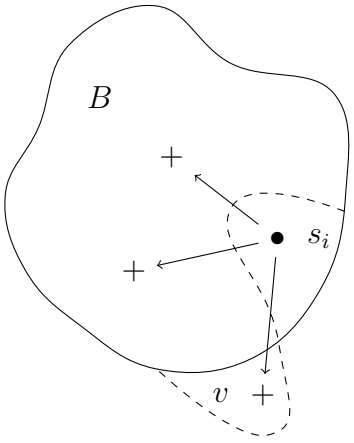
Algorithme : entrée : $G = (S, A)$ un graphe, et $s \in S$.

$B \leftarrow \{s\}$
$i \leftarrow 0$
$V \leftarrow \{s\}$

Tant que $B \neq \varnothing$
    $s_i \leftarrow$ un élément que l'on extrait de $B$
    $i \leftarrow i + 1$

    Pour chaque voisin $v$ de $s_i$
        Si $v \notin V$
            $V \leftarrow V \cup \{v\}$
            $B \leftarrow B \cup \{v\}$

Renvoyer $s_0 \dots s_{i-1}$



### 4.2.2 Remarque

Cet algorithme facilite la construction d'une arborescence pour le parcours obtenu : par exemple, le père de chaque sommet $v$ est le sommet $s_i$ qui a provoqué l'insertion de $v$ dans $B$.

L'ordre d'extraction des sommets est donc déterminant pour la structure du parcours : le choix des structures de données est important.

### 4.2.3 Parcours en profondeur

On obtient un parcours en profondeur si l'on extrait toujours le dernier sommet inséré : $B$ se comporte comme une pile. L'ensemble $V$ peut être représenté par un tableau de booléens (sa fonction indicatrice). On utilise pour $G$ les listes d'adjacence car la seule opération sur $G$ est le parcours de l'ensemble des voisins d'un sommet.

Implémentation en OCaml, en ajoutant un fonction de traitement des sommets :

```
let dfs (traitement : int -> unit) (g : graphe) (s : int) : unit =
    let b = Stack.create () in
    let v = Array.make (Array.length g) false in
    Stack.push s b;
    v.(s) <- true;

    while not (Stack.is_empty b) do
        let s = Stack.pop b in
        traitement s;
        List.iter (
            fun s' ->
                if not v.(s') then begin
                    v.(s') <- true;
                    Stack.push s' b
                end
        )
        g.(s)
    done
```

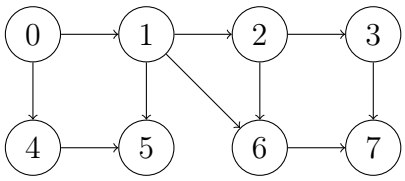
Complexité :

• Spatiale : $\mathcal{O}(|S|)$ ($v$ de taille $|S|$ et $b$ de taille au plus $|S|$ car chaque sommet accessible depuis $s$ y est inséré exactement une fois).

• Temporelle : si `traitement` est de complexité $\mathcal{O}(1)$, alors `dfs` est en $\mathcal{O}(|S| + |A|)$

$$\left( \mathcal{O}(|S|) \text{ pour créer } v, \ + \sum_{\substack{s' \text{ accessible} \\ \text{depuis } s}} d_{(+)}(s') \right)$$

### 4.2.4 Exemple




0, 1, 2, 3, 7, 5, 6, 4

### 4.2.5 Remarque

On verra dans le $TD_{30}$ (du cours) une implémentation récursive du parcours où l'on ne manipule pas explicitement $B$ car c'est la pile des appels récursifs.

