Semestre 2 / Cours 3 - Notion de graphe

ITC MPSI & PCSI – Année 2024-2025

LYCÉE MONTAIGNE

Sommaire

1. Introduction

2. Graphes : définitions et premiers exemples

3. Implémentation en Python

4. Un peu de coloriage

Introduction

Objectifs

- Découvrir la notion de graphe et le vocabulaire associé.

Objectifs

- Découvrir la notion de graphe et le vocabulaire associé.
- Savoir implémenter un graphe en Python sous la forme de matrice d'adjacence et de liste d'adjacence.

Objectifs

- Découvrir la notion de graphe et le vocabulaire associé.
- Savoir implémenter un graphe en Python sous la forme de matrice d'adjacence et de liste d'adjacence.
- Utilisation pour l'étude de réseaux informatiques ou de réseaux routiers.

Graphes : définitions et premiers exemples

Graphes orientés et non orientés

Graphes orientés et non orientés

Un *graphe* G est la donnée d'un couple (S, A) où S et A sont deux ensembles finis :

- $S = \{v_1, \ldots, v_n\}$ est l'ensemble des *sommets* (ou *noeuds*) du graphe.

Graphes orientés et non orientés

Un *graphe* G est la donnée d'un couple (S, A) où S et A sont deux ensembles finis :

- $S = \{v_1, \ldots, v_n\}$ est l'ensemble des *sommets* (ou *noeuds*) du graphe.
- $A = \{e_1, \ldots, e_m\}$ est l'ensemble de ses *arêtes* (ou *arcs*). Chaque arête e_i est définie par un *couple* de sommets de S appelés *extrémités* de e_i.

Graphes orientés et non orientés

Un *graphe* G est la donnée d'un couple (S, A) où S et A sont deux ensembles finis :

- $S = \{v_1, \ldots, v_n\}$ est l'ensemble des *sommets* (ou *noeuds*) du graphe.
- $A = \{e_1, \ldots, e_m\}$ est l'ensemble de ses *arêtes* (ou *arcs*). Chaque arête e_i est définie par un *couple* de sommets de S appelés *extrémités* de e_i.

Deux sommets reliés par une arête sont dits *adjacents*.
On appelle *ordre* d'un graphe son nombre de sommets, c'est-à-dire $\mathrm{Card}\, S$.

Graphes orientés et non orientés

Graphes orientés et non orientés

Les arêtes d'un graphe peuvent être orientées : le graphe est alors dit *orienté*.
Un graphe pour lequel les arêtes ne sont pas orientée est dit *non orienté*.

Graphes orientés et non orientés

Les arêtes d'un graphe peuvent être orientées : le graphe est alors dit *orienté*.
Un graphe pour lequel les arêtes ne sont pas orientée est dit *non orienté*.

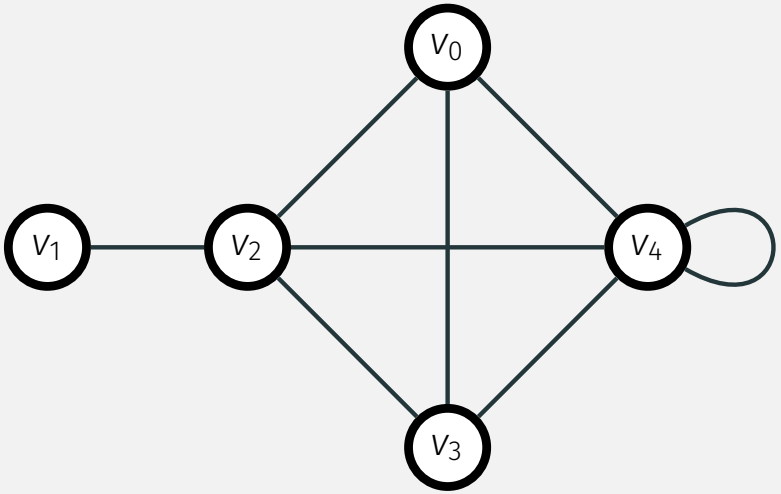
Les graphes peuvent être simplement *représentés par un dessin*.
Chaque *sommet* est représenté par un *cercle*.
Chaque *arête* est représentée par une *ligne courbe* reliant deux sommets adjacents.

Graphes orientés et non orientés

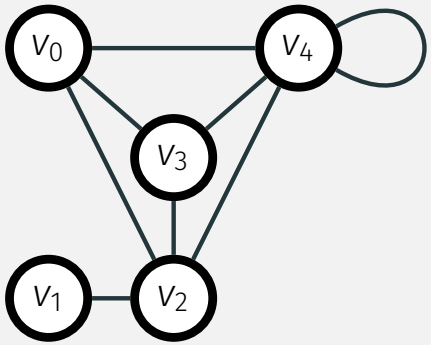
Considérons le *graphe non orienté*

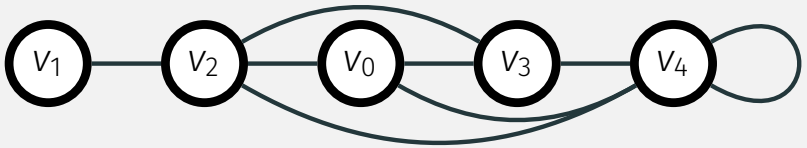
$$G_1 = (S_1, A_1), \quad S_1 = \{v_0, v_1, v_2, v_3, v_4\}$$

$$\begin{aligned} A_1 = \{&(v_0, v_2), (v_0, v_3), (v_0, v_4), \\ &(v_1, v_2), \\ &(v_2, v_0), (v_2, v_1), (v_2, v_3), (v_2, v_4), \\ &(v_3, v_0), (v_3, v_2), (v_3, v_4), \\ &(v_4, v_0), (v_4, v_2), (v_4, v_3), (v_4, v_4)\} \end{aligned}$$



Deux représentations topologiquement équivalentes de G_1 :



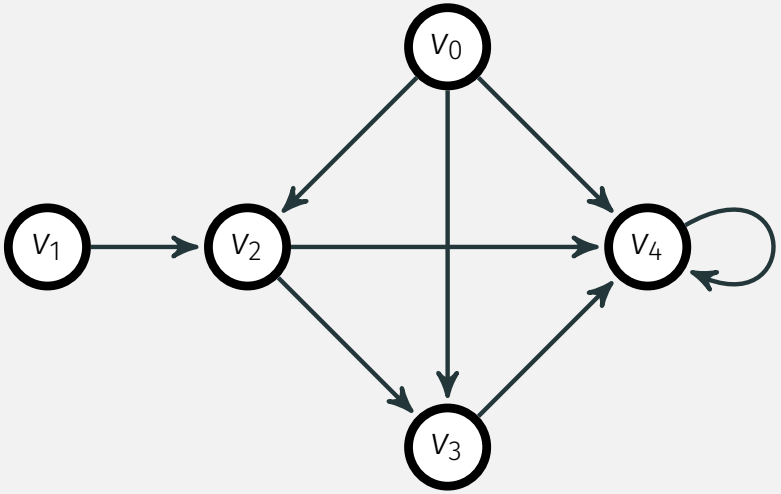


Considérons le *graphe orienté*

$$G_2 = (S_2, A_2)$$

$$S_2 = \{v_0, v_1, v_2, v_3, v_4\}$$

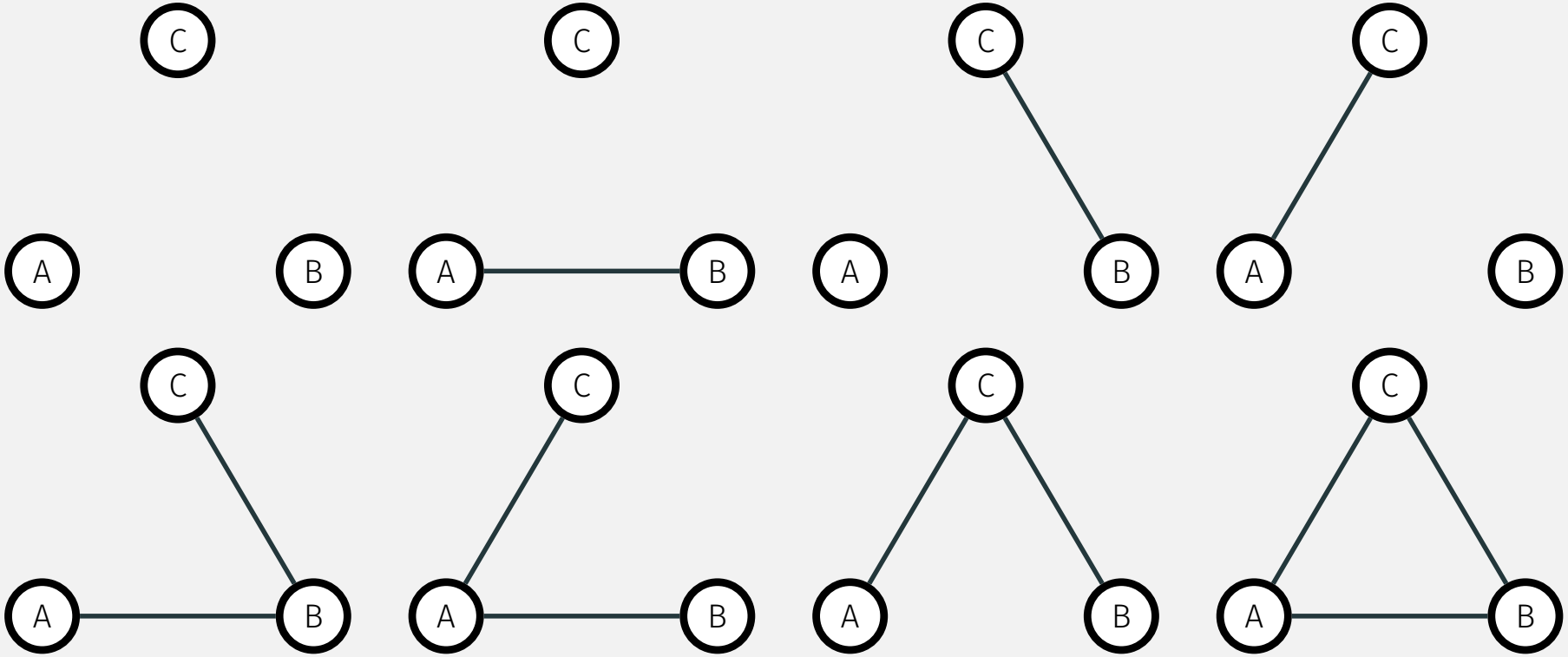
$$A_2 = \{(v_0, v_2), (v_0, v_3), (v_0, v_4), (v_1, v_2), (v_2, v_3), (v_2, v_4), (v_3, v_4), (v_4, v_4)\}$$



Exercice :

1. Dessiner tous les graphes non orientés sans boucle ayant exactement trois sommets.
2. Combien y a-t-il de graphes orientés sans boucle ayant trois sommets ?

