CH X : Graphes

I. Notion de graphe non orienté

I.1. Définition

Définition

Soit S un ensemble fini.

Soit $\mathcal{E}_S$ l'ensemble des parties à deux éléments de S. Autrement dit :

$$\mathcal{E}_S \ = \ \{ \ \{s,t\} \mid (s,t) \in S \times S \ \text{ et } \ s \neq t \ \}$$

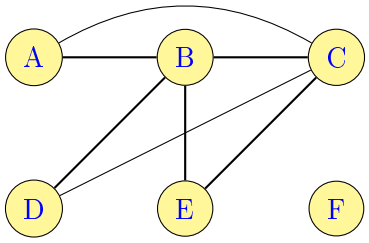
- Un graphe $\mathcal{G}$ est la donnée d'un couple $\mathcal{G} = (S, \mathcal{A})$ où :
  - S est appelé ensemble des **sommets** du graphe $\mathcal{G}$,
  - $\mathcal{A} \subset \mathcal{E}_S$ est appelé ensemble des **arêtes** du graphe $\mathcal{G}$.

Vocabulaire

- On appelle **ordre** du graphe $\mathcal{G}$ le nombre de sommets de $\mathcal{G}$.
  Autrement dit, l'ordre de $\mathcal{G}$ est le cardinal de l'ensemble S.
- Si $\{A, B\} \in \mathcal{A}$ on dit que l'arête $\{A, B\}$ relie les sommets A et B ou encore que l'arête $\{A, B\}$ est **incidente** aux sommets A et B. L'arête $\{A, B\} \in \mathcal{A}$ peut être notée $A - B$.
- On appelle **degré** d'un sommet s et on note $d(s)$ le nombre d'arêtes incidentes à ce sommet.
- Enfin, on dit que deux sommets sont **adjacents** s'ils sont reliés par une arête.

Exemple

On considère le graphe suivant, donné par sa représentation graphique.



Graphe 1

- Le graphe $\mathcal{G} = (S, \mathcal{A})$ est ici défini par :
  - l'ensemble des sommets $S = \{A, B, C, D, E, F\}$,
  - l'ensemble des arêtes $\mathcal{A} = \big\{ \{A, B\}, \{A, C\}, \{B, C\}, \{B, D\}, \{B, E\}, \{C, D\}, \{C, E\} \big\}$.
- Le graphe $\mathcal{G}$ est d'ordre $\text{Card}(S) = 6$.
- Par ailleurs :
  - le sommet A est de degré 1.
  - le sommet B est de degré 4.
  - le sommet F est de degré 0.

Remarque
Un graphe est une construction visant à expliciter / formaliser / modéliser des relations (les arêtes) entre des données (des sommets). S'interroger sur des relations entre données est une démarche extrêmement fréquente que l'on rencontre dans des domaines variés :

- une carte routière décrit les axes routiers (arêtes) reliant des villes (sommets).
- un arbre généalogique décrit la relation de descendance ou d'ascendance (arêtes) entre différents membres d'une même famille.
- en biologie, l'interactome humain est un graphe dont les sommets sont les protéines et dont les arêtes décrivent les interactions entre les protéines.
- un réseau social peut être décrit par les liens d'amitié virtuelle (arêtes) reliant des utilisateurs (sommets).
- sur un sujet donné (climato-scepticisme), on peut construire le graphe dont les sommets sont des usagers de $\mathbb{X}$ (anciennement Twitter). Les arêtes relient deux usagers dès que l'un a relayé l'autre. On s'intéresse alors à la dynamique des interactions. Pour la percevoir visuellement, les longueurs des arêtes seront fonction des interactions : plus un groupe d'usagers relaie des informations des autres usagers du groupe, plus leurs sommets se rapprochent. Cela permet d'observer des communautés, leurs tailles, l'arrivée de nouveaux membres ainsi que les membres les plus actifs. On peut remonter ainsi aux sources principales de la diffusion d'informations climato-sceptiques.

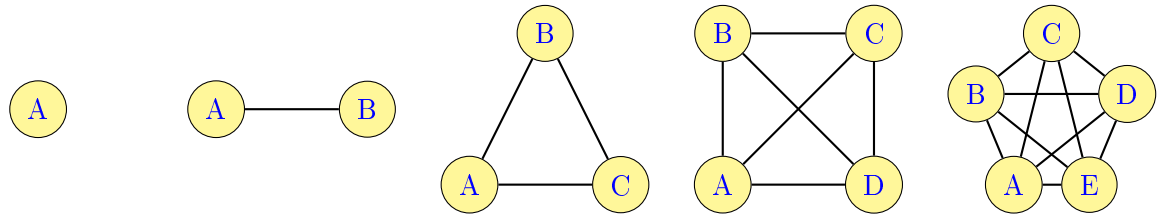
I.2. Notion de graphe complet

Définition

Soit $\mathcal{G} = (S, \mathcal{A})$ un graphe.

- On dit que le graphe $\mathcal{G}$ est **complet** lorsque tous ses sommets sont deux à deux adjacents. Autrement dit, $\mathcal{G}$ est **complet** lorsque tout couple de sommets disjoints est relié par une arête.

Exemple
On peut donner une représentation graphique des graphes complets à n sommets pour différentes valeurs de n.



$n = 1$ $n = 2$ $n = 3$ $n = 4$ $n = 5$

Exercice
Donner une représentation graphique du graphe complet à 6 sommets puis une représentation graphique du graphe complet à 7 sommets.

Théorème 1.

Soit $\mathcal{G} = (S, \mathcal{A})$ un graphe.

On note $n = \text{Card}(S)$ l'ordre de $\mathcal{G}$.

Si $\mathcal{G}$ est un graphe complet, alors $\mathcal{G}$ possède exactement $\dfrac{n(n-1)}{2}$ arêtes.

Démonstration.
On peut faire deux démonstrations différentes.

1) En revenant à la définition

Si le graphe $\mathcal{G}$ est complet, alors $\mathscr{A} = \mathcal{E}_S$ (tout couple de sommets définit une arête). On en déduit donc directement :

$$\text{Card}(\mathscr{A}) \;=\; \text{Card}(\mathcal{E}_S) \;=\; \binom{n}{2} \;=\; \frac{n\,(n-1)}{2}$$

2) Par construction

Démontrons par récurrence : $\forall n \in \mathbb{N}^*$, $\mathscr{P}(n)$,

où $\mathscr{P}(n)$: tout graphe complet à n sommets possède exactement $\dfrac{n\,(n-1)}{2}$ arêtes.

- **Initialisation** :
  - D'une part, un graphe complet à 1 sommet ne possède pas d'arête.
  - D'autre part : $\dfrac{1\,(1-1)}{2} = 0$.

  D'où $\mathscr{P}(1)$.
- **Hérédité** : soit $n \in \mathbb{N}^*$.

  Supposons $\mathscr{P}(n)$ et démontrons $\mathscr{P}(n+1)$ (*i.e.* : tout graphe complet à $n+1$ sommets possède exactement $\dfrac{(n+1)\,n}{2}$ arêtes).
  - Notons $\mathcal{G} = (S, \mathcal{A})$ un graphe complet à $n+1$ sommets.

    Soit $A \in S$. Considérons alors le graphe $\mathcal{G}' = (S', \mathcal{A}')$ défini par :
    - $S' = S \setminus \{A\}$.
    - $\mathcal{A}' = \mathcal{A} \setminus \big\{\ \{A, B\} \mid B \in S \ \text{ et } \ B \neq A\ \big\}$. Autrement dit, $\mathcal{A}'$ est l'ensemble obtenu à partir de $\mathcal{A}$ en supprimant toutes les arêtes incidentes à A.

    Alors $\mathcal{G}'$ est un graphe complet à n sommets.

    On en déduit, par hypothèse de récurrence : $\text{Card}(\mathcal{A}') = \dfrac{n\,(n-1)}{2}$.

    On a alors :

$$\begin{aligned}
\text{Card}(\mathcal{A}) &= \text{Card}\Big(\mathcal{A}' \cup \big\{\ \{A, B\} \mid B \in S \ \text{ et } \ B \neq A\ \big\}\Big) \\
&= \text{Card}\left(\mathcal{A}'\right) + \text{Card}\Big(\big\{\ \{A, B\} \mid B \in S \ \text{ et } \ B \neq A\ \big\}\Big) && \textit{(car ces deux ensembles sont disjoints)} \\
&= \frac{n\,(n-1)}{2} + n && \textit{(car dans un graphe complet, tout sommet est relié à tous les autres sommets)} \\
&= \frac{n}{2}\,\big((n-1)+2\big) \;=\; \frac{n\,(n+1)}{2}
\end{aligned}$$

    D'où $\mathscr{P}(n+1)$.

  Par principe de récurrence : $\forall n \in \mathbb{N}^*$, $\mathscr{P}(n)$. □

I.3. Matrice d'adjacence d'un graphe

Définition

Soit $\mathcal{G} = (S, \mathcal{A})$ un graphe.

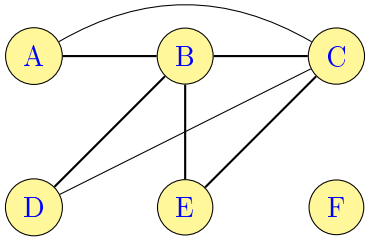
On note $n = \text{Card}(S)$ l'ordre de $\mathcal{G}$. Il existe donc une bijection φ entre $[\![1, n]\!]$ et S.

- On appelle **matrice d'adjacence** du graphe $\mathcal{G}$ la matrice carrée $M \in \mathscr{M}_n(\mathbb{R})$ définie par :

$$M_{i,j} = \begin{cases} 1 & \text{si les sommets } \varphi(i) \text{ et } \varphi(j) \text{ sont adjacents} \\ 0 & \text{si les sommets } \varphi(i) \text{ et } \varphi(j) \text{ ne sont pas adjacents} \end{cases}$$

Exemple

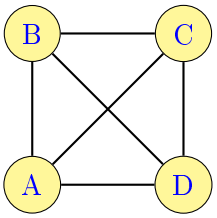
- La matrice d'adjacence du Graphe 1 du premier exemple est donnée par :



Graphe 1

$$\begin{array}{c} \\ \mathrm{A} \\ \mathrm{B} \\ \mathrm{C} \\ \mathrm{D} \\ \mathrm{E} \\ \mathrm{F} \end{array} \begin{array}{c} \begin{array}{cccccc} \mathrm{A} & \mathrm{B} & \mathrm{C} & \mathrm{D} & \mathrm{E} & \mathrm{F} \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

- La matrice d'adjacence du graphe complet à 4 sommets est donnée par :



$$\begin{array}{c} \\ \mathrm{A} \\ \mathrm{B} \\ \mathrm{C} \\ \mathrm{D} \end{array} \begin{array}{c} \begin{array}{cccc} \mathrm{A} & \mathrm{B} & \mathrm{C} & \mathrm{D} \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \end{array}$$

Remarque

Ces deux exemples amènent quelques remarques :

× les matrices d'adjacence de graphes **non orientés** sont symétriques. C'est logique : si $A-B \in \mathcal{A}$ on a aussi $B-A \in \mathcal{A}$.