### 4.2.6 Application : test d'accessibilité

```
let est_accessible_depuis (g : graphe) (s  : int) (s' : int) :
    bool =
    let res = ref false in
    dfs (fun v -> res := !res || v = s') g s;
    !res
```

Autre version, qui est plus efficace dans le meilleurs cas :

```
exception Trouve
let est_accessible_depuis (g : graphe) (s : int) (s' : int) : bool
    =
    try
        dfs (fun v -> if v = s' then raise Trouve) g s;
        false
    with
    | Trouve -> true
```

### 4.2.7 Application : test de connexité

```
let est_connexe (g : graphe) : bool =
    let n = ref 0 in (*n est le nombre de sommets rencontres*)
    dfs (fun _ -> incr n) g 0;
    !n = Array.length g
```

### 4.2.8 Application : calcul des composantes connexes

Idée : on attribue un numéro à chaque composante et on marque de ce numéro les sommets de la composante lors d'un parcours. On lance des parcours tant qu'il reste des sommets non numérotés.

```
let composantes_connexes (g : graphe) : int array =
    let n = Array.length g in
    let composante = Array.make n (-1) in
    let c = ref 0 in

    for i = 0 to n - 1 do
        if composante.(i) = -1 then begin
            dfs (fun v -> composante.(v) <- !c) g i;
            incr c;
        end
    done;
    composante

```

Complexité : $\mathcal{O}\left(|S|^2\right)$ dans le pire cas (pas d'arêtes) à cause de la création de $v$ dans chaque appel à `dfs`

Exo : on peut obtenir une complexité $\mathcal{O}(|S| + |A|)$ en ne créant pas un tableau à chaque nouveau parcours. (Remarque : `not v.(s)` $\leftrightarrow$ `composante.(s) = -1`)

## 4.3 Parcours en largeur

### 4.3.1 Idée

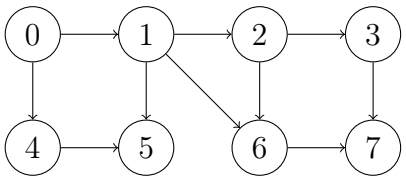
On obtient un parcours en largeur en visitant tous les voisins d'un sommet avant de poursuivre plus en profondeur : il faut extraire de $B$ les sommets par ordre d'insertion donc $B$ se comporte comme une file.

Implémentation : comme `bfs` avec `Queue` à la place de `Stack`.

Remarques :

- Même complexité : $\mathcal{O}(|S| + |A|)$
- On parcourt les sommets par ordre de distance au sommet de départ.

### 4.3.2 Exemple

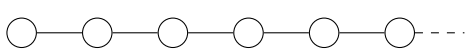



0, 1, 4, 2, 5, 6, 3, 7

Les sommets sont parcourus par ordre de distance au premier sommet.

### 4.3.3 Application : calcul de plus court chemin dans un graphe non pondéré

Idée : comme les sommets sont parcourus par ordre de distance au sommet de départ, une arborescence bien choisie d'un parcours en largeur donne des plus courts chemins du sommet de départ vers les sommets accessibles depuis ce sommet.



On représente l'arborescence à l'aide d'un tableau de prédécesseurs : le prédécesseur d'un sommet donné est le père de ce sommet dans l'arborescence choisie.

```
let plus_courts_chemins (g : graphe) (s : int) : int array =
    let pred = Array.make (Array.length g) (-1) in
    pred.(s) <- s;
    bfs (
        fun s -> List.iter (
            fun s' -> if pred.(s') = -1 then pred.(s') <- s
        ) g.(s)
    ) g s
```

Complexité : $\mathcal{O}(|S| + |A|)$ (au lieu de faire un parcours des voisins de chaque sommet, on en fait 2).

### 4.3.4 Proposition (correction de `plus_courts_chemins`)

On suppose sans perte de généralité que $G$ est connexe.

$\forall s' \in S$, on note :

- $d_{s'}$ la distance de $s$ à $s'$ (*i.e* la longueur d'un plus court chemin de $s$ à $s'$) ;
- $l_{s'}$ la longueur du chemin de $s$ à $s'$ obtenu à l'aide du tableau `pred` ;
- $n_{s'}$ le rang d'insertion de $s'$ dans la file (convention : $n_s = 0$).