Graphes non orientés sans boucle ayant trois sommets.
Il y en a exactement 8 :

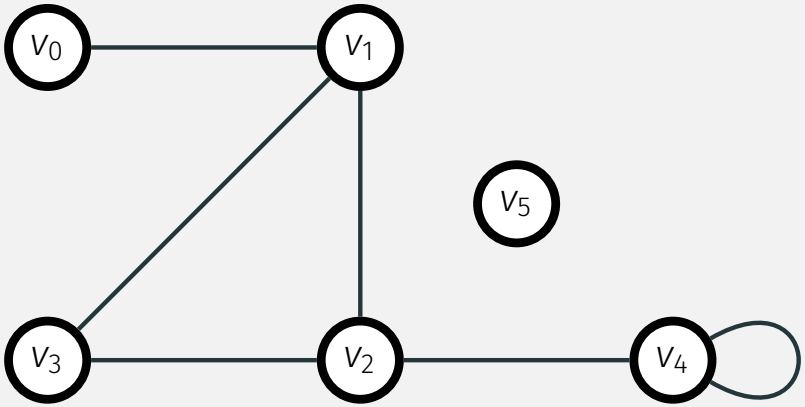


Graphes orientés sans boucle ayant trois sommets :

Chaque couple (u, v) de sommets peut ne pas être relié par un arc, être relié par un arc de u vers v, par un arc de v vers u ou par deux arcs (un de u vers v et un de v vers u), soit 4 possibilités pour chaque couple. Or, il y a 3 couples de sommets, donc au total $4^3 = 64$ graphes orientés différents.

Degré d'un sommet

Dans un graphe non orienté, on appelle *degré* d'un sommet s le nombre d'arêtes noté $\mathrm{d}(s)$ dont ce sommet est une extrémité. Ce degré vaut 0 si le sommet est isolé, et une boucle compte double.



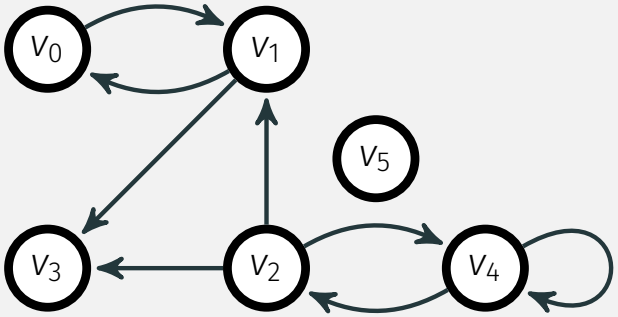
s	v_0	v_1	v_2	v_3	v_4	v_5
$\mathrm{d}(s)$	1	3	3	2	3	0

Degré d'un sommet

Dans un graphe orienté, on appelle :

- *degré sortant* d'un sommet s, noté $\mathrm{d}_+(s)$, le nombre d'arcs ayant s pour extrémité initiale ;
- *degré entrant* d'un sommet s, noté $\mathrm{d}_-(s)$, le nombre d'arcs ayant s pour extrémité finale ;
- *degré* d'un sommet s comme la somme $\mathrm{d}(s)$ définie par :

$$\mathrm{d}(s) = \mathrm{d}_+(s) + \mathrm{d}_-(s).$$



s	v_0	v_1	v_2	v_3	v_4	v_5
$\mathrm{d}_+(s)$	1	2	3	0	2	0
$\mathrm{d}_-(s)$	1	2	1	2	2	0
$\mathrm{d}(s)$	2	4	4	2	4	0

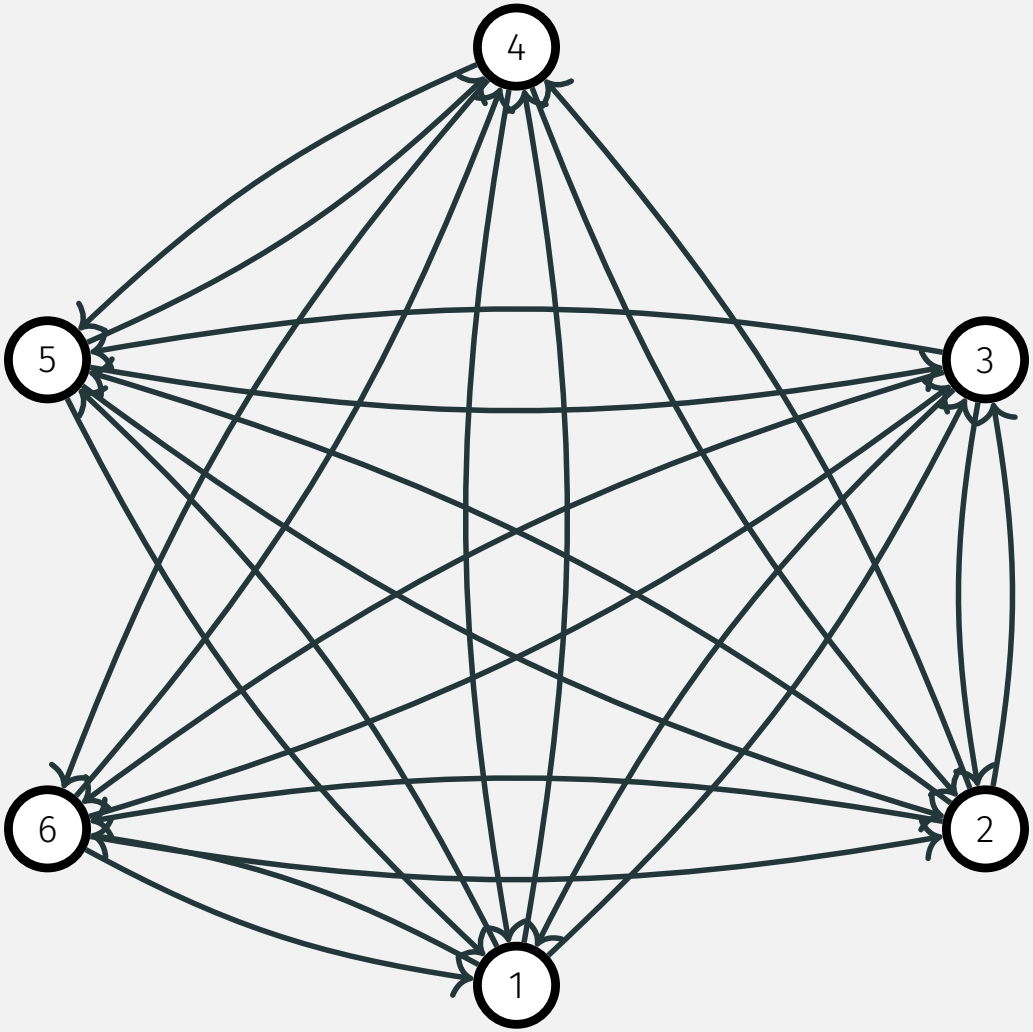
Degré d'un sommet

Exercice : Trois pays envoient chacun à une conférence deux espions ; chaque espion doit espionner tous les espions des autres pays (mais pas son propre collègue !).

1. Représenter cette situation par un graphe orienté d'ordre 6 dans lequel la présence d'un arc du sommet i vers le sommet j signifie que i espionne j.
2. Calculer le nombre d'arcs de ce graphe ainsi que le degré de chaque sommet.

Degré d'un sommet

Notons 1 et 2 les espions du premier pays, 3 et 4 les espions du deuxième pays et 5 et 6 les espions du troisième pays. On obtient le graphe suivant :



Degré d'un sommet

Nombre d'arcs de ce graphe et degré de chaque sommet :

Chaque espion espionne 4 autres espions, donc, chaque sommet est le point de départ de 4 arcs, et de même, est le point d'arrivée de 4 arcs. Il y a donc au total 24 arcs. Chaque sommet a un degré sortant égal à 4 et un degré entrant égal à 4, donc, un degré total égal à 8.

Chemins et cycles

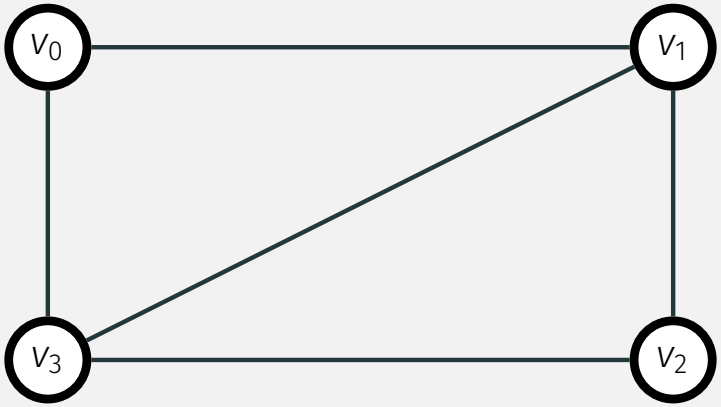
- Dans un graphe *orienté ou non*, on appelle *chemin* reliant un sommet u à un sommet v toute suite *finie* de sommets reliés deux à deux par des arêtes et menant de u à v. La *longueur* du chemin est le nombre d'arêtes (ou d'arcs) dans le chemin.

Chemins et cycles

- Dans un graphe *orienté ou non*, on appelle *chemin* reliant un sommet u à un sommet v toute suite *finie* de sommets reliés deux à deux par des arêtes et menant de u à v. La *longueur* du chemin est le nombre d'arêtes (ou d'arcs) dans le chemin.
- On dit qu'un chemin est *simple* s'il n'emprunte pas deux fois la même arête (ou le même arc).

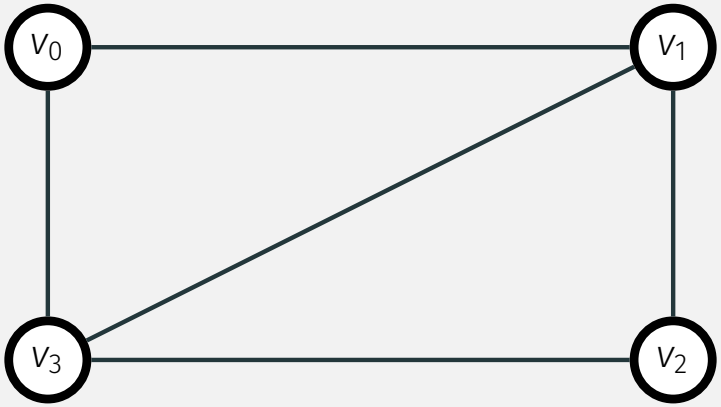
Chemins et cycles

- Dans un graphe *orienté ou non*, on appelle *chemin* reliant un sommet u à un sommet v toute suite *finie* de sommets reliés deux à deux par des arêtes et menant de u à v. La *longueur* du chemin est le nombre d'arêtes (ou d'arcs) dans le chemin.
- On dit qu'un chemin est *simple* s'il n'emprunte pas deux fois la même arête (ou le même arc).
- Un chemin simple reliant un sommet à lui-même et contenant au moins une arête (ou arc) est appelé un *cycle*.