× les diagonales des matrices d'adjacence exposées ci-dessus sont constituées uniquement de 0. Cela témoigne du fait que les graphes d'origine ne contiennent pas de **boucle** (par définition, une boucle est une arête reliant un sommet à lui-même). Il est à noter que les boucles sont tout à fait autorisées par la notion de graphes.

× la matrice d'adjacence d'un graphe complet contient des 1 partout sauf sur sa diagonale. Cela donne une nouvelle manière de déterminer le nombre d'arêtes d'un graphe complet : il suffit de compter le nombre de 1 de la matrice. Deux manières de voir les choses pour un graphe complet à n sommets :

1) il y a n^2 coefficients dans la matrice et n sur la diagonale. Il y a donc $n^2-n=n\,(n-1)$ coefficients de valeur 1 dans la matrice. Or, il y a exactement deux fois plus de coefficients 1 que d'arêtes (l'arête $\{A,B\}$ apparaît dans la matrice pour coder le lien entre A et B mais aussi pour le lien entre B et A). Il y a donc en tout : $\dfrac{n\,(n-1)}{2}$ arêtes dans un tel graphe.

2) il y a :

$$1+2+3+\ldots+(n-1)=\sum_{k=1}^{n-1} k=\frac{n\,(n-1)}{2}$$

coefficients strictement sous la diagonale de la matrice.

II. Accessibilité dans un graphe non orienté

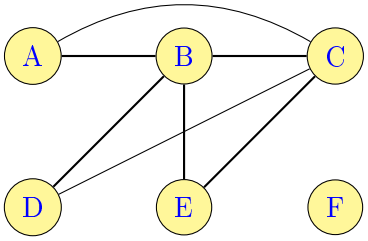
II.1. Notion de chaîne

Définition

Soit $\mathcal{G} = (S, \mathcal{A})$ un graphe. Soit $m \in \mathbb{N}$.

- Une **chaîne** est une suite finie non vide de sommets telle que chaque paire de sommets consécutifs de la suite soit une arête du graphe.
  Le premier sommet d'une chaîne est appelé la **source** de cette chaîne.
  Le dernier sommet d'une chaîne est appelé le **but** de cette chaîne.
- La longueur d'une chaîne est le nombre d'arêtes qui constituent cette chaîne.
  Ainsi, une chaîne de longueur m est la donnée d'une suite de sommets $(s_i)_{i\in[\![0,m]\!]}$ telle que :
  $$\forall i \in [\![0, m]\!],\ s_i - s_{i+1}$$
  Cette chaîne sera notée $(s_0, \dots, s_m)$ ou $s_0 - \dots - s_m$.
  On note au passage qu'une chaîne de longueur m possède $m+1$ sommets.
  Si i et j sont deux entiers de $[\![0, m]\!]$ tels que $i \leqslant j$, alors $(s_i, \dots, s_j)$ est une **sous-chaîne** de la chaîne $(s_0, \dots, s_m)$. Enfin, s'il existe une chaîne de longueur m entre deux sommets $s \in S$ et $t \in S$, on pourra noter : $s \overset{m}{\rightsquigarrow} t$.
- Une chaîne de longueur 0 est la donnée d'un sommet.
- S'il existe une chaîne d'un sommet $s \in S$ à un sommet $t \in S$, on dit que t est **accessible** depuis s (et ainsi, t est aussi accessible depuis s).
- Une chaîne est :
  - × **élémentaire** si aucun sommet n'y figure plus d'une fois, à l'exception de la source et du but qui peuvent coïncider.
  - × **simple** si aucune arête n'y figure plus d'une fois.

Exemple



- A est une chaîne élémentaire de longueur 0.
- $A - B - E$ est une chaîne élémentaire de longueur 2.
- $A - B - E - B - D - B - A - C - E$ est une chaîne non élémentaire de longeur 8.
- $A - B - E - C - A$ est une chaîne élémentaire de longeur 4.
- Le sommet F n'est accessible depuis aucun autre sommet.

II.2. Lien avec la matrice d'adjacence

Théorème 2.

Soit $\mathcal{G} = (S, \mathcal{A})$ un graphe.

On note $n = \mathrm{Card}(S)$ l'ordre de $\mathcal{G}$. Il existe donc une bijection φ entre $[\![1, n]\!]$ et S.

On note M la matrice d'adjacence de $\mathcal{G}$.

Pour tout entier naturel $p \in \mathbb{N}$ et pour tout couple $(i, j) \in [\![1, n]\!] \times [\![1, n]\!]$, on note :

× *$c_{i,j}(p)$ le nombre de chaînes de longeur p reliant les sommets $\varphi(i)$ et $\varphi(j)$.*

× *$m_{i,j}(p)$ le coefficient située à la $i^{ème}$ ligne et $j^{ème}$ colonne de M^p.*

Alors, pour tout $p \in \mathbb{N}$ et pour tout couple $(i, j) \in [\![1, n]\!] \times [\![1, n]\!]$:

$$c_{i,j}(p) = m_{i,j}(p)$$

(le coefficient (i, j) de la matrice M^p contient le nombre de chaînes de longueur p reliant $\varphi(i)$ à $\varphi(j)$)

Démonstration.
Démontrons par récurrence : $\forall p \in \mathbb{N}$, $\mathscr{P}(p)$, où $\mathscr{P}(p) : \forall (i,j) \in [\![1,p]\!] \times [\![1,p]\!]$, $c_{i,j}(p) = m_{i,j}(p)$.

- **Initialisation** :
  Rappelons tout d'abord que les seules chaînes de longueur 0 sont celles constituées seulement d'un sommet. On en conclut que pour tout $(i,j) \in [\![1,n]\!] \times [\![1,n]\!]$:

$$c_{i,j}(0) = \begin{cases} 1 & \text{si } i = j \\ 0 & \text{si } i \neq j \end{cases}$$

  Par ailleurs : $M^0 = I_n$. Ainsi, pour tout $(i,j) \in [\![1,n]\!] \times [\![1,n]\!]$, on a bien : $c_{i,j}(0) = m_{i,j}(0)$.
  D'où $\mathscr{P}(0)$.

- **Hérédité** : soit $p \in \mathbb{N}$.
  Supposons $\mathscr{P}(p)$ et démontrons $\mathscr{P}(p+1)$ (*i.e.* $\forall (i,j) \in [\![1,n]\!] \times [\![1,n]\!]$, $c_{i,j}(p+1) = m_{i,j}(p+1)$).
  Soit $(i,j) \in [\![1,n]\!] \times [\![1,n]\!]$.

$$\begin{aligned}
c_{i,j}(p+1) &= \text{nombre de chaînes de la forme } \varphi(i) \overset{p+1}{\rightsquigarrow} \varphi(j) \\
&= \sum_{r \in S} \text{nombre de chaînes de la forme } \varphi(i) \overset{p}{\rightsquigarrow} r - \varphi(j) \\
&= \sum_{k=1}^{n} \underbrace{\text{nombre de chaînes de la forme } \varphi(i) \overset{p}{\rightsquigarrow} \varphi(k) - \varphi(j)}_{= \begin{cases} c_{i,k}(p) & \text{si } \varphi(k) \text{ et } \varphi(j) \text{ sont adjacents} \\ 0 & \text{si } \varphi(k) \text{ et } \varphi(j) \text{ ne sont pas adjacents} \end{cases}} \\
&= \sum_{k=1}^{n} c_{i,k}(p) \times m_{k,j}(1) \\
&= \sum_{k=1}^{n} m_{i,k}(p) \times m_{k,j}(1) && \textit{(par hypothèse de récurrence)} \\
&= \sum_{k=1}^{n} \left(M^p\right)_{i,k} \times \left(M\right)_{k,j} && \textit{(par définition de } M^p \textit{ et } M\textit{)} \\
&= \sum_{k=1}^{n} m_{i,k}(p) \times m_{k,j}(1) \\
&= \left(M^p \times M\right)_{i,j} && \textit{(par définition du produit de matrices)} \\
&= \left(M^{p+1}\right)_{i,j} \\
&= m_{i,j}(p+1)
\end{aligned}$$

  D'où $\mathscr{P}(p+1)$.

Par principe de récurrence : $\forall p \in \mathbb{N}$, $\mathscr{P}(p)$. □

II.3. Notion de graphe connexe

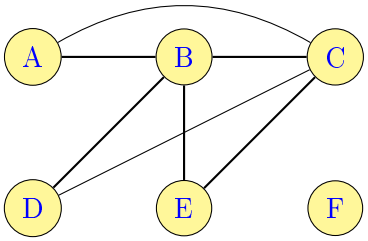
II.3.a) Définition

Définition

Soit $\mathcal{G} = (S, \mathcal{A})$ un graphe.

- Le graphe $\mathcal{G}$ est dit **connexe** s'il existe une chaîne entre tout couple de sommets.
  Autrement dit, un graphe $\mathcal{G}$ est connexe si tout sommet est accessible depuis tous les autres sommets.

Exemple



- Le Graphe 1 n'est pas connexe. En effet, il existe (au moins) un couple de sommets entre lesquels il n'y a pas de chaîne. Par exemple, il n'existe pas de chaîne entre A et F.
- Le sous-graphe ne contenant pas le sommet F est par contre connexe : il existe une chaîne entre tout couple de sommets.

II.3.b) Parcours en profondeur

Exploration en profondeur d'un graphe $\mathcal{G}$ à partir d'un sommet v
Considérons un graphe $\mathcal{G}$ et un sommet v.
La fonction d'exploration est un processus de visite des sommets de $\mathcal{G}$.
Ce processus commence par la visite du sommet v (on marque le sommet v).
Puis, à chaque étape, on visite alors un sommet qui :

× n'a pas encore été visité,

× est relié par une arête au dernier sommet visité.

En pseudo-code, l'algorithme de la fonction d'exploration est le suivant.

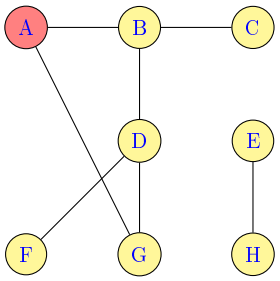
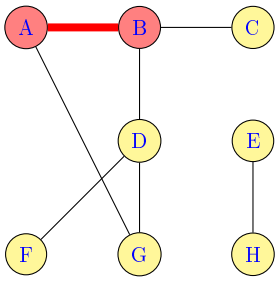
```
1  Explorer(Gr, v) :
2      Si v n'est pas marqué :
3          Marquer(v)
4          Pour tout sommet w adjacent à v et non encore marqué :
5              Explorer(Gr, w)
```

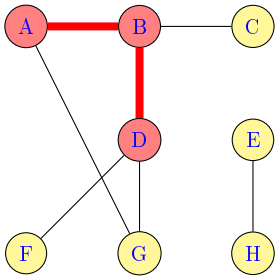
On a noté `Gr` le graphe $\mathcal{G}$ considéré afin de ne pas le confondre avec un éventuel sommet G.