Alors

$$\forall s' \in S, \quad \begin{array}{l}(1)\ d_{s'} = l_{s'} \\ (2)\ \forall s'' \in S,\ d_{s''} < d_{s'} \Rightarrow n_{s''} < n_{s'}\end{array}$$

□ Démonstration :

Par récurrence forte sur le rang d'insertion :

• $n_{s'} = 0$ : alors $s' = s$, $l_s = 0 = d_s$ et (2) est trivial (il n'existe pas de sommet $s''$ avec $d_{s''} < 0$).

• Héréditée : on suppose la propriété vraie jusqu'au rang $n$ et on note $s'$ le sommet de rang $n_{s'} = n + 1$.

(1) On sait que $d_{s'} \leqslant l_{s'}$ par définition. Supposons $d_{s'} < l_{s'}$ $(*)$.

On considère un plus court chemin de $s$ à $s'$, noté $s \rightsquigarrow s'' \to s'$, et on note $p = \texttt{pred}.(s')$

On sait que :

- $d_{s'} = 1 + d_{s''}$ $(**)$ (sinon le chemin choisi ne serait pas un plus court chemin)
- $l_{s'} = 1 + l_p$ $(***)$
- $n_p < n_{s'}$ ($s'$ est inséré lorsque $p$ est extrait).

Par H.R, on sait alors que $d_p = l_p$, d'où $d_{s''} < d_p$ (avec $(*), (**), (***)$).

Donc par H.R, $n_{s''} < n_p$ donc, lorsque $s''$ est extrait, $s'$ n'a pas encore de prédécesseur et $s''$ devrait être le prédécesseur de $s'$ : absurde.

Donc $d_{s'} = l_{s'}$.

Supposons $\exists s'' \in S \mid d_{s''} < d_{s'}$ et $n_{s''} > n_{s'}$ (cas d'égalité impossible)

$(\neg\forall x,\ A(x) \Rightarrow B(x) \equiv \exists x \mid A(x) \wedge \neg B(x))$ On considère un plus court chemin $s \rightsquigarrow q \to s''$ et on note $p = \texttt{pred}.(s')$.

$n_p < n_{s'}$ donc on dispose de l'hypothèse de récurrence pour $p$.

Donc $d_p = l_p = l_{s'} - 1 \overset{(1)}{=} d_{s'} - 1 > d_{s''} - 1 = d_q$

Donc par H.R, $n_q < n_p$.

Alors, comme le prédécesseur de $s''$ est défini au plus tard au moment de l'extraction de $q$, $s''$ est inséré dans la file avant tous les successeurs de $p$, dont $s'$ : absurde. ■

# 5 Plus court chemins dans un graphe pondéré

## 5.1 Contexte

### 5.1.1 Introduction

Certaines applications nécessitent de travailler dans des graphes pondérés (ex : 1.2.2, 1.2.4, page 5). Dans ce contexte, le poids d'un arc représente le coût d'un certain choix et on ne cherche plus à minimiser le nombre d'arcs mais plutôt le coût global des choix.

### 5.1.2 Définition (*poids d'un chemin*)

Soit $G = (S, A, w)$ un graphe pondéré, et $p = s_0 \dots s_n$ un chemin dans $G$.

Le *poids de* $p$ est

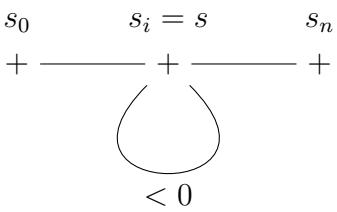
$$w(p) = \sum_{i=0}^{n-1} w(\{s_i, s_{i+1}\})$$

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

### 5.1.3 Remarque

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

Si il existe un cycle $c$ dans $G$ de poids $w(c) < 0$, on ne peut pas vraiment parler de plus courts chemins dans $G$ : $\forall s \in c,\ \forall p = s_0 \dots s_n \mid \exists i \in [\![0\ ;\ n]\!] \mid s_i = s$, on peut construire un chemin $p'$ de poids $w(p') < w(p)$, de $s_0$ à $s_n$ en insérant dans $p$, à la position $i$, le cycle $c$ réordonné pour avoir un cycle de $s$ à lui-même.