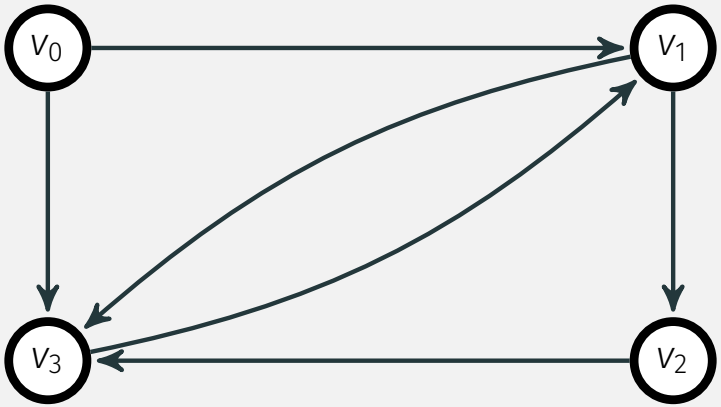


- Dans le graphe non orienté ci-dessus, (v_0, v_1, v_2, v_3), (v_0, v_1, v_3) et (v_0, v_3) sont des chemins simples du sommet v_0 vers le sommet v_3 de longueurs respectives 3, 2 et 1.

Chemins et cycles

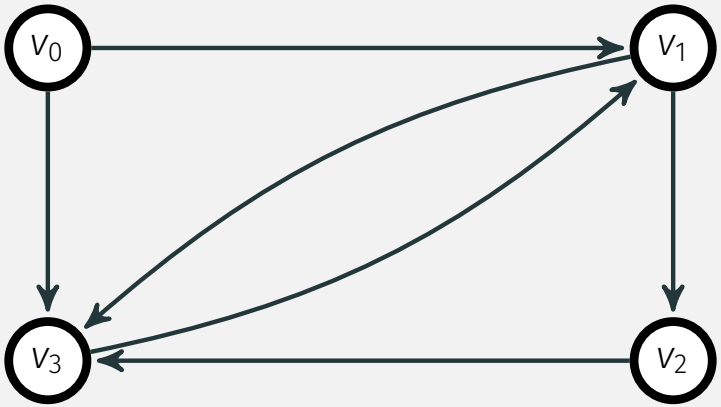


- Dans le graphe non orienté ci-dessus, (v_0, v_1, v_2, v_3), (v_0, v_1, v_3) et (v_0, v_3) sont des chemins simples du sommet v_0 vers le sommet v_3 de longueurs respectives 3, 2 et 1.
- En revanche, (v_0, v_1, v_0) n'est pas un chemin simple ; ce n'est pas un cycle.



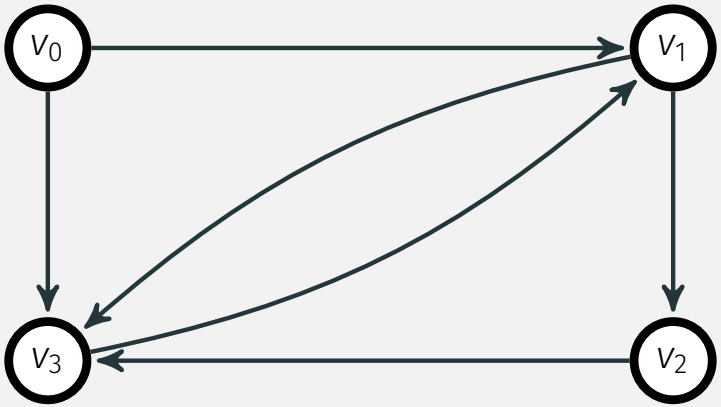
- Dans le graphe orienté ci-dessus, (v_0, v_1, v_2) et (v_0, v_3, v_1, v_2) sont des chemins du sommet v_0 vers le sommet v_2 de longueurs respectives 2 et 3.

Chemins et cycles



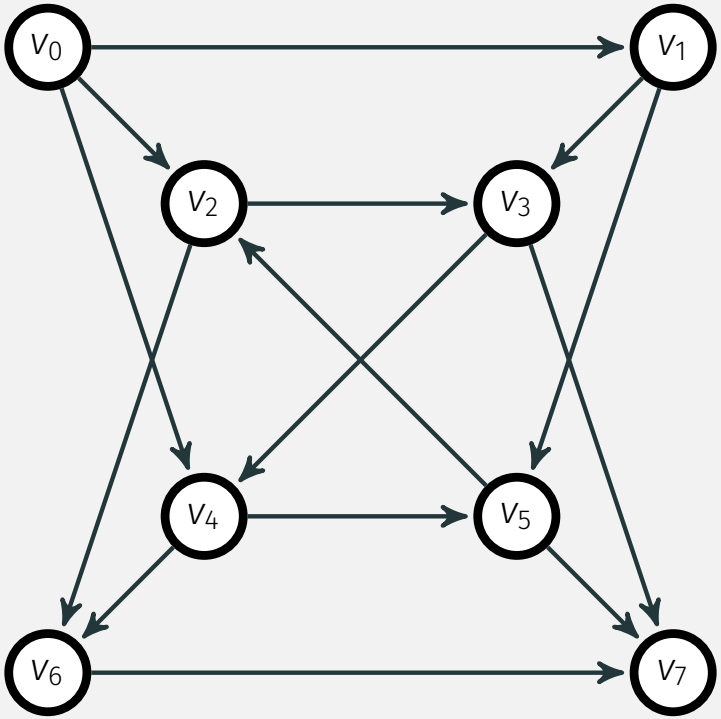
- Dans le graphe orienté ci-dessus, (v_0, v_1, v_2) et (v_0, v_3, v_1, v_2) sont des chemins du sommet v_0 vers le sommet v_2 de longueurs respectives 2 et 3.
- En revanche, il n'existe pas de chemin de v_2 vers v_0.

Chemins et cycles



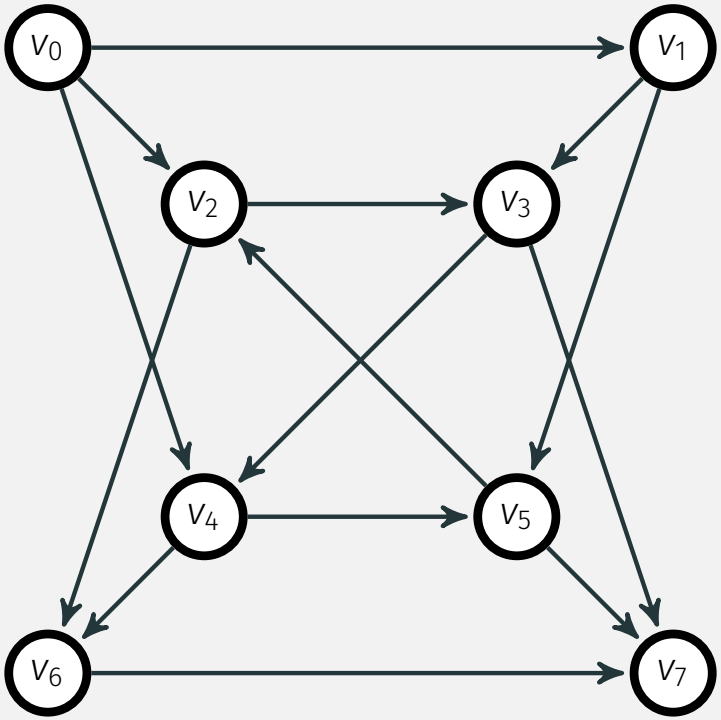
- Dans le graphe orienté ci-dessus, (v_0, v_1, v_2) et (v_0, v_3, v_1, v_2) sont des chemins du sommet v_0 vers le sommet v_2 de longueurs respectives 2 et 3.
- En revanche, il n'existe pas de chemin de v_2 vers v_0.
- Le chemin (v_1, v_2, v_3, v_1) est un cycle.

Exercice : On considère le graphe orienté ci-dessous :



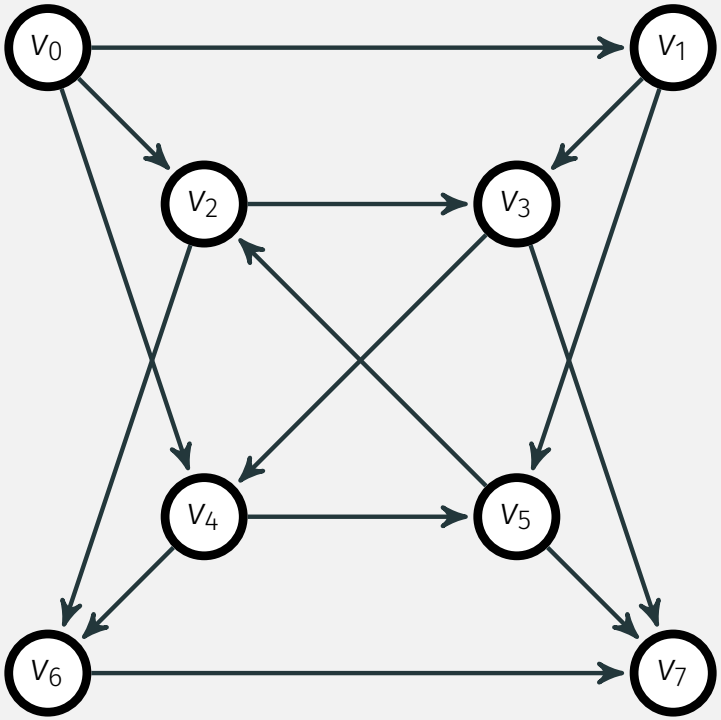
1. Pour tout entier $k \in [\![1, 5]\!]$, peut-on trouver un chemin de longueur k reliant v_0 à v_7 ?
2. Peut-on trouver un chemin reliant v_6 à v_1 ?
3. Déterminer un cycle.

Exercice :



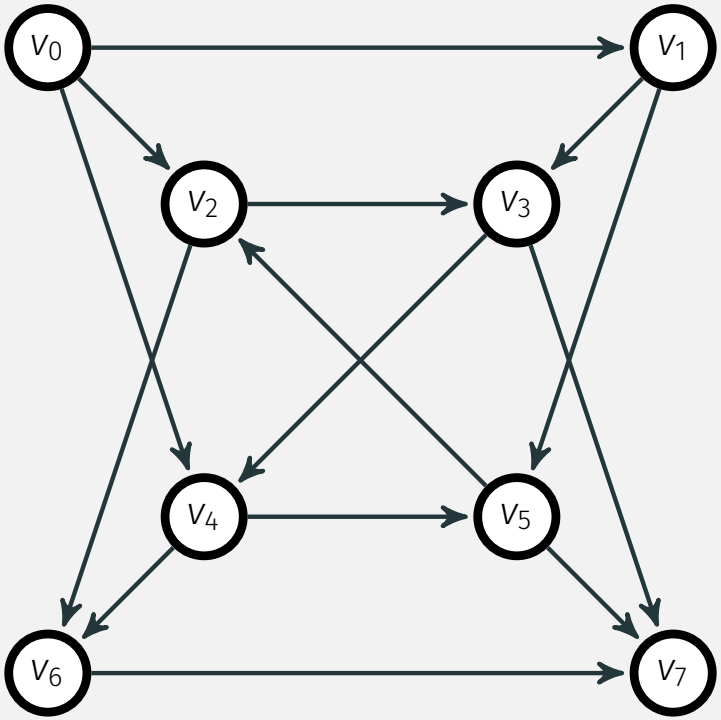
- Pas de chemin de longueur 1, 2 ou 4 reliant v_0 à v_7.