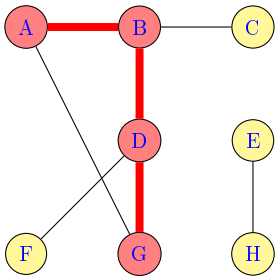
Remarque

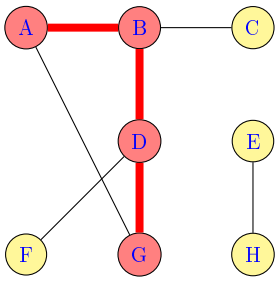
- On dit que l'on procède en profondeur car la visite privilégie toujours un sommet qui est relié par une arête au précédent sommet visité.
- La fonction `Explorer` est récursive. Cela signifie que lors de son exécution, elle fait appel à elle-même. Nous avons déjà rencontré des fonctions de ce type. Par exemple, la fonction factorielle à une définition récursive ($0! = 1$ et pour tout $n \in \mathbb{N}^*$, $n! = n \times (n-1)!$) et il est donc naturel d'en proposer une implémentation récursive.
  Lorsqu'on procède ainsi, il est important de se poser la question de la terminaison. Pour la fonction `Explorer`, la terminaison est assurée par le fait que le graphe possède un nombre fini de sommets. Il n'est pas possible de faire une infinité d'appels à `Explorer` puisque les appels ne se font que sur des sommets non marqués et qu'ils provoquent le marquage.
- Pour bien comprendre la manière dont l'exécution se produit, on propose dans l'illustration suivante d'exhiber la pile d'appels.

Exemple *(algorithme d'exploration sur un exemple)*
On détaille l'algorithme d'exploration du graphe $\mathcal{G}$ suivant à partir du sommet A. Initialement, aucun sommet n'est marqué. Un sommet marqué apparaîtra en rouge sur le graphe.

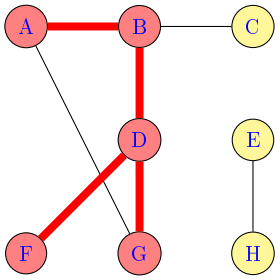
Pile d'appel	Explications	Graphe
`Explorer(Gr,A)`	• Le sommet A n'est pas marqué, on le marque. • L'entrée dans la boucle for provoque l'appel à la fonction `Explorer` pour un sommet adjacent à A et non encore visité : par exemple B.	
`Explorer(Gr,B)` `Explorer(Gr,A)`	• Le sommet B n'étant pas marqué, on le marque. • L'entrée dans la boucle for provoque l'appel à la fonction `Explorer` pour un sommet adjacent à B et non encore visité : par exemple D.	

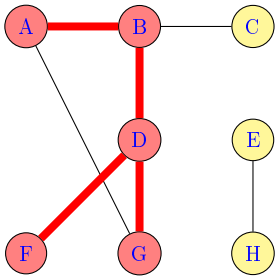
`Explorer(Gr,D)`
`Explorer(Gr,B)`
`Explorer(Gr,A)`
	• Le sommet D n'étant pas marqué, on le marque.
	• L'entrée dans la boucle for provoque l'appel à la fonction `Explorer` pour un sommet adjacent à D et non encore visité : par exemple G.
		

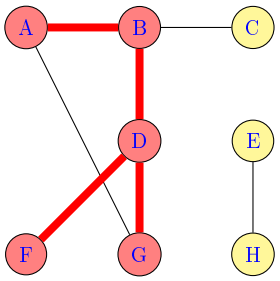
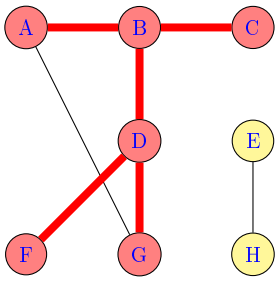
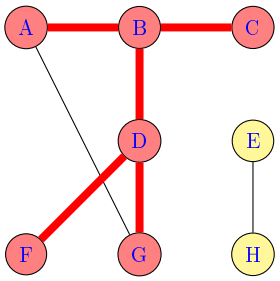
`Explorer(Gr,G)`
`Explorer(Gr,D)`
`Explorer(Gr,B)`
`Explorer(Gr,A)`
	• Le sommet G n'étant pas marqué, on le marque.
	• L'entrée dans la boucle for provoque l'appel à la fonction `Explorer` pour un sommet adjacent à G et non encore visité : il n'y en a pas !
	• Ainsi, le bloc en haut de la pile est dépilé.
		

`Explorer(Gr,D)`
`Explorer(Gr,B)`
`Explorer(Gr,A)`
	• Le sommet D est déjà marqué.
	• L'entrée dans la boucle for provoque l'appel à la fonction `Explorer` pour un sommet adjacent à D et non encore visité : il n'y a que F.
		

Pile d'appel	Explications	Graphe

`Explorer(Gr,F)`
`Explorer(Gr,D)`
`Explorer(Gr,B)`
`Explorer(Gr,A)`
	• Le sommet F n'est pas marqué, on le marque.
	• L'entrée dans la boucle for provoque l'appel à la fonction `Explorer` pour un sommet adjacent à F et non encore visité : il n'y en a pas !
	• Ainsi, le bloc en haut de la pile est dépilé.
		

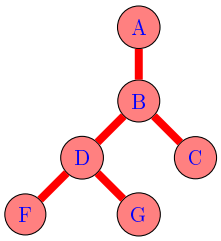
`Explorer(Gr,D)`
`Explorer(Gr,B)`
`Explorer(Gr,A)`
	• Le sommet D est déjà marqué.
	• L'entrée dans la boucle for provoque l'appel à la fonction `Explorer` pour un sommet adjacent à D et non encore visité : il n'y en a pas !
	• Ainsi, le bloc en haut de la pile est dépilé.
		

`Explorer(Gr,B)` `Explorer(Gr,A)`	• Le sommet B est déjà marqué. • L'entrée dans la boucle for provoque l'appel à la fonction `Explorer` pour un sommet adjacent à B et non encore visité : il n'y a que C.	
`Explorer(Gr,C)` `Explorer(Gr,A)`	• Le sommet C n'est pas marqué, on le marque. • L'entrée dans la boucle for provoque l'appel à la fonction `Explorer` pour un sommet adjacent à C et non encore visité : il n'y en a pas ! • Ainsi, le bloc en haut de la pile est dépilé.	
`Explorer(Gr,A)`	• Le sommet A est déjà marqué. • L'entrée dans la boucle for provoque l'appel à la fonction `Explorer` pour un sommet adjacent à A et non encore visité : il n'y en a pas ! • Ainsi, le bloc en haut de la pile est dépilé.	

La pile d'exécution est alors vide ce qui met fin à l'algorithme.

Remarque

- Il n'existe pas qu'une seule manière d'explorer un graphe à partir d'un sommet. Lors de l'appel `Explorer(Gr,A)`, un choix est fait parmi les sommets adjacents à A et non encore marqués. Ce choix dépend de l'implémentation qui est faite de la fonction d'exploration.
- Dans l'exemple précédent, on a fait apparaître en rouge les sommets visités ainsi que les arêtes qui ont permis de mener à ces sommets. Le graphe obtenu est connexe et acyclique.
  C'est donc, par définition, un **arbre**.



  - Le sommet en haut de l'arbre est appelé **racine**.
  - Les arêtes définissent une relation de descendance (resp. ascendance). Plus précisément, lorsqu'une arête relie un sommet s_1 à un sommet s_2 à un niveau plus bas, on dit que s_1 est le **père** de s_2 (resp. s_2 est un **fils** de s_1).

- Deux sommets n'ont pas été visités lors de l'étape d'exploration. Le graphe n'a donc pas été complètement parcouru. L'algorithme du parcours en profondeur réalise une visite complète de tous les sommets. Pour ce faire, on fait appel à la fonction d'exploration pour l'un des deux sommets non encore marqués (E et H).

Algorithme de parcours en profondeur

L'algorithme du parcours en profondeur consiste à appeler successivement la fonction d'exploration à chacun des sommets qui n'a pas encore été visité.

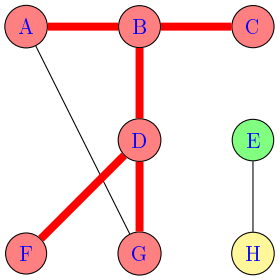
```
1  Parcours_Profondeur(Gr) :
2      On initialise tous les sommets comme non marqués
3      Pour tout sommet u non marqué :
4          Explorer(Gr, u)
```

Exemple

La parcours en profondeur commence par l'exploration depuis le sommet A. Cette exploration a déjà été décrite. Il reste alors à lancer la fonction d'exploration à partir d'un sommet non encore exploré : E par exemple.

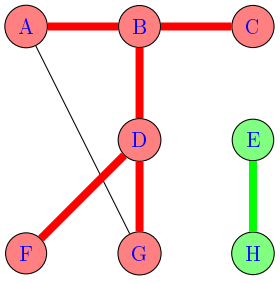
`Explorer(Gr,E)`

- Le sommet E n'est pas marqué, on le marque.
- L'entrée dans la boucle for provoque l'appel à la fonction `Explorer` pour un sommet adjacent à E et non encore visité : il n'y a que F.



`Explorer(Gr,H)`
`Explorer(Gr,E)`

- Le sommet H n'est pas marqué, on le marque.
- L'entrée dans la boucle for provoque l'appel à la fonction `Explorer` pour un sommet adjacent à H et non encore visité : il n'y en a pas.
- Ainsi, le bloc en haut de pile est dépilé.



Le bloc `Explorer(Gr,E)` est à son tour dépilé. La pile est alors vide et l'algorithme prend fin.

II.3.c) Premières applications du parcours en profondeur

Les applications de l'algorithme du parcours en profondeur sont nombreuses :

× si le premier appel de la fonction d'exploration ne marque pas tous les sommets, on peut affirmer que le graphe n'est pas connexe.

× lors d'un parcours en profondeur, on réalise plusieurs appels à la fonction d'exploration. À la fin de chaque appel, le sous-graphe obtenu en ne conservant que les sommets qui ont été visités lors de cette exploration, est un graphe connexe. Un parcours en profondeur permet donc de repérer les **composantes connexes** d'un graphe.

II.4. Cycle d'un graphe

II.4.a) Définition

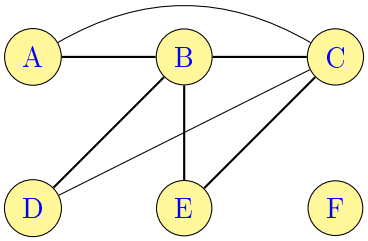
Définition

Soit $\mathcal{G} = (S, \mathcal{A})$ un graphe.

- Un **cycle** est une chaîne simple, qui comporte au moins une arête, et dont le sommet de départ et d'arrivée sont les mêmes.
- On dit que $\mathcal{G}$ est **acyclique** s'il ne contient pas de cycle.
- **Caractère eulérien**
  - × Une chaîne **eulérienne** est une chaîne qui passe par toutes les arêtes de $\mathcal{G}$ exactement une fois.
  - × Un **cycle eulérien** est une chaîne eulérienne qui est un cycle.
  - × On dit de $\mathcal{G}$ qu'il est **eulérien** s'il admet un cycle eulérien.
- **Caractère hamiltonien**
  - × Une chaîne **hamiltonienne** est une chaîne qui passe par tous les sommets de $\mathcal{G}$ exactement une fois.
  - × Un **cycle hamiltonien** est une chaîne hamiltonienne qui est un cycle.
  - × On dit de $\mathcal{G}$ qu'il est **hamiltonien** s'il admet un cycle hamiltonien.
- Un graphe qui ne possède qu'un sommet et qui ne possède pas d'arête est considéré eulérien.
- Un graphe qui ne possède qu'un sommet et qui ne possède pas d'arête est considéré hamiltonien.