### 5.1.4 Définition (*plus court chemin / distance*)

Soit $G = (S, A, w)$ un graphe pondéré sans cycle de poids $< 0$.

Un chemin $p = s_0 \dots s_n$ est un *plus court chemin* de $s_0$ à $s_n$ si $\forall p'$ chemin de $s_0$ à $s_n$, $w(p) \leqslant w(p')$.

$\forall s, s' \in S$ la distance de $s$ à $s'$ est $d(s, s') = \min\{w(p) \mid p \text{ chemin de } s \text{ à } s'\}$ (ou $+\infty$ si il n'y a pas de chemin de $s$ à $s'$).

□ Démonstration de l'existence du min :

On note $d = \min\{w(p) \mid p \text{ chemin simple de } s \text{ à } s'\}$ (existe car cet ensemble est fini).

Montrons que $d = d(s, s')$, *i.e* $\forall p$ chemin de $s$ à $s'$, $w(p) \geqslant d$.

Si $p$ est simple, c'est vrai par définition.

Si $p$ n'est pas simple, alors en notant $p = s_0 \dots s_n$, $\exists i, j \in [\![0\ ;\ n]\!] \mid i < j$ et $s_i = s_j$.

Donc on peut couper le circuit de $s_i$ à $s_j$, de poids $\geqslant 0$, et obtenir $p' = s_0 \dots s_i s_{j+1} \dots s_n$ tel que $w(p') \leqslant w(p)$ et $|p'| < |p|$

En procédant par induction bien fondée avec les chemins simples comme cas de base, on montre qu'il existe un chemin simple $p''$ de $s_0$ à $s_n$ tel que $\underbrace{w(p'')}_{\geqslant d} \leqslant w(p)$

■

### 5.1.5 Exercice

Montrer que $d$ satisfait l'inégalité triangulaire.

### 5.1.6 Problèmes

On s'intéresse aux problèmes suivants :

(1) Déterminer des plus courts chemins entre tous les couples de sommets ;

(2) Déterminer des plus courts chemins d'un sommet donné vers tous les sommets.

Remarque : résoudre l'un de ces problèmes permet de résoudre l'autre, mais il existe des algorithmes spécialisés pour chacun des deux.

## 5.2 Algorithme de Floyd-Warshall

### 5.2.1 Principe

On cherche à résoudre le problème 5.1.6 (1) (page 31) dans un graphe pondéré $G = (S, A, w)$ sans cycle de poids strictement négatif, en cherchant des chemins simples (possible grâce au point 5.1.4, page 31).

L'idée de l'algorithme de Floyd-Warshall est la suivante : on procède par programmation dynamique en cherchant $\forall k \in [\![0\ ;\ |S|]\!]$, $\forall s, s' \in S$, un plus court chemin de $s$

à $s'$ n'utilisant que des sommets intermédiaires d'indice strictement inférieur à $k$ (étant donné une numérotation des sommets).

### 5.2.2 Relation de récurrence

• Cas de base ($k = 0$) : comme il n'y a pas de sommet d'indice strictement négatif, il ne peut pas y avoir de sommet intermédiaire, donc les plus courts chemins sont les arcs / arêtes.

En notant $\forall s, s' \in S,\ d^{(k)}(s, s')$ la longueur d'un plus court chemin de $s$ à $s'$ n'utilisant que des sommets intermédiaires d'indice $< k$, on sait que $d^{(0)}(s, s') = w(\{s, s'\})$ avec la convention $w(\{s, s'\}) = +\infty$ si $\{s, s'\} \notin A$.

• Héréditée : Soient $s, s' \in S$.