Exercice :



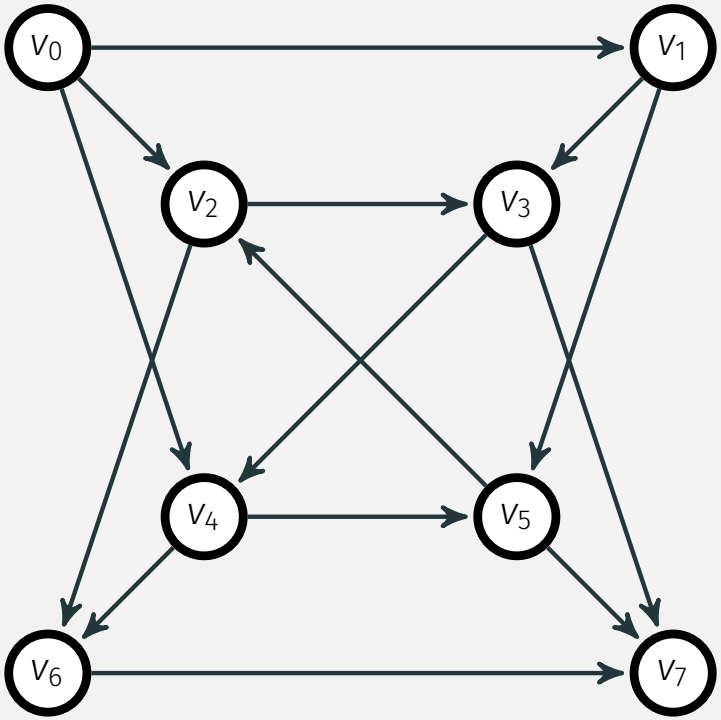
- Pas de chemin de longueur 1, 2 ou 4 reliant v_0 à v_7.
- Le chemin (v_0, v_1, v_3, v_7) est de longueur 3.

Exercice :



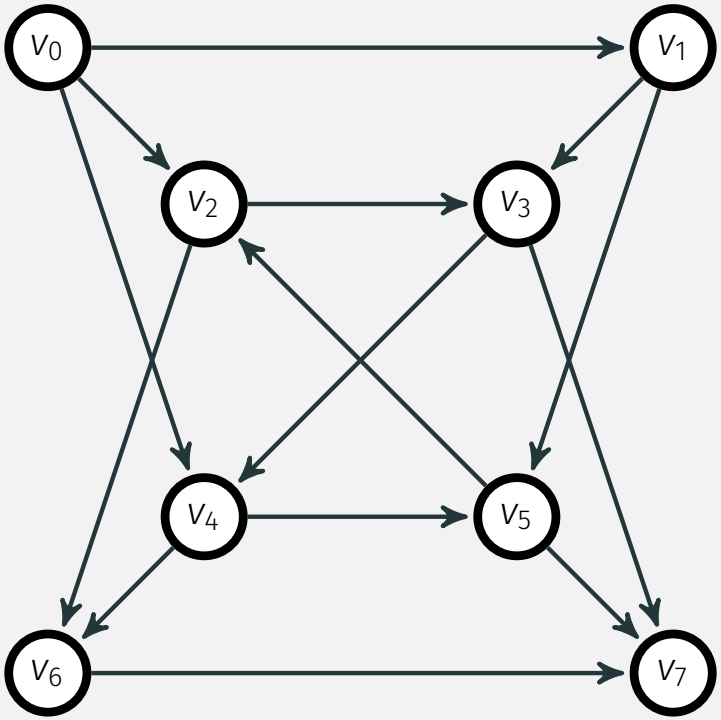
- Pas de chemin de longueur 1, 2 ou 4 reliant v_0 à v_7.
- Le chemin (v_0, v_1, v_3, v_7) est de longueur 3.
- Le chemin $(v_0, v_2, v_3, v_4, v_5, v_7)$ est de longueur 5 reliant v_0 à v_7.

Exercice :



- Pas de chemin de longueur 1, 2 ou 4 reliant v_0 à v_7.
- Le chemin (v_0, v_1, v_3, v_7) est de longueur 3.
- Le chemin $(v_0, v_2, v_3, v_4, v_5, v_7)$ est de longueur 5 reliant v_0 à v_7.
- Pas de chemin reliant v_6 à v_1.

Exercice :



- Pas de chemin de longueur 1, 2 ou 4 reliant v_0 à v_7.
- Le chemin (v_0, v_1, v_3, v_7) est de longueur 3.
- Le chemin $(v_0, v_2, v_3, v_4, v_5, v_7)$ est de longueur 5 reliant v_0 à v_7.
- Pas de chemin reliant v_6 à v_1.
- $(v_2, v_3, v_4, v_5, v_2)$ est un cycle.

Longueur d'un chemin, distance entre deux sommets

- La longueur d'un chemin γ est le nombre d'arêtes/d'arcs qui le composent. On la note $\delta(\gamma)$

Longueur d'un chemin, distance entre deux sommets

- La longueur d'un chemin γ est le nombre d'arêtes/d'arcs qui le composent. On la note $\delta(\gamma)$
- Par convention, il existe un chemin de longueur nulle d'un sommet s à lui-même. *(il ne faut pas confondre avec les éventuels chemins* $(s, s, \ldots, s)$*, si le graphe contient la boucle* (s, s)*)*

Longueur d'un chemin, distance entre deux sommets

- La longueur d'un chemin γ est le nombre d'arêtes/d'arcs qui le composent. On la note $\delta(\gamma)$
- Par convention, il existe un chemin de longueur nulle d'un sommet s à lui-même. *(il ne faut pas confondre avec les éventuels chemins* $(s, s, \ldots, s)$*, si le graphe contient la boucle* (s, s)*)*
- La distance $d_u[v]$ d'un sommet u à un sommet v est le minimum des longueurs des chemins menant de u à v.

Longueur d'un chemin, distance entre deux sommets

- La longueur d'un chemin γ est le nombre d'arêtes/d'arcs qui le composent. On la note $\delta(\gamma)$
- Par convention, il existe un chemin de longueur nulle d'un sommet s à lui-même. *(il ne faut pas confondre avec les éventuels chemins* $(s, s, \ldots, s)$*, si le graphe contient la boucle* (s, s)*)*
- La distance $d_u[v]$ d'un sommet u à un sommet v est le minimum des longueurs des chemins menant de u à v.
- En notant $\Gamma_u(v)$ l'ensemble des chemins de u à v on a donc :

$$d_u[v] = \min_{\gamma \in \Gamma_u(v)} \delta(\gamma)$$

Longueur d'un chemin, distance entre deux sommets

- La longueur d'un chemin γ est le nombre d'arêtes/d'arcs qui le composent. On la note $\delta(\gamma)$
- Par convention, il existe un chemin de longueur nulle d'un sommet s à lui-même. *(il ne faut pas confondre avec les éventuels chemins* $(s, s, \ldots, s)$*, si le graphe contient la boucle* (s, s)*)*
- La distance $d_u[v]$ d'un sommet u à un sommet v est le minimum des longueurs des chemins menant de u à v.
- En notant $\Gamma_u(v)$ l'ensemble des chemins de u à v on a donc :

$$d_u[v] = \min_{\gamma \in \Gamma_u(v)} \delta(\gamma)$$

- Une « vraie » distance ?

Longueur d'un chemin, distance entre deux sommets

- La longueur d'un chemin γ est le nombre d'arêtes/d'arcs qui le composent. On la note $\delta(\gamma)$
- Par convention, il existe un chemin de longueur nulle d'un sommet s à lui-même. *(il ne faut pas confondre avec les éventuels chemins* $(s, s, \ldots, s)$*, si le graphe contient la boucle* (s, s)*)*
- La distance $d_u[v]$ d'un sommet u à un sommet v est le minimum des longueurs des chemins menant de u à v.
- En notant $\Gamma_u(v)$ l'ensemble des chemins de u à v on a donc :

$$d_u[v] = \min_{\gamma \in \Gamma_u(v)} \delta(\gamma)$$

- Une « vraie » distance ?
  - ◇ La propriété de séparation est vérifiée : $\mathrm{d}_u[v] = 0 \iff u = v$.

Longueur d'un chemin, distance entre deux sommets

- La longueur d'un chemin γ est le nombre d'arêtes/d'arcs qui le composent. On la note $\delta(\gamma)$
- Par convention, il existe un chemin de longueur nulle d'un sommet s à lui-même. *(il ne faut pas confondre avec les éventuels chemins* $(s, s, \ldots, s)$*, si le graphe contient la boucle* (s, s)*)*
- La distance $d_u[v]$ d'un sommet u à un sommet v est le minimum des longueurs des chemins menant de u à v.
- En notant $\Gamma_u(v)$ l'ensemble des chemins de u à v on a donc :

$$d_u[v] = \min_{\gamma \in \Gamma_u(v)} \delta(\gamma)$$

- Une « vraie » distance ?
  - ◇ La propriété de séparation est vérifiée : $\mathrm{d}_u[v] = 0 \iff u = v$.
  - ◇ L'inégalité triangulaire est vérifiée : $\mathrm{d}_u[v] \leqslant \mathrm{d}_u[w] + \mathrm{d}_w[v]$.

Longueur d'un chemin, distance entre deux sommets

- La longueur d'un chemin γ est le nombre d'arêtes/d'arcs qui le composent. On la note $\delta(\gamma)$
- Par convention, il existe un chemin de longueur nulle d'un sommet s à lui-même. *(il ne faut pas confondre avec les éventuels chemins* $(s, s, \ldots, s)$*, si le graphe contient la boucle* (s, s)*)*
- La distance $d_u[v]$ d'un sommet u à un sommet v est le minimum des longueurs des chemins menant de u à v.
- En notant $\Gamma_u(v)$ l'ensemble des chemins de u à v on a donc :

$$d_u[v] = \min_{\gamma \in \Gamma_u(v)} \delta(\gamma)$$

- Une « vraie » distance ?
  - ◇ La propriété de séparation est vérifiée : $\mathrm{d}_u[v] = 0 \iff u = v$.
  - ◇ L'inégalité triangulaire est vérifiée : $\mathrm{d}_u[v] \leqslant \mathrm{d}_u[w] + \mathrm{d}_w[v]$.
  - ◇ La propriété de symétrie est bien vérifiée pour les graphes non-orientés, elle ne l'est pas toujours dans les graphes orientés.