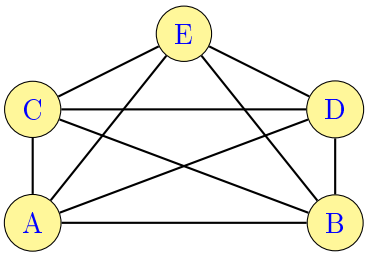
Exemple

On reprend l'exemple du Graphe 1.



Graphe 1

- Le Graphe 1 est eulérien : en effet, $A - B - D - C - E - B - C - A$ est un cycle eulérien.
- Ce graphe n'est pas hamiltonien. En effet, il n'existe pas de cycle passant par tous les sommets car le sommet F ne possède pas d'arête incidente.



Graphe 2

- Le Graphe 2 est eulérien : en effet, $A - B - D - E - C - A - D - C - B - E - A$ est un cycle eulérien.
- Le Graphe 2 est hamiltonien : en effet, $A - B - D - E - C - A$ est un cycle hamiltonien.

II.4.b) Une nouvelle application du parcours en profondeur : la détection de cycle

Théorème 3.

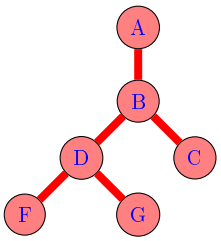
Soit $\mathcal{G} = (S, \mathcal{A})$ un graphe.

On suppose que tous les sommets de $\mathcal{G}$ sont de degré supérieur ou égal à 2.

Alors le graphe $\mathcal{G}$ possède au moins un cycle (élémentaire).

Démonstration.
Soit s_1 un sommet de $\mathcal{G}$.
On exécute alors l'algorithme d'exploration de $\mathcal{G}$ depuis le sommet s_1. On obtient alors un marquage successif des sommets accessibles depuis s_1. Notons s_2, ..., s_k, ces sommets.

- Le fait que les sommets sont marqués dans l'ordre s_1, ..., s_k, ne signifie pas pour autant qu'il existe une arête entre deux de ces sommets consécutifs.
- Pour bien le comprendre, reprenons le parcours réalisé précédemment. Lors de ce parcours, les sommets ont été marqués dans l'ordre suivant : A, B, D, G, F, C. Il n'existe pas d'arête entre F et C. Pour rappel, le sommet C est marqué à la fin de l'exploration du sommet B (et il existe donc une arête $B-C$) tandis que le sommet F est marqué lors de l'exploration du sommet D (et il existe donc une arête $D-F$). Rappelons que le graphe obtenu en conservant sommets marqués et arêtes parcourues pendant l'algorithme est un arbre. On obtient l'arbre suivant dans le graphe précédent :



S'il n'y a pas d'arête entre F et C, on lit très bien ici que F a été marqué par D, lui-même marqué par B. Il existe donc un chaîne élémentaire $F-D-B$. Le sommet C est, quant à lui, marqué par le sommet B. Ainsi, $B-C$ est un chaîne (c'est même une arête puisque cette chaîne est de longueur 1). Finalement, on a démontré qu'il existe une chaîne élémentaire $F-D-B-C$ qui relie F à C.

Le sommet s_k possède les propriétés suivantes :

× il a été marqué lors de l'exploration d'un des sommets s_1, ..., s_{k-1}. Ainsi, il existe $i \in [\![1, k-1]\!]$ tel que $s_i - s_k$ (s_i apparaît comme le père de s_k dans l'arbre construit à l'issue de l'algorithme).

× il est de degré supérieur ou égal à 2, il existe forcément une arête incidente à s_k autre que celle qui a permis son marquage. Par définition de s_k (dernier sommet marqué lors de l'exploration à partir de s_1), cette nouvelle arête ne peut mener vers un sommet non encore marqué. On en conclut que cette arête mène vers un sommet déjà marqué.
Ainsi, il existe $j \in [\![1, k-1]\!]$, tel que $s_k - s_j$. Cette arête étant différente de l'arête de marquage, on a de plus $j \neq i$.

Enfin, d'après la remarque encadrée ci-dessus, il existe une chaîne élémentaire entre s_j et s_i.
En concaténant cette chaîne $s_j - \ldots - s_i$ à l'arête $s_k - s_j$, on obtient alors une chaîne élémentaire $s_j - \ldots - s_i - s_k - s_j$ qui constitue un cycle. □

II.4.c) Détection de cycles eulériens

Théorème 4.

Soit $\mathcal{G} = (S, \mathcal{A})$ un graphe.

On suppose que $\mathcal{G}$ n'admet pas de boucle (arête qui relie un sommet à lui-même).

On suppose que $\mathcal{G}$ est connexe.

Le graphe $\mathcal{G}$ est eulérien $\Leftrightarrow$ *Chaque sommet de $\mathcal{G}$ est de degré pair*

Démonstration.
On procède par double implication.

($\Rightarrow$) Supposons que $\mathcal{G}$ est eulérien. Il admet donc un cycle eulérien.
Par définition, ce cycle est une chaîne simple (aucune arête n'y figure plus d'une fois) :

- × qui a le même sommet source et but,
- × qui passe par toutes les arêtes de $\mathcal{G}$.

Rappelons que cette chaîne :

- × n'est pas forcément élémentaire. En effet, un sommet peut y figurer plus d'une fois.
- × ne regroupe pas forcément tous les sommets de $\mathcal{G}$. Un sommet s qui n'y figure pas est un sommet de degré 0 (qui est bien un nombre pair). En effet, s ne peut posséder d'arête incidente : si c'était le cas, cette arête ferait partie de la chaîne eulérienne.

On se sert alors de cette chaîne pour déterminer le degré de chaque sommet qui y figure.

Notons n la longueur cette chaîne. Il existe alors n sommets (pas forcément distincts !) notés s_0, …, s_{n-1} tels que : $s_0 - s_1 - \ldots - s_{n-1} - s_0$. Parcourons cette chaîne :

- × on rencontre tout d'abord s_1. Une arête y amène $(s_0 - s_1)$ et une arête permet de poursuivre le parcours $(s_1 - s_2)$. Ainsi, le sommet s_1 possède au moins deux arêtes incidentes. Ce sommet peut être à nouveau rencontré lors du parcours. À chaque fois que c'est le cas, on trouve deux nouvelles arêtes incidentes à s_1 : une qui y mène et l'autre qui permet de continuer le parcours. Ainsi, à chaque nouvelle occurrence de s_1, on trouve 2 nouvelles arêtes incidentes à s_1. À la fin du parcours, toutes les arêtes ont été visitées (puisque le cycle contient toutes les arêtes du graphe) et s_1 a donc un nombre pair d'arêtes incidentes.
- × il en est de même de tous les sommets s_2, …, s_{n-1}.
- × il reste alors à traiter le cas du sommet s_0. Ce sommet peut être présent lors du parcours (dans la chaîne $s_1 - \ldots - s_{n-1}$) et on a donc dénombré jusque là un nombre pair d'arêtes incidentes de s_0. Ce sommet admet deux autres arêtes : une qui y mène $(s_{n-1} - s_0)$ et celle qui permet de débuter le parcours $(s_0 - s_1)$. Le sommet s_0 admet alors bien un nombre pair d'arêtes incidentes : celles dénombrées jusque là plus les deux qu'on vient d'ajouter.

Toutes les arêtes ayant été visitées, on a bien trouvé le degré de chaque sommet composant le cycle.

($\Leftarrow$) On démontre par récurrence forte : $\forall n \in \mathbb{N}$, $\mathscr{P}(n)$
où $\mathscr{P}(n)$: tout graphe $\mathcal{G}$ connexe et sans boucle qui possède n arêtes et dont chaque sommet est de degré pair est eulérien.

- **Initialisation** :
  Soit $\mathcal{G}$ un graphe connexe et sans boucle à 0 arête et dont chaque sommet est de degré pair. Alors $\mathcal{G}$ possède un seul sommet (sinon il ne serait pas connexe). Un tel graphe est eulérien.
  D'où $\mathscr{P}(0)$.
- **Hérédité** : soit $n \in \mathbb{N}$.
  On suppose que la propriété est vraie pour tout $i \in [\![0, n]\!]$ et on démontre $\mathscr{P}(n+1)$.

  Soit $\mathcal{G}$ un graphe connexe et sans boucle, qui possède $n+1$ arêtes et dont chaque sommet est de degré pair.

  - Les sommets de ce graphe étant de degré supérieur ou **égal à** 2, il possède un cycle élémentaire. Notons $r \in \mathbb{N}^*$ la longueur de ce cycle. Il existe alors r sommets (deux à deux distincts) s_1, ..., s_r tels que : $s_1 - \ldots - s_r - s_1$.
    On considère alors le sous-graphe de $\mathcal{G}$ obtenu en supprimant les arêtes présentes dans ce cycle. On note $\mathcal{G}'$ ce graphe.
  - Remarquons tout d'abord que les sommets de $\mathcal{G}'$ sont tous de degré pair. En effet, l'élimination des arêtes du cycle :
    - × n'a aucun effet sur le degré des sommets qui ne sont pas présents dans le cycle.
    - × diminue de 2 (une arête qui mène au sommet et une qui mène au sommet suivant) le degré des sommets présents dans le cycle.
  - L'idée naturelle est alors d'appliquer l'hypothèse de récurrence à $\mathcal{G}'$: c'est un graphe sans boucle, dont tous les sommets sont de degré pair, et dont le nombre d'arêtes est dans $[\![0, n]\!]$. Cependant, ce graphe n'est pas forcément connexe !
    L'idée est alors d'appliquer l'hypothèse de récurrence à chacune des composantes connexes de ce graphe. Par hypothèses de récurrences, chaque composante connexe de $\mathcal{G}'$ possède un cycle eulérien.
  - Pour obtenir le cycle eulérien démontrant la propriété souhaitée, combine alors le cycle initial avec les cycles de chaque composante connexe.
    Plus précisément, on parcourt le cycle $s_1 - \ldots - s_r - s_1$ dans l'ordre des sommets s_1, ..., s_r. Lors du parcours d'un sommet s_i (avec $i \in [\![1, r]\!]$), deux cas se présentent :
    - × soit s_i est seul dans sa composante connexe de $\mathcal{G}'$. On passe alors au sommet suivant s_{i+1}.
    - × soit s_i est un sommet d'une composante connexe de $\mathcal{G}'$ (non réduite à un sommet) non encore visitée. Dans ce cas, on insère le cycle eulérien liée à cette composante connexe. Ce cycle nous ramène sur s_i et on peut alors passer au sommet suivant s_{i+1}.