Un plus court chemin de $s$ à $s'$ n'utilisant que des sommets intermédiaires d'indice $< k + 1$ peut ne pas passer par le sommet d'indice $k$ et dans ce cas réalise $d^{(k)}(s, s')$.

Si un tel chemin passe par $k$, on peut supposer qu'il y passe une seule fois puisqu'on cherche des chemins simples.

Les portions de chemins de $s$ à $k$ et de $k$ à $s'$ sont alors des plus court chemins (sinon le chemin initial ne serait pas un plus court chemin : propriété de sous-structure optimale) n'utilisant que des sommets intermédiaires d'indice $< k$.

Ainsi,

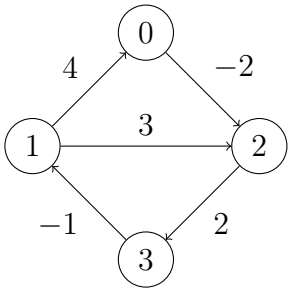
$$d^{(k+1)}(s, s') = \min\big(d^{(k)}(s, s'), d^{(k)}(s, k) + d^{(k)}(k, s')\big)$$

Résolution du problème : c'est $d^{(|S|)}$ qui nous intéresse car tous les sommets intermédiaires sont autorisés.

Exo relation de récurrence sur $p^{(k)}(s, s')$, le prédécesseur de $s'$ sur un plus court chemin de $s$ à $s'$ n'utilisant que des sommets intermédiaires d'indice $< k$.

$$\begin{cases} p^{(0)}(s, s') = s \\ p^{(k+1)}(s, s') = \begin{cases} p^{(k)}(s, s') \text{ si } d^{(k+1)}(s, s') = d^{(k)}(s, s') \\ p^{(k)}(k, s') \text{ sinon} \end{cases} \end{cases}$$

### 5.2.3 Exemple



$$d^{(0)} : \begin{pmatrix} +\infty & +\infty & -2 & +\infty \\ 4 & +\infty & 3 & +\infty \\ +\infty & +\infty & +\infty & 2 \\ +\infty & -1 & +\infty & +\infty \end{pmatrix} \quad d^{(1)} : \begin{pmatrix} +\infty & +\infty & -2 & +\infty \\ 4 & +\infty & \boxed{2} & +\infty \\ +\infty & +\infty & +\infty & 2 \\ +\infty & -1 & +\infty & +\infty \end{pmatrix}$$

$$d^{(2)} : \begin{pmatrix} +\infty & +\infty & -2 & +\infty \\ 4 & +\infty & 2 & +\infty \\ +\infty & +\infty & +\infty & 2 \\ \boxed{-3} & -1 & \boxed{1} & +\infty \end{pmatrix} \quad d^{(3)} : \begin{pmatrix} +\infty & +\infty & -2 & \boxed{0} \\ 4 & +\infty & 2 & \boxed{4} \\ +\infty & +\infty & +\infty & 2 \\ -3 & -1 & 1 & \boxed{3} \end{pmatrix}$$

$$d^{(4)} : \begin{pmatrix} \boxed{3} & \boxed{-1} & -2 & 0 \\ 4 & \boxed{3} & 2 & 4 \\ \boxed{5} & \boxed{1} & \boxed{3} & 2 \\ -3 & -1 & 1 & 3 \end{pmatrix}$$

Remarque : dans la matrice d'adjacence pondérée, on peut choisir de placer des 0 sur la diagonale puisque le plus court chemin d'un sommet à lui-même est le chemin vide.

### 5.2.4 Implémentation en C

• Structures de données : on choisit une matrice pour $d^{(k)}$, $p^{(k)}$ et aussi pour $G$ (matrice d'adjacence pondérée, car c'est directement $d^{(0)}$). On pourrait utiliser des listes d'adjacence car la complexité du calcul de $d^{(0)}$ à partir de ces listes est dominé par la complexité du reste de l'algorithme.