Exemples dans un graphe non orienté

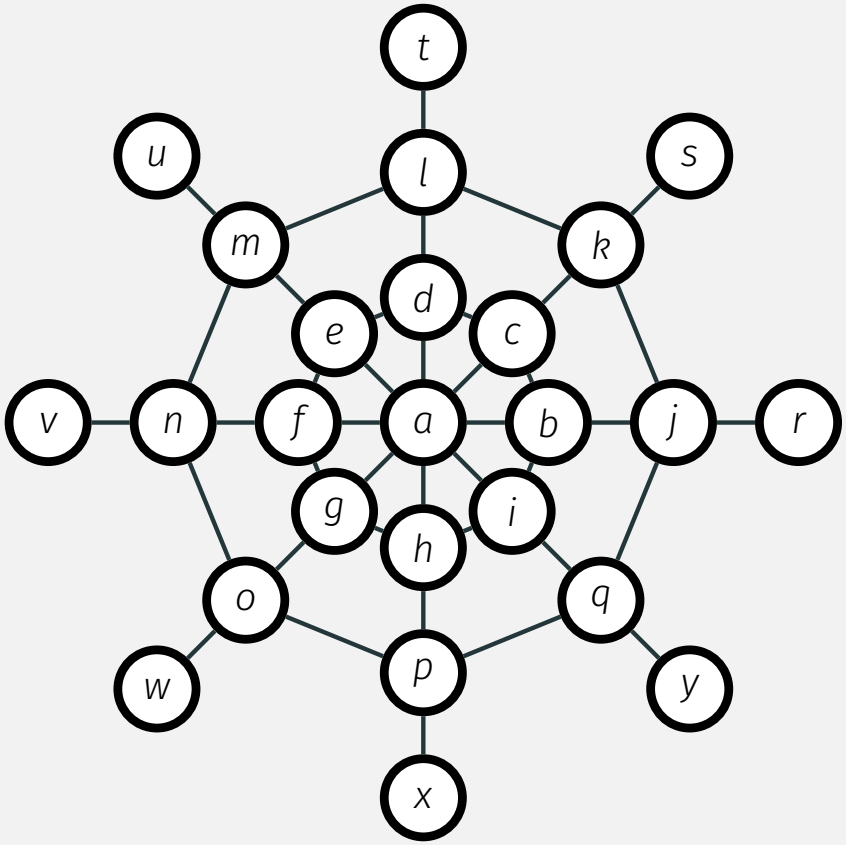


Figure 2 – Graphe G_1 non orienté et non pondéré

- $\gamma_1 = (a, b, j, r)$ et $\delta(\gamma_1) = 3$,

Exemples dans un graphe non orienté

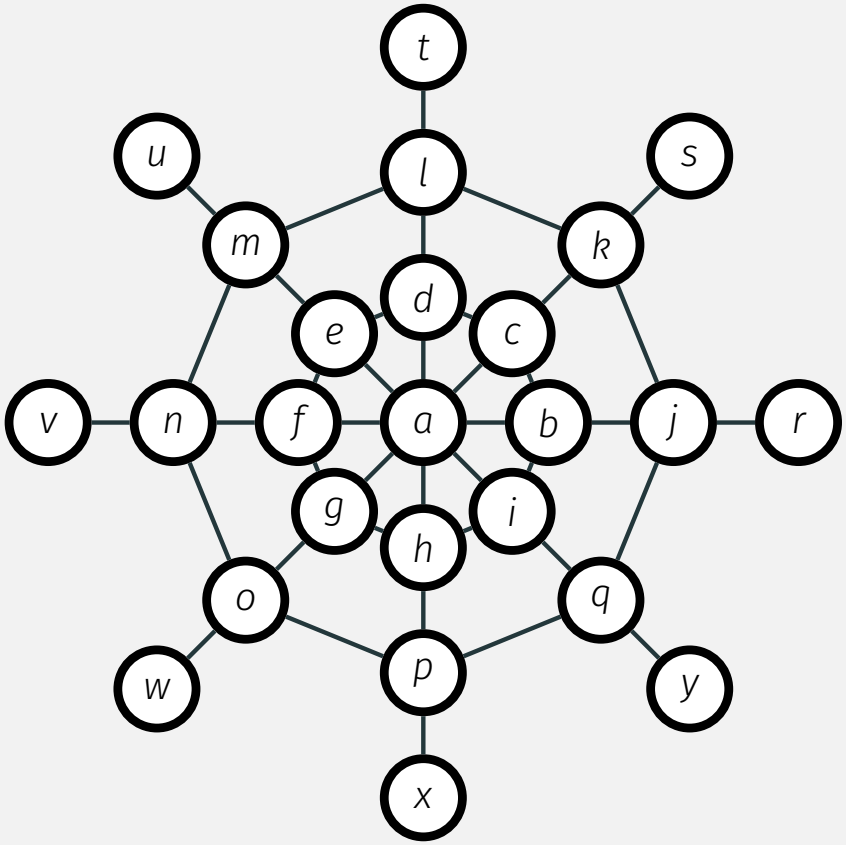


Figure 2 – Graphe G_1 non orienté et non pondéré

- $\gamma_1 = (a, b, j, r)$ et $\delta(\gamma_1) = 3$,
- $\gamma_2 = (a, e, d, l, k, j, r)$ et $\delta(\gamma_2) = 6$,

Exemples dans un graphe non orienté

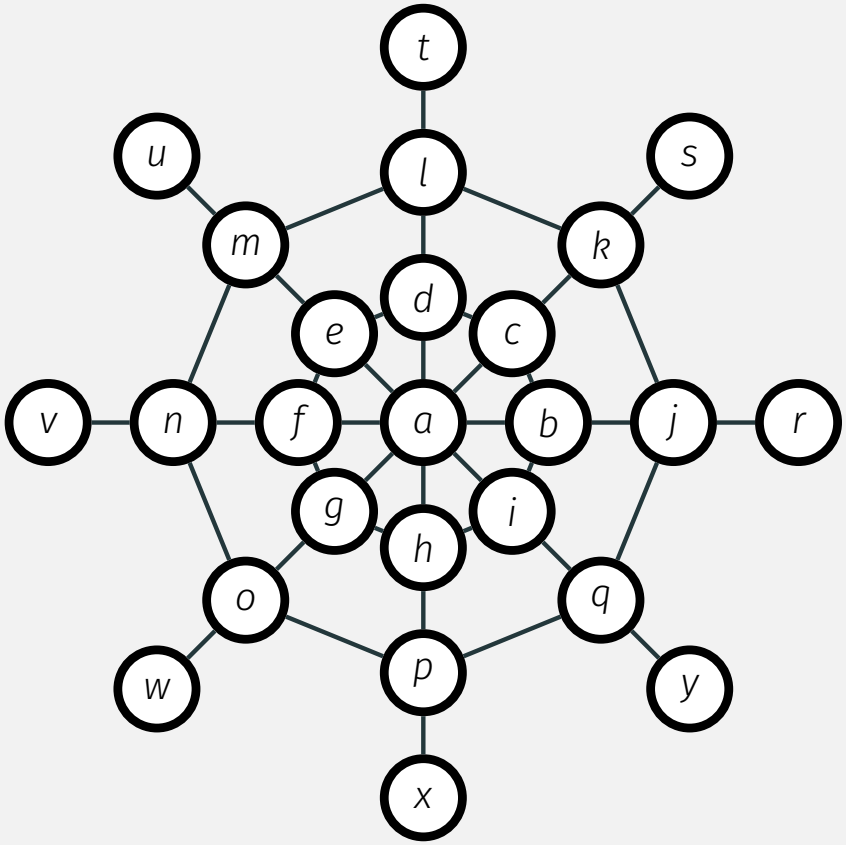


Figure 2 – Graphe G_1 non orienté et non pondéré

- $\gamma_1 = (a, b, j, r)$ et $\delta(\gamma_1) = 3$,
- $\gamma_2 = (a, e, d, l, k, j, r)$ et $\delta(\gamma_2) = 6$,
- $\mathrm{d}_a[j] = 2$ et $\mathrm{d}_a[r] = 3$.