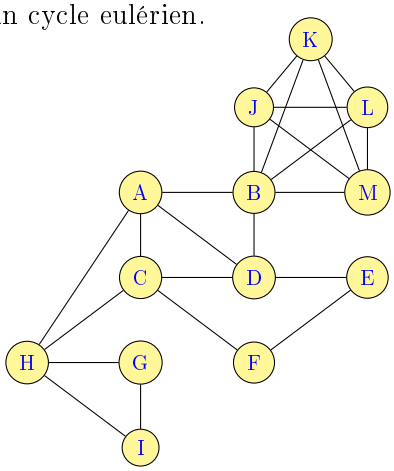
    Le cycle ainsi construit est un cycle eulérien de $\mathcal{G}$.

  D'où $\mathscr{P}(n+1)$.

Par principe de récurrence : $\forall n \in \mathbb{N}$, $\mathscr{P}(n)$.
On en conclut alors que tout graphe connexe et sans boucle dont chaque sommet est de degré pair est bien eulérien. □

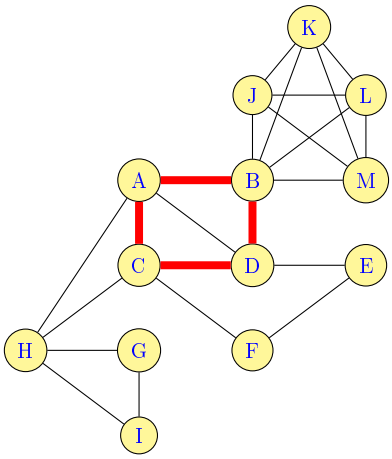
Illustration de « l'algorithme de construction d'un cycle eulérien »

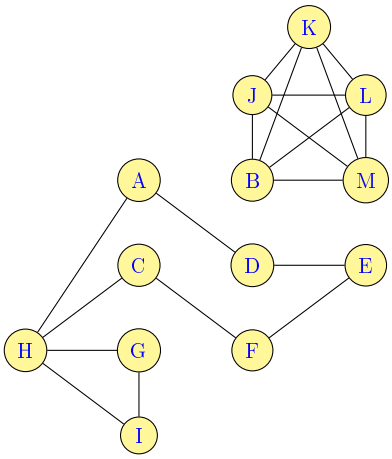
On considère le graphe $\mathcal{G}$ défini ci-dessous. Ce graphe est connexe, sans boucle et tous ses sommets sont de degré pair. Il admet donc un cycle eulérien.



Le sens réciproque de la démonstration commence par l'exhibition d'un cycle de $\mathcal{G}$. Illustrons l'algorithme de construction d'un cycle eulérien en fonction du choix du cycle initial.

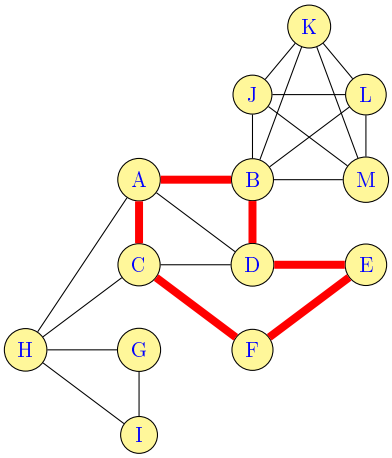
1) Si on choisit initialement le cycle $A-B-C-D-A$.

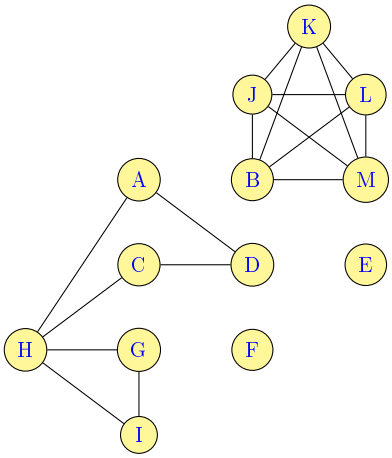




Le graphe $\mathcal{G}'$ obtenu a 2 composantes connexes toutes eulériennes

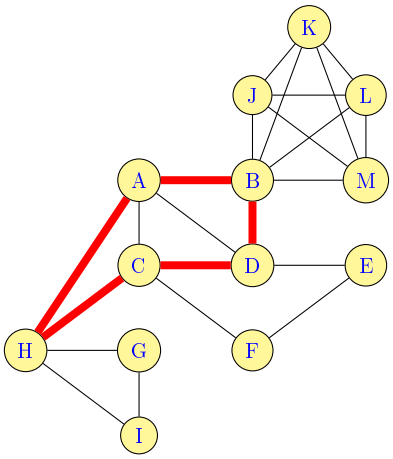
2) Si on choisit initialement le cycle $A-B-D-E-F-C-A$.

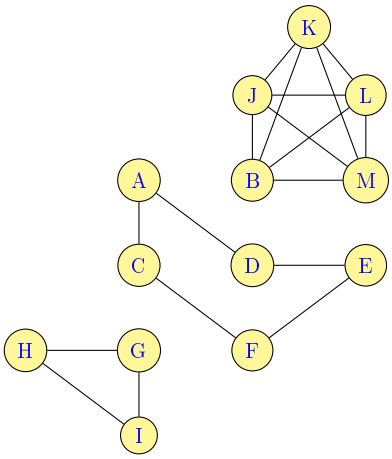




Le graphe $\mathcal{G}'$ obtenu a 4 composantes connexes toutes eulériennes

3) Si on choisit initialement le cycle $H - A - B - D - C - H$.





Le graphe $\mathcal{G}'$ obtenu a 3 composantes connexes toutes eulériennes

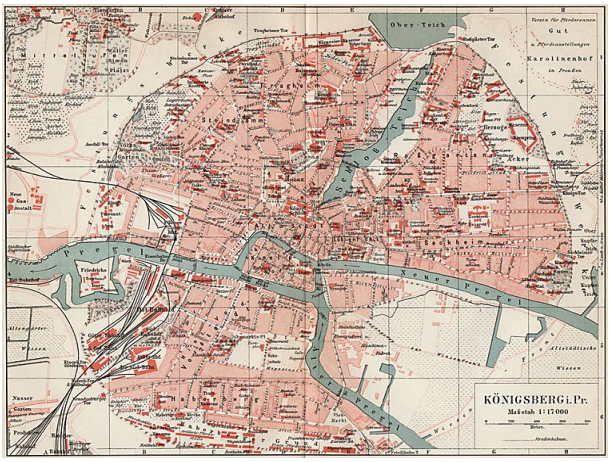
4) Autre choix (sur proposition des élèves !).

II.4.d) Application : le problème des ponts de Königsberg

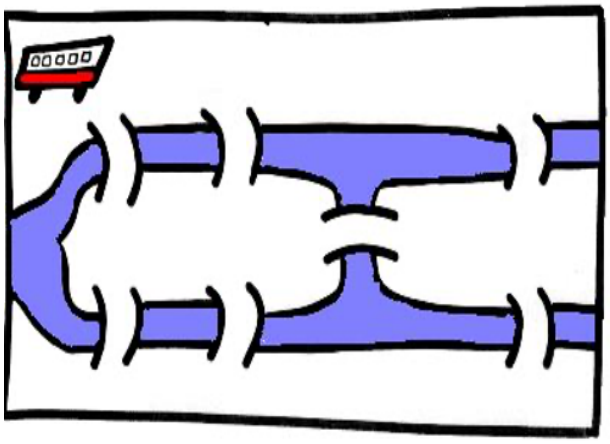
La ville de Königsberg (aujourd'hui nommée Kaliningrad) est une ville de Russie qui a la particularité de posséder 7 ponts (dont 5 mènent au Kneiphof, l'île centrale de la ville).
Cette particularité a donné lieu à la question suivante : existe-t-il une promenade permettant d'effectuer une unique fois la traversée de chacun des ponts et permettant de revenir au point initial ?

Cette interrogation est restée célèbre car est considérée comme étant à l'origine de la théorie des graphes. Cette origine ainsi que la résolution du problème (en 1735) est accordée au mathématicien Leonhard Euler qui fut le premier à apporter une modélisation mathématique du problème.

- Nous présentons ci-dessous la carte de Königsberg en 1905.

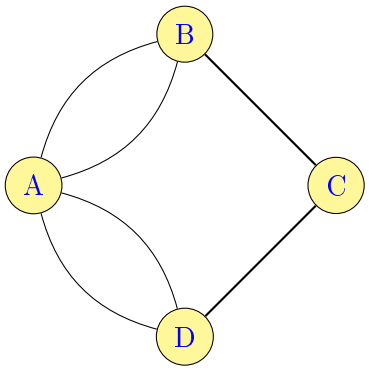


Carte de la ville en 1905



Première modélisation

- **Modélisation sous forme de graphe** : chaque zone est représentée par un sommet et chaque pont entre deux zones définit une arête entre les sommets correspondants. On obtient le graphe suivant :



Modélisation sous forme de graphe

Ce graphe étant établi, la question de la promenade revient à savoir si ce graphe possède un cycle eulérien.
On est dans le cadre d'application du théorème : le graphe est connexe et sans boucle. On en déduit qu'il est eulérien si et seulement si tous ses sommets sont de degré pair. Or, le sommet B est de degré impair égal à 3 (il en est de même de D. On en conclut que ce graphe n'est pas eulérien. Il n'existe donc pas de promenade permettant de traverser une unique fois tous les points de Königsberg !

III. Notion de graphe orienté

- La notion de graphe orienté diffère de celle de graphe non orienté par le fait que les arêtes d'un graphe orienté, appelées des arcs, on un sens de parcours : l'existence d'un arc $A \to B$ signifie que l'on peut passer du sommet A au sommet B mais pas forcément du sommet B au sommet A.
- La notion de graphe orienté donne lieu à un vocabulaire spécifique.

Définition

Soit S un ensemble fini.

- Un graphe orienté $\mathcal{G}$ est la donnée d'un couple $\mathcal{G} = (S, \mathcal{A})$ où :
  - S est appelé ensemble des **sommets** du graphe $\mathcal{G}$,
  - $\mathcal{A} \subset S \times S$ est appelé ensemble des **arcs** du graphe $\mathcal{G}$.