On supposera que la matrice est linéarisée :

```c
typedef double* graphe;

floyd_warshall(graphe g, int n) {
    double* d = (double*) malloc(n*n*sizeof(double));
    int* p = (int*) malloc(n*n*sizeof(int));

    for (int i = 0 ; i < n ; i++)
        for (int j = 0 ; j < n ; j++) {
            d[i*n + j] = g[i*n + j];
            p[i*n + j] = i;
        }

    for (int k = 0 ; k < n ; k++)
        for (int i = 0 ; i < n ; i++)
            for (int j = 0 ; j < n ; j++)
                if (d[i*n + k] + d[k*n + j] < d[i*n + j]) {
                    d[i*n + j] = d[i*n + k] + d[k*n + j];
                    p[i*n + j] = p[k*n + j];
                }

    free(d);
    return p;
}
```

Complexité :

- Temporelle : $\mathcal{O}(|S|^3)$
- Spatiale : $\mathcal{O}(|S|^2)$

## 5.3 Algorithme de Dijkstra

### 5.3.1 Principe

On cherche à résoudre le problème 5.1.6 (2) (page 31) dans un graphe pondéré $G = (S, A, w)$, où $w : A \longrightarrow \boxed{\mathbb{R}^+}$. On cherche donc, étant donné $s \in S$, à calculer $d(s, s')$, $\forall s' \in S$.

L'algorithme de Dijkstra est un algorithme glouton qui se présente comme une variante du parcours en largeur : on parcourt les sommets par ordre de distance à $s$ croissante, mais ici la distance est définie par la pondération du graphe. La file du parcours en largeur est remplacée par une file de priorité dont les priorités sont des estimations des distances (puisque ces dernières ne sont pas connues).

### 5.3.2 Algorithme glouton

$\forall s' \in S$, on note $w(s')$ l'estimation de $d(s, s')$.

Initialement, $\begin{cases} w(s) = 0 \\ w(s') = +\infty \ \forall s' \neq s \end{cases}$

On construit incrémentalement un ensemble $E \subseteq S \mid \forall s' \in E,\ w(s') = d(s, s')$.

Initialement, $E = \varnothing$, puis à chaque itération, on insère dans $E$ un sommet $s' \in S \setminus E \mid w(s')$ est minimal (choix glouton). Lors de l'insertion de $s'$ dans $E$, on observe les voisins de $s'$ pour essayer d'améliorer l'estimation pour ces sommets : $\forall s'' \in S \setminus E$ voisin de $s'$, on remplace $w(s'')$ par

$$\min(w(s''), d(s, s') + w(\{s', s''\}))$$

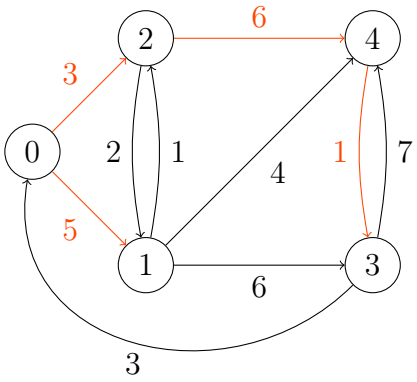
(resp. $(s', s'')$ dans le cas orienté).

En parallèle, on met à jour un tableau de prédécesseurs pour représenter les plus courts chemins.

Remarque : $S \setminus E$ se comporte comme une file de priorité min où $w(s')$ est le priorité de l'élément $s'$.

Quitte à compléter $G$ avec des arêtes / arcs de poids $+\infty$, c'est aussi la bordure utilisée dans l'algorithme générique de parcours.

### 5.3.3 Exemple




| $s'$ | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| $w(s')$ | 0 | 5 | 3 | 10 | 9 |
| $p(s')$ | 0 | 0 | 0 | 4 | 2 |

E : 0, 2, 1, 4, 3

### 5.3.4 Correction de l'algorithme de Dijkstra

On démontre l'invariant suivant :

$$\begin{cases} \forall s' \in E,\ w(s') = d(s, s') \\ \forall s'' \in S \setminus (E \cup \{s\}),\ w(s'') = \min\limits_{s' \in E}(d(s, s') + w(\{s', s''\})) \end{cases}$$

• Initialement : $E = \varnothing,\ \forall s'' \in S \setminus \{s\},\ w(s'') = +\infty = \min \varnothing$ (par convention) : l'invariant est vrai.