Exemples dans un graphe orienté

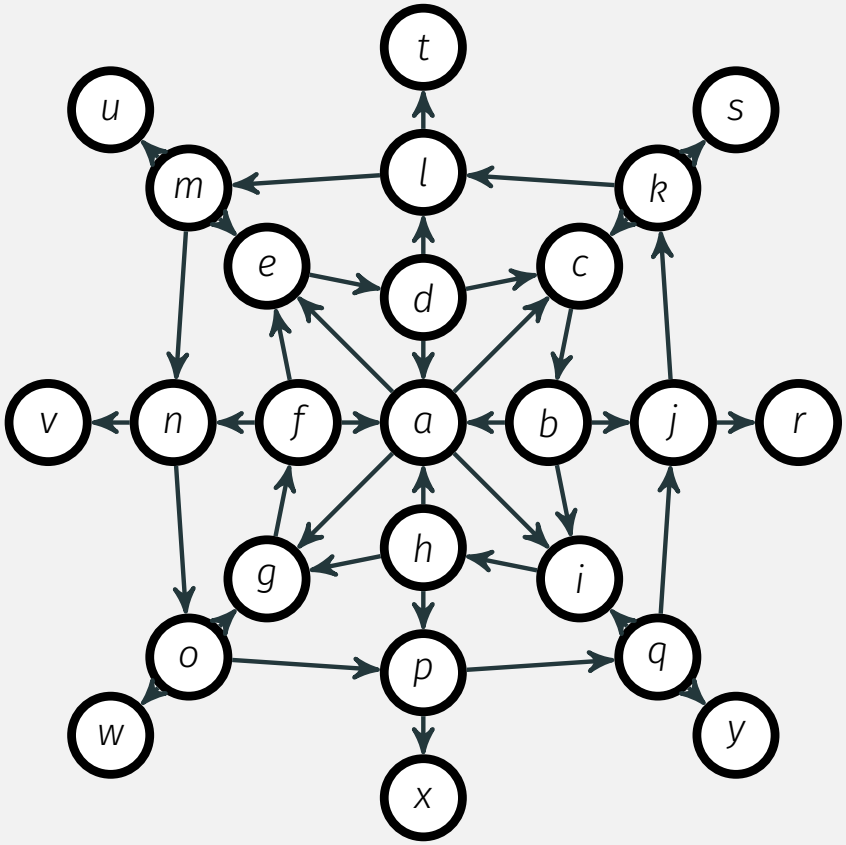


Figure 3 – Graphe G_{1b} orienté et non pondéré

- $\gamma_3 = (a, c, b, j, r)$ et $\delta(\gamma_3) = 4$,

Exemples dans un graphe orienté

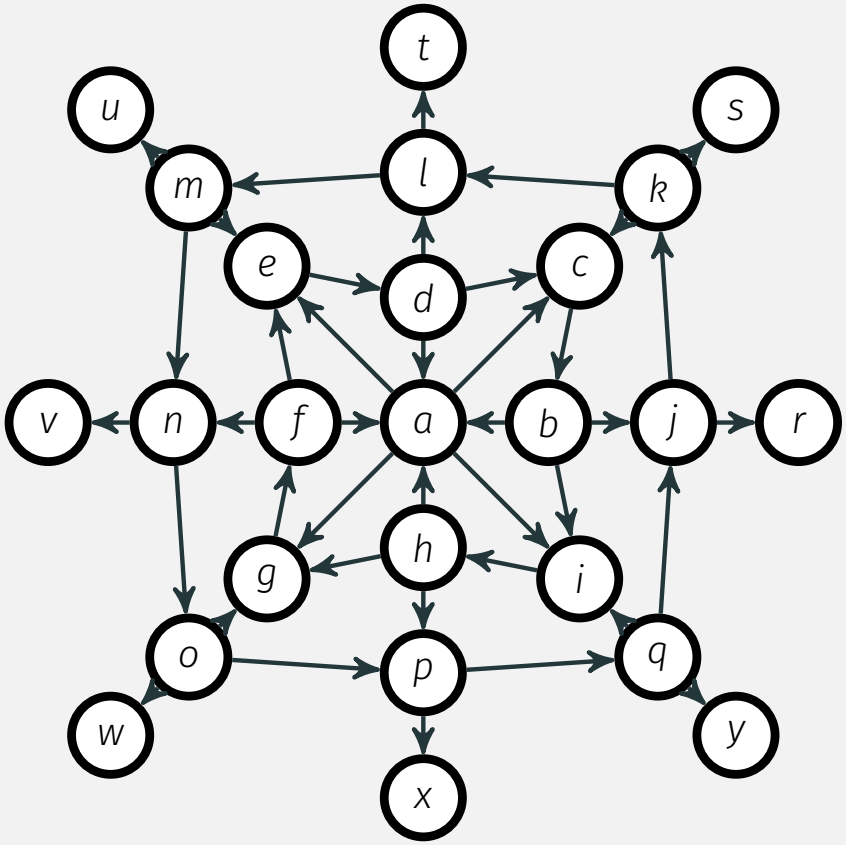


Figure 3 – Graphe G_{1b} orienté et non pondéré

- $\gamma_3 = (a, c, b, j, r)$ et $\delta(\gamma_3) = 4$,
- $\gamma_4 = (a, i, h, p, q, j, r)$ et $\delta(\gamma_4) = 6$,

Exemples dans un graphe orienté

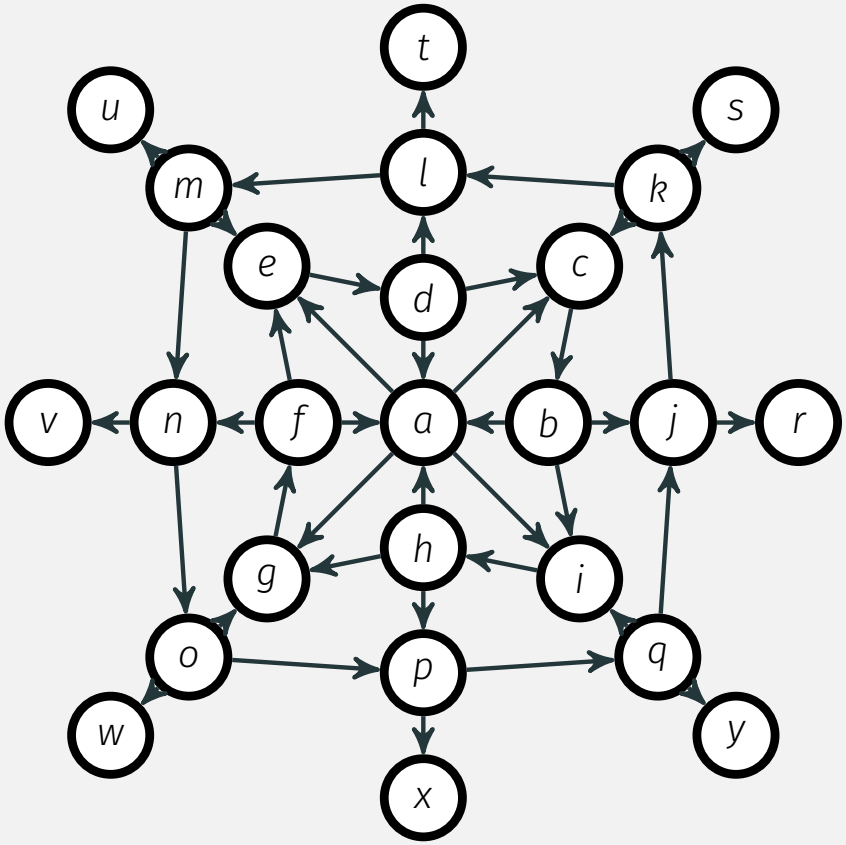


Figure 3 – Graphe G_{1b} orienté et non pondéré

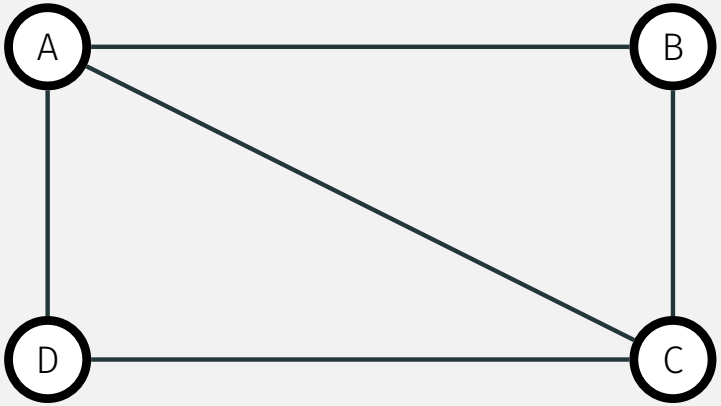
- $\gamma_3 = (a, c, b, j, r)$ et $\delta(\gamma_3) = 4$,
- $\gamma_4 = (a, i, h, p, q, j, r)$ et $\delta(\gamma_4) = 6$,
- $\mathrm{d}_p[q] = 1$ et $\mathrm{d}_q[p] = 3$.

Connexité dans un graphe non-orienté

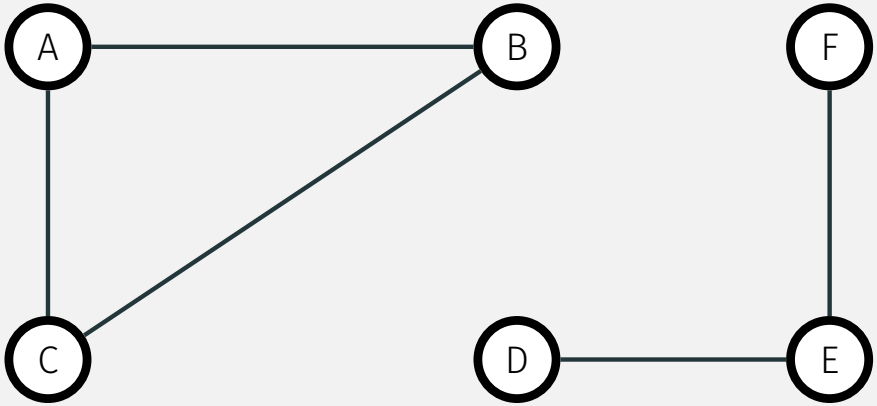
Un graphe *non orienté* est dit *connexe* si, quels que soient les sommets u et v de de graphe, il existe un chemin reliant u et v.

Connexité dans un graphe non-orienté

Un graphe *non orienté* est dit *connexe* si, quels que soient les sommets u et v de de graphe, il existe un chemin reliant u et v.



Graphe connexe



Graphe non connexe

Graphes étiquetés, pondération d'un graphe

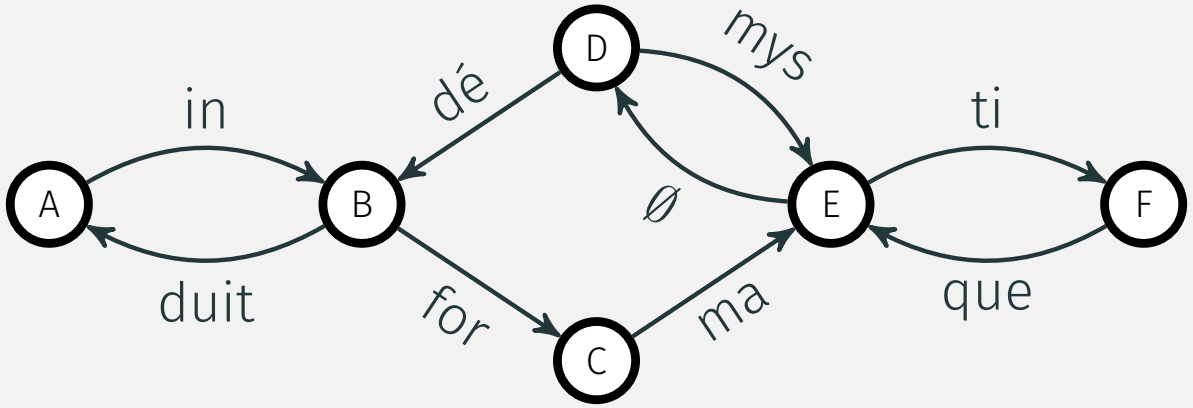
- On dit qu'un graphe (orienté ou non) est *étiqueté* lorsque ses arêtes (ou arcs) sont affectées d'étiquettes (mots, lettres, symboles, chiffres...).

Graphes étiquetés, pondération d'un graphe

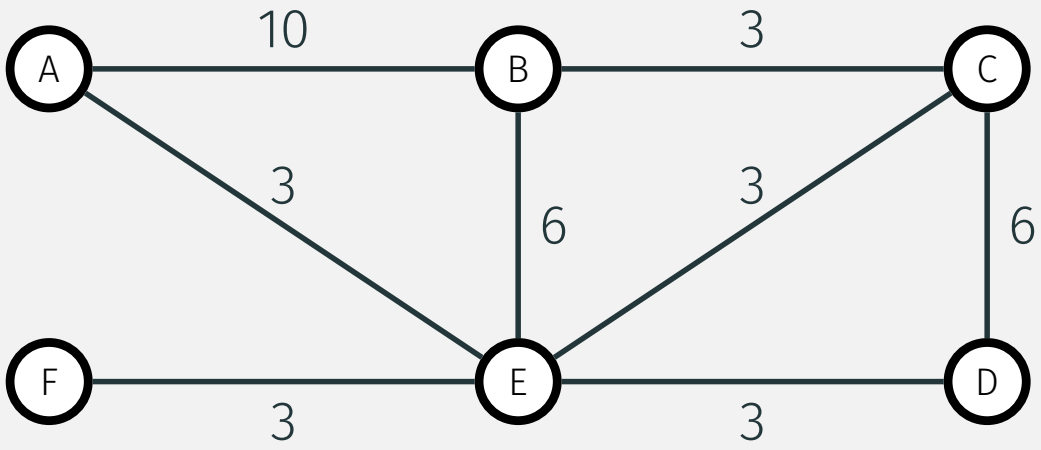
- On dit qu'un graphe (orienté ou non) est *étiqueté* lorsque ses arêtes (ou arcs) sont affectées d'étiquettes (mots, lettres, symboles, chiffres...).
- Le graphe est dit *pondéré* lorsque c'est un graphe étiqueté dont toutes les étiquettes sont des nombres réels positifs ou nuls.