Vocabulaire

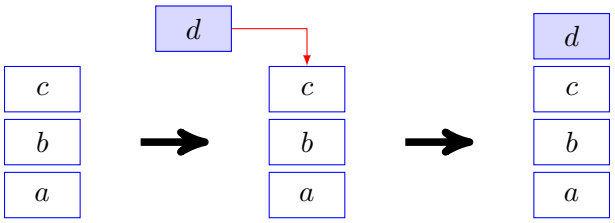
- On appelle **ordre** du graphe $\mathcal{G}$ le nombre de sommets de $\mathcal{G}$.
  Autrement dit, l'ordre de $\mathcal{G}$ est le cardinal de l'ensemble S.
- Si $(A, B) \in \mathcal{A}$ on dit que l'arc (A, B) relie les sommets A et B ou encore que l'arc (A, B) est **incident** aux sommets A et B. L'arc $(A, B) \in \mathcal{A}$ peut être notée $A \to B$.
  Un tel arc est dit **sortant** pour le sommet A et **entrant** pour le sommet B.
- On appelle **degré** d'un sommet le nombre d'arcs incidents à ce sommet.
  On peut distinguer le **degré entrant** d'un sommet s, noté $d^-(s)$ qui est le nombre d'arcs entrant vers s et le **degré sortant** de s, noté $d^+(s)$ qui est le nombre d'arcs sortant de s.
- On dit que deux sommets sont **adjacents** s'ils sont reliés par un arc.
- La notion d'arêtes étant remplacées par la notion d'arcs, la notion de chaîne est quant à elle nommée **chemin** (un chemin est une suite finie non vide de sommets telle que chaque paire de sommets consécutifs de la suite soit un arc du graphe).
- De même, dans un graphe orienté, on parle de préférence de la notion de **circuit** en lieu et place de la notion de cycle (un circuit est une chemin simple, qui comporte au moins une arête, et dont le sommet de départ et d'arrivée sont les mêmes).

IV. Structures de données et parcours itératifs

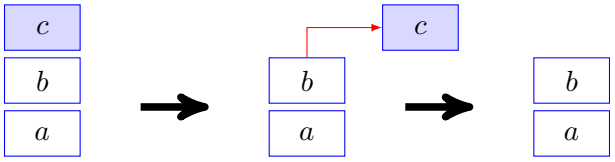
IV.1. Structures séquentielles

IV.1.a) Pile

Une **pile** (*stack* en anglais) est une structure de données linéaire qui se distingue par ses conditions d'accès et d'ajout d'éléments : c'est le principe du « dernier arrivé, premier servi » (principe du LIFO : *Last In, First Out*). Un peu comme une pile d'assiettes, c'est la dernière assiette posée sur une pile qui sera la première utilisée.



Action d'empiler



Action de dépiler

Une telle structure est toujours munie des opérations suivantes :

× une fonction de création d'une pile vide,

× une fonction déterminant si une pile est vide,

× une fonction permettant d'empiler un élément au sommet de la pile (*push*),

× une fonction permettant de dépiler et de renvoyer l'élément au sommet d'une pile non vide (*pop*),

× une fonction permettant de connaître l'élément en haut d'une pile non vide.

Implémentation d'une pile sous Python

Une liste **Python** permet d'implémenter une pile LIFO :

× la création d'une pile vide s'effectue à l'aide de la commande : `[]`,

× le test déterminant si une pile `L` est vide s'effectue à l'aide de la commande : `L == []`,

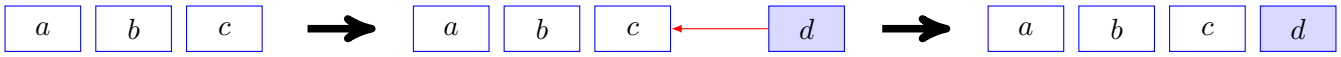
× l'empilage s'effectue à l'aide de la méthode `append`,

× le dépilage s'effectue à l'aide de la méthode `pop`,

× l'accès à l'élément en haut de la pile s'effectue à l'aide de l'appel `L[-1]`.

IV.1.b) File

Une **file** (*queue* en anglais) est une structure de données linéaire qui se distingue par ses conditions d'accès et d'ajout d'éléments : c'est le principe du « premier arrivé, premier servi » (principe du FIFO : *First In, First Out*). Un peu comme file d'attente, c'est le premier client arrivé à la caisse qui sera le premier servi, et donc le premier à partir de la file.



Action d'enfiler

Action de défiler

Une telle structure est toujours munie des opérations suivantes :

× une fonction de création d'une file vide,

× une fonction déterminant si une file est vide,

× une fonction permettant d'enfiler un élément à droite de la file,

× une fonction permettant de défiler et de renvoyer l'élément à gauche d'une file non vide,

× une fonction permettant de connaître l'élément à gauche d'une file non vide.

Implémentation d'une file sous Python

On pourrait penser à implémenter une file à l'aide d'une liste. L'action de défiler le premier élément d'une liste `L` s'effectuerait alors avec la commande `L.pop(0)`. Cependant, la complexité de cet appel est en $\underset{n\to+\infty}{O}(n)$ (où n est la longueur de la liste `L`), au lieu du $\underset{n\to+\infty}{O}(1)$ souhaité.

On utilisera plutôt une *double ended queue* ou *deque* telle qu'elle est disponible dans le module `collections`.

```
import collections
```

× La commande suivante permet de créer une file vide.

```
file = collections.deque()
```

× La commande suivante permet de créer une file à partir d'une liste.

```
L = collections.deque([1,2,3,4,5])
```

× La fonction `len` permet de connaître la longueur de la file.

```
n = len(file)
```

× La méthode `append` permet d'enfiler l'élément `x` à droite de la file.

```
file.append(x)
```

× La méthode `popleft` permet de défiler et renvoyer l'élément à gauche de la file.

```
y = file.popleft()
```

IV.1.c) File de priorité

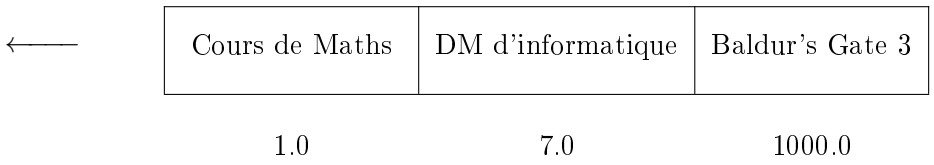
Une **file de priorité** (*priority queue* en anglais) est une structure de donnée linéaire où chaque élément de la file possède une priorité. Intuitivement, c'est avec une telle structure de données que l'on fonctionne pour dresser une liste de tâches. Supposons que l'on ait plusieurs tâches à accomplir, listées par importance décroissante :

× travailler le cours de maths du jour,

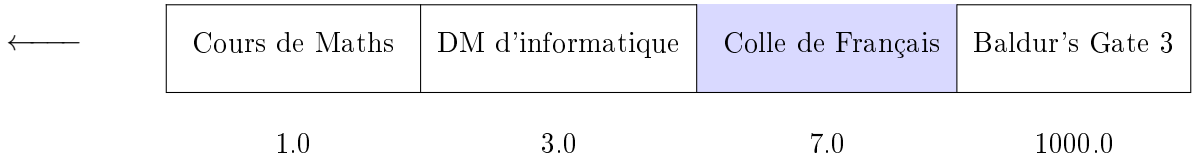
× finir le DM d'informatique pour la semaine prochaine,

× poursuivre votre quête en cours sur Baldur's Gate 3.

On peut se représenter cet ensemble de tâches comme suit.



La flèche indique ici que la prochaine tâche à accomplir est de relire votre cours de maths. Les nombres flottants placés en dessous de chaque tâche sont des priorités : plus ce nombre est bas, plus la tâche est prioritaire.
Si l'on apprend maintenant que la prochaine colle de français a lieu dans trois jours, il faut se ménager du temps pour la préparer. On insère alors cette nouvelle tâche dans la file avec une priorité de 3.0.



Une telle structure est toujours munie des opérations suivantes :

× une fonction de création d'une file de priorité vide,

× une fonction déterminant si une file de priorité est vide,

× une fonction permettant d'enfiler un élément accompagné d'une priorité,

× une fonction permettant de défiler l'élément ayant la priorité la plus basse,

× une fonction permettant de connaître l'élément ayant la priorité la plus basse.

Notons que l'élément enfilé ne permet pas forcément d'obtenir une file triée par ordre de priorité. Plus précisément, dans une file de priorité :

× le premier élément est toujours celui de priorité la plus faible,

× les autres ne sont pas triés par ordre croissant. Il y a quand même un tri effectué mais à l'aide d'un tas binaire, mais nous nous écartons du programme.

V. Parcours de graphes

V.1. Parcours en profondeur : une utilisation des piles

On cherche ici à implémenter l'algorithme de parcours en profondeur sous **Python**.
On suppose ici que l'on dispose d'un graphe $\mathcal{G}$ à n sommets numérotés de 0 à $n-1$. On stockera sa matrice d'adjacence à l'aide d'une liste de listes.

- Écrire une commande permettant d'obtenir, à partir de la matrice d'adjacence du graphe $\mathcal{G}$, la liste d'adjacence de $\mathcal{G}$: la liste donc le $i^{\text{ème}}$ élément est la liste des sommets adjacents au sommet i.

```
[[i for i in range(len(L)) if L[i] == 1] for L in M]
```

- Écrire une fonction `profondeur` qui prend en paramètre sommet `s` du graphe $\mathcal{G}$ et la liste d'adjacence `L_adj` de ce graphe.
  - × On utilisera une liste `visite` de longueur n, dont tous les éléments sont initialisés à `False`. L'élément d'indice i de cette liste sera mis à jour lorsque le sommet numéro i sera visité.
  - × On utilisera une liste `pile` qui contiendra la liste des sommets en cours de d'exploration.

```
def profondeur(s, L_adj) :
    n = len(L_adj)
    visite = [False for k in range(n)]
    pile = [s]
    while len(pile) != 0
        x = pile.pop()
        if visite[x] == False :
            visite[x] = True
            for y in L_adj[x] :
                pile.append(y)
```

  On note que tous les sommets accessibles du sommet `s` vont être marqués. Si tous les sommes ont été marqués à la fin de l'exécution de la fonction, alors on peut en conclure que le graphe $\mathcal{G}$ étudié est connexe.

- Proposer une implémentation récursive de l'algorithme de parcours en profondeur.

```
def profondeur_rec(L_adj, visite, x) :
    if visite[x] == False :
        visite[x] = True
        for y in L_adj[x] :
            profondeur_rec(L_adj, visite, y)

def profondeur(L_adj, s) :
    n = len(L_adj)
    visite = [False for k in range(n)]
    profondeur_rec(L_adj, visite, s)
```

- Écrire enfin une fonction `parcours_profondeur` qui effectue le parcours en profondeur complet d'un graphe $\mathcal{G}$.

```
def parcours_profondeur(L_adj) :
    n = len(L_adj)
    visite = [False for k in range(n)]
    for s in range(n) :
        if visite[s] == False :
            profondeur_rec(L_adj, visite, s)
```

- Écrire une fonction `existe_chemin` qui prend en paramètre la liste d'adjacence `L_adj` d'un graphe $\mathcal{G}$ et deux sommets `x` et `y` de ce graphe, et qui renvoie `True` si `y` est accessible de `x` et `False` sinon. On pourra utiliser la fonction `profondeur_rec` définie précédemment.

```
def existe_chemin(L_adj, x, y) :
    n = len(L_adj)
    visite = [False for k in range(n)]
    profondeur_rec(L_adj, visite, x)
    return visite[y]
```

- Écrire une fonction `tous_vrai` qui prend en paramètre une liste `L` de booléens et qui renvoie `True` si tous les éléments de `L` sont les booléens `True` et renvoie `False` sinon.

```
def tous_vrai(L) :
    for el in L :
        if not el :
            return False
    return True
```

- En déduire une fonction `est_connexe` qui prend en paramètre la liste d'adjacence d'un graphe $\mathcal{G}$ et qui renvoie `True` si le graphe $\mathcal{G}$ est connexe et `False` sinon.

```
def est_connexe(L_adj) :
    n = len(L_adj)
    visite = [False for k in range(n)]
    profondeur_rec(L_adj, visite, 0)
    return tous_vrai(visite)
```

V.2. Parcours en largeur : une utilisation des files

Le principe du parcours en largeur est celui d'une exploration en traitant d'abord les voisins avant de traiter leurs voisins. Par rapport au parcours en profondeur, où les branches étaient explorées d'abord en direction des successeurs, il s'agit cette fois de placer les voisins d'un sommet en priorité par rapport aux voisins de ces voisins.
Pour cela, on reprend le concept de la liste de marquage et on fonctionne avec une file FIFO :

- on dépile le sommet en tête de file,
- s'il est déjà marqué, on passe au suivant
- sinon, on empile tous ses voisins, en queue de file (structure FIFO).