• Invariant : on traite la première itération à part.

− à la première itération, $s' = s$.

$w(s) = 0 = d(s, s)$ (car les poids sont positifs)

$\forall s'' \neq s,\ w(s'')$ ne change pas si $s$ et $s''$ ne sont pas adjacents : ok pour le calcul du minimum, et $w(s'')$ devient $w(\{s, s''\})$ si $s$ et $s''$ sont adjacents : encore ok.

− Aux itérations suivantes : on suppose

$$\begin{cases} \forall s' \in E,\ w(s') = d(s, s') \\ \forall s'' \in S \setminus (E \cup \{s\}),\ w(s'') = \min\limits_{s' \in E}(d(s, s') + w(\{s', s''\})) \end{cases}$$

On choisit $s' \in S \setminus E \mid w(s')$ est minimal pour l'insérer dans $E$.

Montrons que $w(s') = d(s, s')$.

On considère un chemin de poids minimal de $s$ à $s'$.

Il s'écrit $p = s \rightsquigarrow u \to v \rightsquigarrow s'$, où $v$ est le premier sommet du chemin tel que $v \notin E$

$$\begin{aligned} d(s,s') = w(p) &= \underbrace{w(s \rightsquigarrow u)}_{=d(s,u)} + w(\{u,v\}) + \underbrace{w(v \rightsquigarrow s')}_{\geqslant 0} \\ &\geqslant \min_{u \in E}(d(s,u) + w(\{u,v\})) \\ &= w(v) \text{ car } v \notin E \text{ (et } v \neq s \text{ car } s \in E) \\ &\geqslant w(s') \text{ par définition de } s' \end{aligned}$$

Donc $d(s,s') \geqslant w(s')$

Or $w(s')$ est le poids d'un chemin de $s$ à $s'$ : $w(s') = \min_{u \in E}(d(s,u) + w(\{u,s'\}))$

Donc en choisissant $u \in E$ qui réalise ce minimum et en concaténant un plus court chemin de $s$ à $u$ et l'arête $\{u,s'\}$, on obtient un chemin de $s$ à $s'$ de poids $w(s')$.

Donc $w(s') \geqslant d(s,s')$ d'où $d(s,s') = w(s')$.

$\forall s'' \in S \setminus (E \cup \{s,s'\})$, si $s'$ et $s''$ ne sont pas adjacents, alors $w(s'')$ est inchangé : ok pour le calcul du minimum.

Si $s'$ et $s''$ sont adjacents, $w(s'')$ devient

$$\begin{aligned} & \min(w(s''), d(s,s') + w(\{s', s''''\})) \\ = \ & \min\left(\min_{u \in E}(d(s,u) + w(\{u,s''\})), d(s,s') + w(\{s',s''\})\right) \\ = \ & \min_{u \in E \cup \{s'\}}(d(s, u + w(\{u,s''\}))) \end{aligned}$$

• Correction : à la fin de l'algorithme, $E = S$, donc l'invariant donne $\forall s' \in S,\ w(s') = d(s,s')$, donc si les mises à jour de prédécesseurs sont cohérents avec le calcul des $w(s')$, alors on obtient des plus court chemins de $s$ vers tous les sommets.