Graphes étiquetés, pondération d'un graphe

- On dit qu'un graphe (orienté ou non) est *étiqueté* lorsque ses arêtes (ou arcs) sont affectées d'étiquettes (mots, lettres, symboles, chiffres...).
- Le graphe est dit *pondéré* lorsque c'est un graphe étiqueté dont toutes les étiquettes sont des nombres réels positifs ou nuls.



Un exemple de graphe orienté et étiqueté.



Un exemple de graphe non orienté et pondéré.

Implémentation en Python

Implémentations

Nous présentons dans cette partie deux manières différentes de représenter en langage Python la notion abstraite de graphe.

Implémentations

Nous présentons dans cette partie deux manières différentes de représenter en langage Python la notion abstraite de graphe. Il s'agit de choisir une **structure de données** permettant de programmer, le plus efficacement possible, les fonctionnalités liées à la manipulation des graphes. Ces deux possibilités sont :

Implémentations

Nous présentons dans cette partie deux manières différentes de représenter en langage Python la notion abstraite de graphe. Il s'agit de choisir une **structure de données** permettant de programmer, le plus efficacement possible, les fonctionnalités liées à la manipulation des graphes. Ces deux possibilités sont :

- l'utilisation de matrices d'adjacence, qui reposent sur des **tableaux bi-dimensionnels** du module Numpy ;

Implémentations

Nous présentons dans cette partie deux manières différentes de représenter en langage Python la notion abstraite de graphe. Il s'agit de choisir une **structure de données** permettant de programmer, le plus efficacement possible, les fonctionnalités liées à la manipulation des graphes. Ces deux possibilités sont :

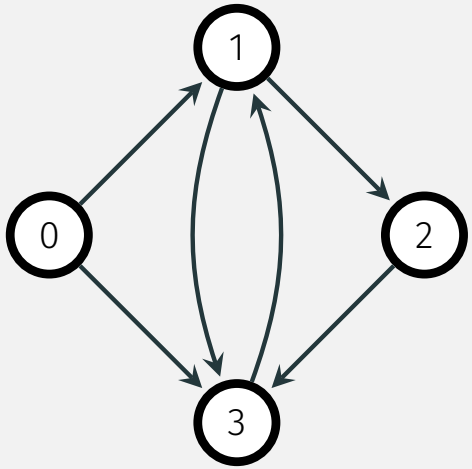
- l'utilisation de matrices d'adjacence, qui reposent sur des **tableaux bi-dimensionnels** du module Numpy ;
- l'utilisation de listes d'adjacence, qui reposent sur des **dictionnaires**.

Matrice d'adjacence

On considère un graphe **orienté** formé de N noeuds, désignés par les entiers de 0 à $N-1$. Il est représenté par une matrice de N lignes et N colonnes, dont la case de ligne i et colonne j contient le booléen `True` s'il y a un arc du noeud i vers le noeud j, et `False` sinon.

Matrice d'adjacence

On considère un graphe **orienté** formé de N noeuds, désignés par les entiers de 0 à $N-1$. Il est représenté par une matrice de N lignes et N colonnes, dont la case de ligne i et colonne j contient le booléen `True` s'il y a un arc du noeud i vers le noeud j, et `False` sinon.



De matrice d'adjacence :

	0	1	2	3
0	`False`	`True`	`False`	`True`
1	`False`	`False`	`True`	`True`
2	`False`	`False`	`False`	`True`
3	`False`	`True`	`False`	`False`

Matrice d'adjacence

Cette **matrice d'adjacence** est implémentée à l'aide d'un tableau bidimensionnel du module `numpy`.

Matrice d'adjacence

Cette **matrice d'adjacence** est implémentée à l'aide d'un tableau bidimensionnel du module `numpy`. La fonction `array` de ce module construit un tableau à partir de la liste de ses lignes. Par exemple, la matrice de l'exemple précédent s'obtient par :

```
import numpy as np
G = np.array([
  [False, True, False, True],
  [False, False, True, True],
  [False, False, False, True],
  [False, True, False, False]])
```

Matrice d'adjacence

Cette **matrice d'adjacence** est implémentée à l'aide d'un tableau bidimensionnel du module `numpy`. La fonction `array` de ce module construit un tableau à partir de la liste de ses lignes. Par exemple, la matrice de l'exemple précédent s'obtient par :

```
import numpy as np
G = np.array([
  [False, True, False, True],
  [False, False, True, True],
  [False, False, False, True],
  [False, True, False, False]])
```

Un graphe d'ordre N sans aucun arc pourra être défini par la commande :

```
G = np.zeros((N,N), dtype=bool)
```

Exercice

La variable G contient la matrice d'adjacence d'un graphe orienté.

1. Écrire une fonction `nb_arcs(G : np.array) -> int` renvoyant le nombre total d'arcs du graphe.

Exercice

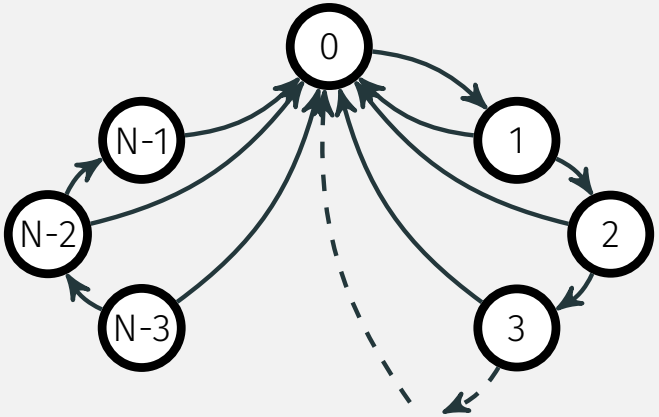
La variable G contient la matrice d'adjacence d'un graphe orienté.

1. Écrire une fonction `nb_arcs(G : np.array) -> int` renvoyant le nombre total d'arcs du graphe.
2. Écrire une fonction `voisins(G : np.array , n : int) -> list` renvoyant la liste des noeuds voisins du noeud n (noeuds accessibles à partir de n en suivant un seul arc).

Exercice

La variable G contient la matrice d'adjacence d'un graphe orienté.

1. Écrire une fonction `nb_arcs(G : np.array) -> int` renvoyant le nombre total d'arcs du graphe.
2. Écrire une fonction `voisins(G : np.array , n : int) -> list` renvoyant la liste des noeuds voisins du noeud n (noeuds accessibles à partir de n en suivant un seul arc).
3. Écrire une fonction `un_graphe(N : int) -> np.array` renvoyant un graphe formé de $N \geqslant 2$ noeuds (numérotés de 0 à $N-1$) où chaque noeud $1 \leqslant i \leqslant N-2$ a pour voisins les deux noeuds 0 et $i+1$ (le noeud 0 ayant pour unique voisin 1, et le noeud $N-1$ pour unique voisin 0) :



Corrigé : 1)

```
def nb_arcs(G : np.array) -> int :
    """
    Renvoie le nombre d'arcs contenus dans le graphe \
    ↪ de matrice d'adjacence G
    """
    N = len(G) # nombre de sommets = nombre de \
    ↪ lignes de la matrice
    nba = 0
    for i in range(N):
        for j in range(N):
            if G[i,j]:
                nba += 1
    return nba
```

Corrigé : 2)

```
def voisins(G : np.array , n : int) -> list :
    """
    Renvoie la liste des noeuds voisins du noeud n \
    ↪ dans le
    graphe de matrice d'adjacence G.
    Attention ! n doit être un noeud du graphe.
    """
    N = len(G)
    assert 0 <= n < N  # Vérifie que n est un noeud
    vois = []
    for j in range(N):
        if G[n,j]:
            vois.append(j)
    return vois
```

Corrigé : 3)

```python
def un_graphe(N : int) -> list :
    """
    Renvoie la matrice d'adjacence d'un graphe à N \
    ↪ noeuds où le noeud 1<=i<=N-2
    a pour voisins les noeuds 0 et i+1 (le noeud 0 \
    ↪ ayant pour unique voisin 1,
    et le noeud N-1 pour unique voisin 0).
    Attention ! on doit avoir N>=2
    """
    assert N>=2
    G = np.zeros((N, N),dtype=bool)
    G[0, 1] = True
    for i in range(1, N-1):
        G[i, 0] = True
        G[i, i+1] = True
    G[N-1, 0] = True
    return G
```

Bilan sur les matrices d'adjacence

Bilan sur les matrices d'adjacence

- Avantage : simplicité de la mise en oeuvre.

Bilan sur les matrices d'adjacence

- Avantage : simplicité de la mise en oeuvre.
- Inconvénient : coût en mémoire proportionnel à N^2, inutilement élevé si le graphe comporte peu d'arcs.

Liste/dictionnaire d'adjacence

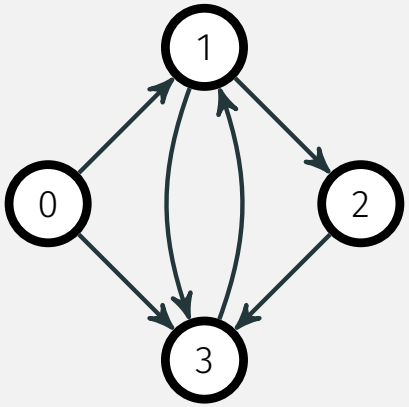
Le graphe est stocké en utilisant un **dictionnaire G**, dont une **clé** est un noeud du graphe, et la **valeur** associée est la liste de ses noeuds voisins.

Liste/dictionnaire d'adjacence

Le graphe est stocké en utilisant un **dictionnaire G**, dont une **clé** est un noeud du graphe, et la **valeur** associée est la liste de ses noeuds voisins. Les noeuds n'ont plus besoin d'être des entiers.

Liste/dictionnaire d'adjacence

Le graphe est stocké en utilisant un **dictionnaire G**, dont une **clé** est un noeud du graphe, et la **valeur** associée est la liste de ses noeuds voisins. Les noeuds n'ont plus besoin d'être des entiers. Exemple :

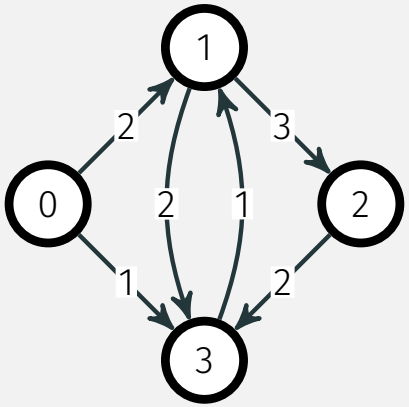



```
0 : [1,3], 1 : [2,3], 2 : [3], 3 : [1]
```

Liste/dictionnaire d'adjacence

Dans le cas d'un graphe pondéré, on ajoute simplement les poids dans les valeurs du dictionnaire.