On recommence tant que la pile n'est pas vide.
Plus précisément on code une fonction `largeur` prenant en paramètre la liste `L_adj` d'adjacence d'un graphe, de la façon suivante :

- **Initialisation :** on crée :
  - une liste de marquage `visite` : une liste de booléens tous initialisés à `False`
  - une file d'éléments restants `file` : une liste initialement vide
- **Corps du programme :** pour chaque sommet `s`, si `s` n'est pas encore visité :
  - on empile `s`
  - tant que la pile n'est pas vide :
    - on dépile un sommet `t`,
    - si `t` n'est pas marqué, on le marque et on empile tous ses voisins.

Comme l'empilage a lieu en queue de file et le dépilage en tête de file, on se retrouve bien à parcourir les autres voisins d'un sommet donné, avant de se lancer dans les voisins des voisins. Le parcours est donc bien effectué « en largeur ».

- Importer la bibliothèque `collections`.
- Écrire une fonction `largeur` d'exploration en largeur à partir d'un sommet `s` d'un graphe $\mathcal{G}$.

```
def largeur(L_adj, s) :
    n = len(L_adj)
    visite = [False for k in range(n)]
    file = collections.deque()
    file.append(s)
    while len(file) != 0 :
        x = file.popleft()
        if visite[x] == False :
            visite[x] = True
            for y in L_adj[x] :
                file.append(y)
```

- Écrire enfin une fonction `parcours_largeur` qui effectue le parcours en largeur complet d'un graphe $\mathcal{G}$.

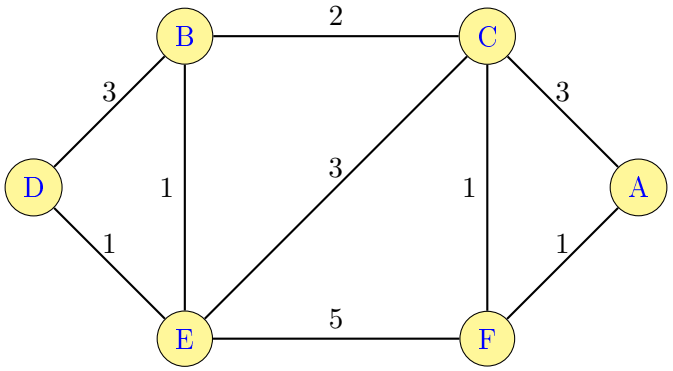
```
def largeur_rec(L_adj, visite, s) :
    if visite[x] == False :
        visite[x] = True
        for y in L_adj[x] :
            largeur_rec(L_adj, visite, y)

def parcours_largeur(L_adj) :
    n = len(L_adj)
    visite = [False for k in range(n)]
    for s in range(n) :
        largeur(L_adj, visite, s)
```

VI. Recherche des plus courts chemins dans un graphe : Algorithme de Dijkstra

VI.1. Objectif

L'algorithme de Dijkstra est un algorithme permettant de déterminer les plus courts chemins entre certains points d'un graphe. Avant de détailler le fonctionnement de cet algorithme, commençons par introduire le vocabulaire nécessaire sur les graphes.

Exemple
On considère le graphe suivant, donné par sa représentation graphique.



On rappelle que le graphe $\mathcal{G} = (S, A)$ est ici défini par :
× l'ensemble des sommets $S = \{A, B, C, D, E, F\}$,
× l'ensemble des arêtes $\mathcal{A} = \{(A, B), \ldots, (E, F), \ldots, (F, A)\}$.

Remarque
Le graphe présenté ici possède les caractéristiques suivantes.
- Il n'est pas **orienté** : si $(u, v) \in \mathcal{A}$ alors on a $(v, u) \in \mathcal{A}$.
  Ainsi, si on peut aller de u à v, alors on peut aussi aller de v à u.
- Il est **simple** : il y a au plus une arête entre deux sommets.
- Il est **pondéré** : à chaque arête on associe une valeur positive appelée poids.
  Ce poids représente la distance entre les deux sommets.

Définition
On appelle **matrice des poids** d'un graphe la matrice $G = (g_{i,j})_{(i,j)\in S^2}$ telle que pour tout couple de sommets (i, j) :
× si $(i, j) \in \mathcal{A}$ alors $g_{i,j}$ est égal au poids de l'arête (i, j),
× si $(i, j) \notin \mathcal{A}$ alors $g_{i,j} = +\infty$.

Exemple
Le graphe précédent possède 6 sommets.
Après renommage des sommets, la matrice des poids de ce graphe est :

$$\begin{array}{c} \\ \mathrm{D} \\ \mathrm{B} \\ \mathrm{E} \\ \mathrm{C} \\ \mathrm{F} \\ \mathrm{A} \end{array} \begin{array}{c} \begin{array}{cccccc} \mathrm{D} & \mathrm{B} & \mathrm{E} & \mathrm{C} & \mathrm{F} & \mathrm{A} \end{array} \\ \begin{pmatrix} 0 & 3 & 1 & +\infty & +\infty & +\infty \\ 3 & 0 & 1 & 2 & +\infty & +\infty \\ 1 & 1 & 0 & 3 & 5 & +\infty \\ +\infty & 2 & 3 & 0 & 1 & 3 \\ +\infty & +\infty & 5 & 1 & 0 & 1 \\ +\infty & +\infty & +\infty & 3 & 1 & 0 \end{pmatrix} \end{array}$$

On peut représenter cette matrice en **Python** sous la forme d'une liste de listes.

```
import numpy as np

Inf = np.inf
G = [ [0,3,1,Inf,Inf,Inf], [3,0,1,2,Inf,Inf], [1,1,0,3,5,Inf], \
[Inf,2,3,0,1,3], [Inf,Inf,5,1,0,1], [Inf,Inf,Inf,3,1,0] ]
```

- Quelle remarque peut-on faire sur la matrice de poids de ce graphe ?

C'est une matrice symétrique.

- De quelle propriété provient cette caractéristique ?

Le graphe précédent n'est pas orienté.
Ainsi chaque arête (u, v) fournit une arête (v, u) de même poids.

VI.2. Brève présentation de l'algorithme

L'algorithme de Dijkstra consiste en la recherche des plus courts chemins menant d'un sommet unique $s \in S$ à chaque autre sommet d'un graphe pondéré $\mathcal{G} = (S, \mathcal{A})$.
Tous les arcs de $\mathcal{G}$ sont supposés de poids positif ce qui permet d'empêcher la présence de cycles de poids strictement négatifs.

Notation
Pour tout $t \in S$, on nomme $\delta(t)$ la longueur du plus court chemin menant de s à t.

VI.3. Calcul de la longueur des plus courts chemins

Avant de déterminer les plus courts chemins entre s et tout autre sommet t, on commence par déterminer la longueur de chacun de ces plus courts chemins *i.e.* la fonction δ.

Pour ce faire, on utilise un attribut $d[v]$ du sommet $v \in S$ qui contient le poids du supposé plus court chemin menant de s à v. Cet attribut est corrigé au fur et à mesure de l'algorithme de sorte à contenir $\delta(v)$ à la fin de l'exécution. Évidemment, cet attribut est, à chaque étape de l'algorithme, un majorant du poids d'un plus court chemin menant de s à v. Autrement dit :

$$\forall v \in S, \quad \delta(v) \leqslant d[v]$$

Détaillons la fonctionnement de cet attribut.

Mise à jour de l'attribut $d[v]$

1) **Initialisation**

Au début de l'algorithme, on considère :

× $d[s] \leftarrow 0$,

× $d[v] \leftarrow +\infty$ pour tout sommet $v \in S \setminus \{s\}$.

Exemple

Sur l'exemple précédent, et si on choisit $s = D$, l'étape d'initialisation s'écrit :

D	B	E	C	F	A
0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$

2) **Principe de relâchement**

L'opération de relâchement d'une arc $(u, v) \in S \times S$ consiste en un test permettant de savoir s'il est possible, en passant par u, d'améliorer le plus court chemin jusqu'à v.

Si oui, il faut mettre à jour $d[v]$.

Plus précisément, on effectue le test suivant :

$$d[u] + g_{u,v} \overset{?}{<} d[v]$$

- Si le test est négatif, on n'effectue pas de mise à jour.
- Si le test est positif, cela signifie qu'un chemin de plus petite taille est exhibé pour passer du sommet s au sommet v :
  × on passe de s à u (ce chemin à le poids $d[u]$),
  × on termine ce chemin en utilisant l'arc u, v de poids $g_{u,v}$.
  Il convient alors d'effectuer la mise à jour :

$$d[v] \leftarrow d[u] + g_{u,v}$$

Exemple

1) Sur l'exemple précédent, on prend $u = D$

On cherche alors à déterminer, pour tout sommet $v \in S$ si on peut mettre à jours $d[v]$ à l'aide d'un chemin dont le dernier arc est (u, v).

Étant donnée l'étape d'initialisation, cela consiste à considérer les chemins de s à v de taille 1 :

D	B	E	C	F	A
0	3	1	$+\infty$	$+\infty$	$+\infty$

2) On prend alors $u = E$

On cherche alors à déterminer, pour tout sommet $v \in S$ si on peut mettre à jour $d[v]$ à l'aide d'un chemin dont le dernier arc est (u, v).

Étant donnée l'étape précédente, cela consiste à considérer les chemins de s à v de taille 2 (autrement dit les chemins $s \to u \to v$) :

D	B	E	C	F	A
0	2	1	4	6	$+\infty$

3) On prend alors $u = B$

On cherche alors à déterminer, pour tout sommet $v \in S$ si on peut mettre à jour $d[v]$ à l'aide d'un chemin dont le dernier arc est (u, v).

Étant donnée l'étape précédente, cela consiste à considérer les chemins de s à v de taille 3 (autrement dit les chemins $s \to \ldots \to u \to v$) :

D	B	E	C	F	A
0	2	1	4	6	$+\infty$

4) On prend alors $u = C$

On cherche alors à déterminer, pour tout sommet $v \in S$ si on peut mettre à jour $d[v]$ à l'aide d'un chemin dont le dernier arc est (u, v).