### 5.3.5 Implémentation

• Structures de données : on utilise une file de priorité min pour $S \setminus E$ qui doit permettre la mise à jour des priorités. On utilise des tableaux pour $w$ et les prédécesseurs. On représente $G$ par des listes d'adjacence pondérées car à chaque itération on a besoin de parcourir les voisins d'un sommet donné.

```
struct arc {
    double poids;
    int cible;
};

struct elem {
    struct arc val;
    struct elem* next;
};

typedef struct elem* liste;
typedef liste* graphe;
```

On suppose implémenté un type `fp` (file de priorité) proposant les primitives suivantes :

- `fp create(double* w, int n)` : crée une file contenant $[\![0\ ;\ n-1]\!]$ avec les priorités `w[i]` ;

– `bool is_empty(fp f)` : test de vacuité ;

– `int take_min(fp, f)` : extraction de l'élément de priorité min ;

– `void update(fp f, int i, double w)` : définit $w$ comme la nouvelle priorité de $i \in f$.

Code :

```c
int* dijkstra(graphe g, int n, int s) {
    double* w = (double*) malloc(n * sizeof(double));
    for (int i = 0 ; i < n ; i++) {
        if (i == s)
            w[i] = 0;
        else
            w[i] = INFINITY; //in math.h
    }

    fp f = create(w, n);

    int* p = (int*) malloc(n * sizeof(int));
    for (int i = 0 ; i < n ; i++)
        p[i] = -1;

    while (!is_empty(f)) {
        int u = take_min(f);
        liste l = g[u];

        while (l != NULL) {
            struct arc a = l->val;
            if(w[u] + a.poids < w[a.cible]) {
                w[a.cible] = w[u] + a.poids;
                p[a.cible] = u;
                update(f, a.cible, w[a.cible]);
            }

            l = l->next;
        }
    }

    free(w);
    //free memory for f if necessary here
    return p;
}
```

Complexité :

- Spatiale : $\mathcal{O}(|S|)$
- Temporelle : cela dépend de l'implémentation du type `fp` :
  – Implémentation naïve : tableau `w` des priorités + tableau de booléens + nombre d'éléments (pour le test de vacuité en temps constant).
    * Initialisation : $\mathcal{O}(|S|)$ ;
    * Test de vacuité : $\mathcal{O}(1)$ (répété $n+1$ fois) ;
    * Extraction du min : $\mathcal{O}(|S|)$ (répété $|S|$ fois) ;
    * Mise à jour de priorité : $\mathcal{O}(1)$ (répété $d_{(+)}(u)$ fois $\forall u \in S$).

* Au total : $\mathcal{O}(|S|) + \mathcal{O}(|S|) + \mathcal{O}\big(|S|^2\big) + \underbrace{\mathcal{O}\left(\sum_{u\in S} d_{(+)}(u)\right)}_{=\mathcal{O}(|A|)=\mathcal{O}\left(|S|^2\right)} = \mathcal{O}\big(|S|^2\big)$

– Implémentation à l'aide d'un tas :
  * Initialisation : $\mathcal{O}(|S|)$ (*via* une construction par forêt) ;
  * Test de vacuité : $\mathcal{O}(1)$ répété $|S|+1$ fois ;
  * Extraction du min : $\mathcal{O}(\log|S|)$ répété $|S|$ fois ;
  * Mise à jour de priorité : $\mathcal{O}(\log|S|)$ (percolation) répété $d_{(+)}(u)$fois $\forall u \in S$.
  * Au total :

$$\mathcal{O}(|S|) + \mathcal{O}(|S|) + \mathcal{O}(|S|\log|S|) + \underbrace{\mathcal{O}\left((\log|S|)\sum_{u\in S} d_+(u)\right)}_{=\mathcal{O}(|A|\log|S|)} = \mathcal{O}\Big((|S|+|A|)\log|S|\Big)$$

Si le graphe est *dense*, *i.e*, $|A| = \Theta\big(|S|^2\big)$, on obtient $\big(|S|^2\log|S|\big)$ : c'est pire.
Cela peut être meilleurs s'il y a peu d'arêtes, par exemple si $|A| = \mathcal{O}(|S|)$, ce qui donne $\mathcal{O}(|S|\log|S|)$
(H.P) : Avec une implémentation à l'aide de *tas de Fibonacci*, on obtient $\mathcal{O}(|S|\log|S| + |A|)$.