Étant donnée l'étape précédente, cela consiste à considérer les chemins de s à v de taille 4 (autrement dit les chemins $s \to \ldots \to \ldots \to u \to v$) :

D	B	E	C	F	A
0	2	1	4	5	7

5) On prend alors $u = F$

On cherche alors à déterminer, pour tout sommet $v \in S$ si on peut mettre à jour $d[v]$ à l'aide d'un chemin dont le dernier arc est (u, v).

Étant donnée l'étape précédente, cela consiste à considérer les chemins de s à v de taille 5 (autrement dit les chemins $s \to \ldots \to \ldots \to \ldots \to u \to v$) :

D	B	E	C	F	A
0	2	1	4	5	6

6) On prend alors $u = A$

On cherche alors à déterminer, pour tout sommet $v \in S$ si on peut mettre à jour $d[v]$ à l'aide d'un chemin dont le dernier arc est (u, v).

Étant donnée l'étape précédente, cela consiste à considérer les chemins de s à v de taille 6 (autrement dit les chemins $s \to \ldots \to \ldots \to \ldots \to \ldots \to u \to v$) :

D	B	E	C	F	A
0	2	1	4	5	6

Sélection du sommet u à chaque étape

À chaque étape de l'algorithme, on sélectionne le sommet u qui vérifie les propriétés suivantes :

- u n'a pas encore été sélectionné,
- l'attribut $d[u]$ est le plus faible (parmi tous les sommets non encore sélectionnés).

Cet algorithme fait partie des algorithmes **glouton** : on sélectionne à chaque étape la sous-solution optimale *i.e.* le sommet réalisant la meilleure distance.

Fin de l'algorithme

Une fois tous les sommets visités, on a trouvé tous les plus courts chemins (de s à tout v) de taille inférieure ou égale au nombre de sommets en tout. On a donc trouvé tous les plus courts de chemins (de s à tout v).

VI.4. Implémentation

► Appliquer l'algorithme de Dijkstra avec origine D.

D	B	E	C	F	A
0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
0	3	1	$+\infty$	$+\infty$	$+\infty$
0	2	1	4	6	$+\infty$
0	2	1	4	6	$+\infty$
0	2	1	4	5	7
0	2	1	4	5	6
0	2	1	4	5	6

► Appliquer l'algorithme de Dijkstra avec origine E.

D	B	E	C	F	A

► Appliquer l'algorithme de Dijkstra avec origine C.

D	B	E	C	F	A

▶ Implémenter `DijkstraDist(G, depart)` qui prend en paramètre la matrice des poids `G` et le sommet `depart` et renvoie la taille des plus courts chemins de `depart` à tout sommet v.

```python
def DijkstraDist(G, depart) :
    # On récupère le nombre de sommets du graphe
    n = len(G)

    # Initialisation du tableau des plus courts chemins
    # Le booléen pour savoir si le sommet a déjà été sélectionné
    pcc = [Inf for i in range(n)]
    visite = [False for i in range(n)]
    sommet_u = depart
    dist_u = 0
    pcc[depart] = 0
    # Le premier sommet sélectionné est le sommet depart
    visite[depart] = True

    # On compte le nombre de sommets sélectionnés
    cpt = 0
    while cpt != n-1 :
        # À chaque étape, la solution optimale doit être conservée
        # (pour sélection du sommet correspondant à l'étape suivante)
        minimum = Inf
        # Étape de relâchement
        for k in range(n) :

            # Si le sommet k n'a pas encore été sélectionné
            if visite[k] == False :
                dist_uv = G[sommet_u][k]
                # Distance totale du chemin s -> ... -> u -> v
                dist_totale = dist_u + dist_uv

                # Mise à jour du tableau des plus courts chemins
                if dist_totale < pcc[k] :
                    pcc[k] = dist_totale

                # Mise à jour de la solution minimale à cette étape
                if pcc[k] < minimum :
                    minimum = pcc[k]
                    prochain_sommet_select = k

        # On a traité complétement un sommet
        cpt = cpt + 1

        # Le sommet à traiter est sélectionné et d[u] est mis à jour
        sommet_u = prochain_sommet_select
        visite[sommet_u] = True
        dist_u = pcc[sommet_u]

    return pcc, visite
```

VI.5. Exhiber les plus courts chemins

▶ En reprenant les exemples précédents (avec origine D puis avec origine E), expliquer comment on peut obtenir les plus courts chemins à l'aide des calculs de taille précédents.

- Si à une étape, $d[v]$ a été modifié, il faut se rappeler de quel sommet u provient cette modification.
- L'idée étant alors que l'arc (u, v) sera un arc du plus court chemin.

▶ Modifier le premier tableau (avec origine en D) pour qu'il prenne en compte cette nouvelle information.

D	B	E	C	F	A
0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
0	3	1^{D}	$+\infty$	$+\infty$	$+\infty$
0	2^{E}	1	4^{E}	6^{E}	$+\infty$
0	2	1	4	6	$+\infty$
0	2	1	4	5^{C}	7
0	2	1	4	5	6^{F}
0	2	1	4	5	6

▶ Reconstituer alors le plus court chemin de D vers A, celui de D vers E et celui de D vers B.

- Le plus court chemin de D vers A est réalisé par $D \to E \to C \to F \to A$.
- Le plus court chemin de D vers E est réalisé par $D \to E$.
- Le plus court chemin de D vers B est réalisé par $D \to E \to B$.

▶ Implémenter la fonction `DijsktraDistChemin(G, depart)` qui prend en paramètre la matrice des poids `G` et le sommet origine `depart` et renvoie la taille des plus courts chemins de `depart` à tout sommet v ainsi que le dernier sommet utilisé pour calculer cette taille.

On écrira seulement les trois lignes qui diffèrent de la fonction précédente.

- On introduit une ligne entre les lignes 8 et 9 : on doit se souvenir d'une information supplémentaire.

```
predecesseur = [None for i in range(n)]
```

- On doit ajouter une ligne 33 pour se souvenir de quel sommet provient la modification.

```
predecesseur[k] = sommet_u
```

- On doit ajouter la liste `predecesseur` à la liste des paramètres de sortie.

```
return pcc, visite, predecesseur
```

▶ Implémenter la fonction `dijkstraPCC(G, depart, arrivee)` qui permet d'obtenir le plus court chemin de `depart` à `arrivee`.

```
def dijkstraPCC(G, depart, arrivee) :
    predecesseur = dijkstraDistChemin(G, depart)
    predecesseur = predecesseur[2]
    # Reconstitution du plus court chemin
    chemin = []
    # On reconstitue le plus court chemin d'arrivee vers depart
    ville = arrivee
    chemin.append(ville)
    while ville != depart :
        ville = predecesseur[ville]
        chemin.append(ville)
    # On demande le miroir de la liste obtenue pour
    # que les sommets apparaissent dans l'ordre
    return list(reversed(chemin))
```

VI.6. Correction de l'algorithme

On note S l'ensemble des sommets, et $\mathcal{A}$ l'ensemble des arêtes du graphe $\mathcal{G}$ étudié. On note n l'ordre du graphe $\mathcal{G}$. Ainsi : $n = \text{Card}(S)$.

Terminaison

- La variable `n - cpt` définit une suite strictement décroissante d'entiers positifs. En effet, la variable `cpt`est incrémenté de 1 à chaque tour de boucle `while`.
  Cette variable est donc un variant de la boucle `while`.
- La variable `n - k` définit une suite strictement décroissante d'entiers positifs. En effet, la variable `k` est incrémentée de 1 à chaque tour de boucle `for`.
  Cette variable est donc un variant de la boucle `for`.

Ainsi, l'algorithme se termine.

Correction partielle

On note, pour tout $k \in [\![1, n]\!]$, A_k le contenu de la liste `visite` et s_k la valeur de `sommet_u`, à la $k^{\text{ème}}$ itération de la boucle principale (boucle `while`).

Soit $k \in [\![1, n]\!]$. La proposition suivante est un invariant de boucle.

$\mathcal{P}(k)$: « après l'itération k de la boucle principale, pour tout $v \in S$, `pcc[v]` contient soit $+\infty$, soit le plus petit poids total parmi les chemins menant de s à v en utilisant les éléments de A_k contenant la valeur `True` »

Dit plus simplement, on va prouver qu'à chaque étape k, l'algorithme « fait de son mieux » avec les sommets déjà utilisés (ceux de valeur `True` dans A_k, c'est-à-dire les sommets déjà visités).

Démontrons par récurrence : $\forall k \in [\![1, n]\!]$, $\mathscr{P}(k)$.

- **Initialisation** :
  On sait : $_1 = s =$ `depart`. En effet, s est le seul élément de la liste `pcc` présentant un poids non infini.
  Puis, pour chaque élément $v \in S$, l'élément `pcc[v]` se trouve laissé à $+\infty$ si aucune arête ne relie s à v, et par le poids de l'arête $s - v$ si celle-ci existe.
  Finalement :
  - $A_1[s] =$ `True`
  - pour tout $v \neq s$, $A_1[v] =$ `False`.

  Et on a bien déterminé les plus courts chemins issus de s, passant par $s_1 = s$, vers chaque sommet de S.
- **Hérédité** : soit $k \in [\![1, n-1]\!]$.
  Supposons $\mathscr{P}(k)$ et démontrons $\mathscr{P}(k+1)$.
  Par hypothèse de récurrence, le sommet s_{k+1} est le sommet le plus proche de s en passant par les sommets déjà visités (les sommets v tels que : $A_k[v] =$ `True`). Alors :
  - pour chaque sommet $v \in S$ non visité, on regarde si passer par s_{k+1} est plus court : la distance obtenue est donc bien $+\infty$ si le chemin est inexistant, ou sinon, égale à ce qui se fait de plus court en passant par les points déjà visités et s_{k++1}.
  - pour chaque sommet $v \in S$ déjà visité, passer par s_{k+1} ne peut qu'augmenter (au sens large) le poids du chemin.

  D'où $\mathscr{P}(k+1)$.

L'algorithme est partiellement correct et se termine. Il est donc correct.

VI.7. Complexité

Complexité
Soit $\mathcal{G} = (S, \mathcal{A})$ un graphe. On note toujours $n = \text{Card}(S)$. On note de plus : $p = \text{Card}(\mathcal{A})$.
On considère l'affectation et la comparaison comme opérations élémentaires.

- **Initialisation** : on effectue :
  - 1 affectation en ligne 3,
  - 2 affectations de n étapes chacunes en lignes 7 et 8,
  - 5 affectations entre les lignes 9 à 16.
- **Structure itérative** :
  On effectue n tours de boucle `while`. Pour chaque tour de boucle `while`, on effectue :
  - 1 comparaison (en ligne 17),
  - 1 affectation en ligne 20,
  - n tours de boucles `for`. Pour chaque tour de boucle `for`, on effectue dans le pire cas :
    - 3 comparaisons (en lignes 25, 31 et 35),
    - 5 affectations.
  - 4 affectations entre les lignes 40 et 45

Finalement, on effectue le nombre d'opérations élémentaires suivant :

$$1 + 2 + 5 + n \times (1 + 1 + n \times (3 + 5) + 4) \underset{n \to +\infty}{=} O(n^2)$$

L'algorithme, dans la version décrite ci-dessus est donc en $\underset{n \to +\infty}{O}(n^2)$.