Dans le cas d'un graphe pondéré, on ajoute simplement les poids dans les valeurs du dictionnaire. Exemple :




```
0 : [(1, 2), (3, 1)], 1 : [(2, 3), (3, 2))], 2 : [(3, 2)], 3 : [(1, 1)]
```

Exercice

`G` contient le dictionnaire d'adjacence d'un graphe.
Écrire les versions correspondantes des fonctions `nb_arcs`, `voisins` et `un_graphe` de l'exercice précédent.

Question 1 :

```
def nb_arcs(G : dict) -> int :
    """
    Renvoie le nombre d'arcs contenus dans le graphe \
    ↪ de dictionnaire d'adjacence G
    """
    nba = 0
    for s in G :
        nba += len(G[s]) # nombre de voisins de s
    return nba
```

Question 2 :

```
def voisins(G : dict, n : int) -> list :
    """
    Renvoie la liste des noeuds du noeud n dans le \
    ↪ graphe de dictionnaire d'adjacence G.
    Attention ! n doit être un noeud du graphe.
    """
    assert n in G  # Vérifie que n est un noeud
    return G[n]
```

Question 3 :
Une solution itérative :

```
def un_graphe(N : int) -> list :
    """
    Renvoie le dictionnaire d'adjacence d'un graphe à N sommets
    où le noeud 1<=i<=N-2 a pour voisins les noeuds 0 et i+1
    (le noeud 0 ayant pour unique voisin 1, et le noeud N-1
    pour unique voisin 0).
    Attention ! on doit avoir N>=2
    """
    assert N>=2
    G = {}
    G[0] = [1]
    for i in range(1, N-1) :
         G[i] = [0,i+1]
    G[N-1] = [0]
    return G
```

Question 3 :
Une solution récursive :

```
def un_graphe(N : int) -> list :
    """
    Renvoie le dictionnaire d'adjacence d'un graphe à N sommets où le noeud \
    ↪ 1<=i<=N-2 a pour voisins les noeuds 0 et i+1 (le noeud 0 ayant pour \
    ↪ unique voisin 1, et le noeud N-1 pour unique voisin 0).
    Attention ! on doit avoir N>=2
    """
    assert N>=2
    if N == 2 :
        return {0:[1], 1:[0]}
    else :
        G = un_graphe(N-1)
        G[N-2].append(N-1)
        G[N-1] = [0]
        return G
```

Bilan sur les dictionnaires d'adjacence

Moins gourmande en espace mémoire, cette solution est généralement préférée à celle des matrices d'adjacence.

Cas d'un graphe non orienté

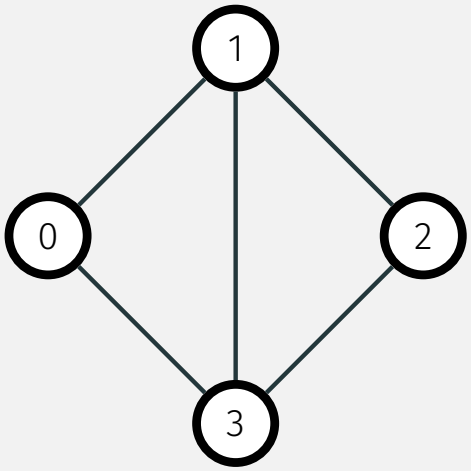
On utilise une représentation de graphe **orienté** (matrice ou liste d'adjacence) où chaque arête non orientée :

Cas d'un graphe non orienté

On utilise une représentation de graphe **orienté** (matrice ou liste d'adjacence) où chaque arête non orientée :

est représentée par les **deux** arcs orientés :

Exemple



De matrice d'adjacence :

	0	1	2	3
0	`False`	`True`	`False`	`True`
1	`True`	`False`	`True`	`True`
2	`False`	`True`	`False`	`True`
3	`True`	`True`	`True`	`False`

De dictionnaire d'adjacence : `{0 : [1,3], 1 : [0,2,3],2 : [1,3],3 : [0,1,2]}`.

Un peu de coloriage

Coloriage d'un graphe non orienté

Objectif : associer une **couleur** à chaque sommet d'un graphe non orienté (représentant une carte) de sorte que :

Coloriage d'un graphe non orienté

Objectif : associer une **couleur** à chaque sommet d'un graphe non orienté (représentant une carte) de sorte que :

- deux sommets voisins aient une couleur **différente** ;

Coloriage d'un graphe non orienté

Objectif : associer une **couleur** à chaque sommet d'un graphe non orienté (représentant une carte) de sorte que :

- deux sommets voisins aient une couleur **différente** ;
- en utilisant le **moins** de couleurs possibles.

Coloriage d'un graphe non orienté

Objectif : associer une **couleur** à chaque sommet d'un graphe non orienté (représentant une carte) de sorte que :

- deux sommets voisins aient une couleur **différente** ;
- en utilisant le **moins** de couleurs possibles.

Un célèbre théorème, dit **des quatre couleurs**, affirme qu'il est toujours possible d'obtenir un tel coloriage en utilisant au plus quatre couleurs.

Un algorithme glouton

- Théorème vrai plus généralement pour les graphes *planaires* : c'est-à-dire représentable dans le plan sans qu'aucune arête (ou arc pour un graphe orienté) n'en croise une autre.

Un algorithme glouton

- Théorème vrai plus généralement pour les graphes *planaires* : c'est-à-dire représentable dans le plan sans qu'aucune arête (ou arc pour un graphe orienté) n'en croise une autre.
- Mais il n'existe pas d'algorithme efficace permettant d'obtenir un tel coloriage optimal à tous les coups.

Un algorithme glouton

- Théorème vrai plus généralement pour les graphes *planaires* : c'est-à-dire représentable dans le plan sans qu'aucune arête (ou arc pour un graphe orienté) n'en croise une autre.
- Mais il n'existe pas d'algorithme efficace permettant d'obtenir un tel coloriage optimal à tous les coups.
- Voici un algorithme **glouton** conduisant généralement à une bonne solution :

Un algorithme glouton

- Théorème vrai plus généralement pour les graphes *planaires* : c'est-à-dire représentable dans le plan sans qu'aucune arête (ou arc pour un graphe orienté) n'en croise une autre.
- Mais il n'existe pas d'algorithme efficace permettant d'obtenir un tel coloriage optimal à tous les coups.
- Voici un algorithme **glouton** conduisant généralement à une bonne solution :
  - ◇ les couleurs sont représentées par des entiers successifs à partir de 0 ;

Un algorithme glouton

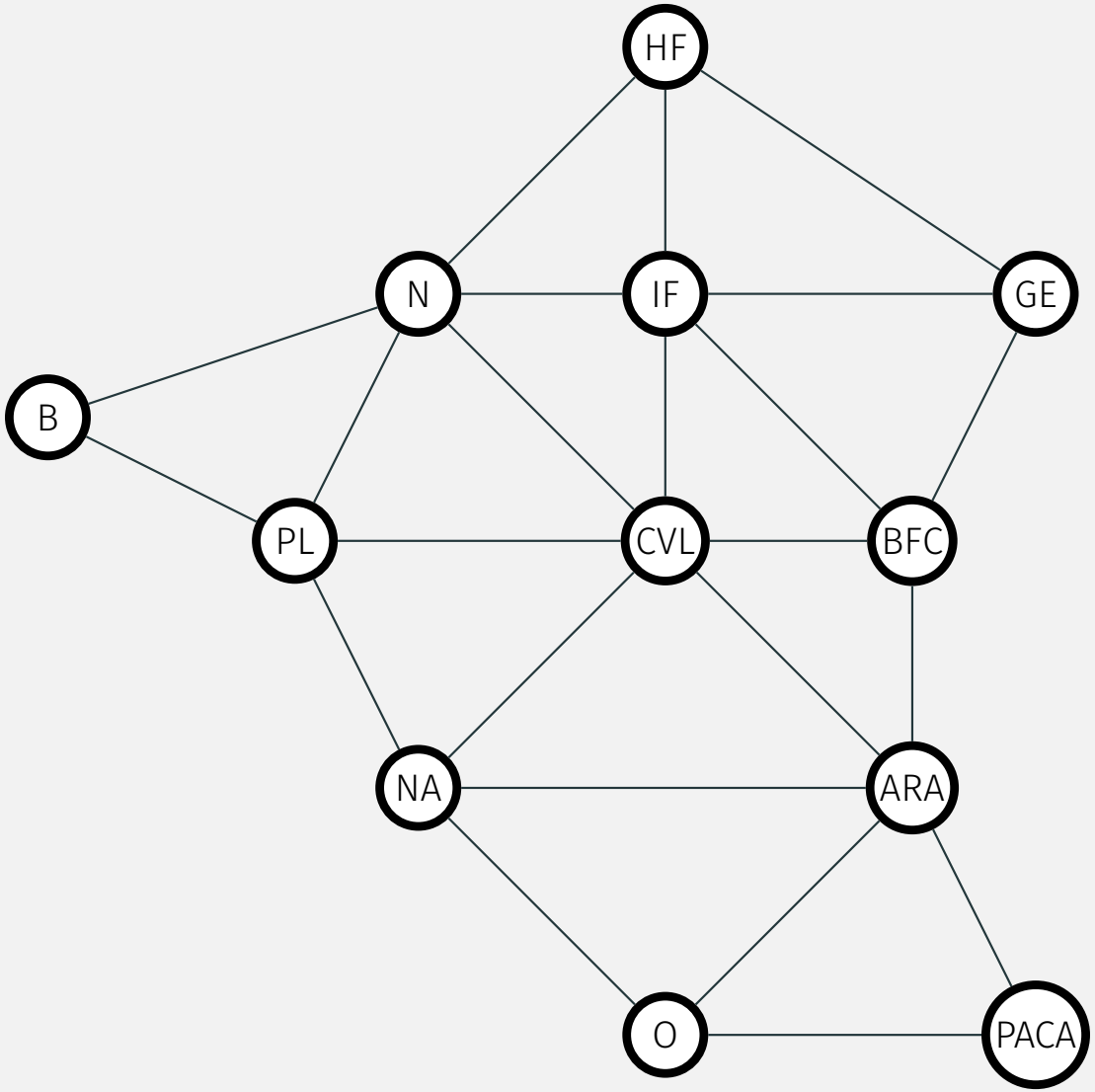
- Théorème vrai plus généralement pour les graphes *planaires* : c'est-à-dire représentable dans le plan sans qu'aucune arête (ou arc pour un graphe orienté) n'en croise une autre.
- Mais il n'existe pas d'algorithme efficace permettant d'obtenir un tel coloriage optimal à tous les coups.
- Voici un algorithme **glouton** conduisant généralement à une bonne solution :
  - les couleurs sont représentées par des entiers successifs à partir de 0 ;
  - on parcourt un à un tous les sommets (dans un ordre quelconque), et on associe au sommet courant la plus petite couleur qui n'est pas déjà utilisée pour ses sommets voisins.

  Cet algorithme sera programmé en TP.

Exemple

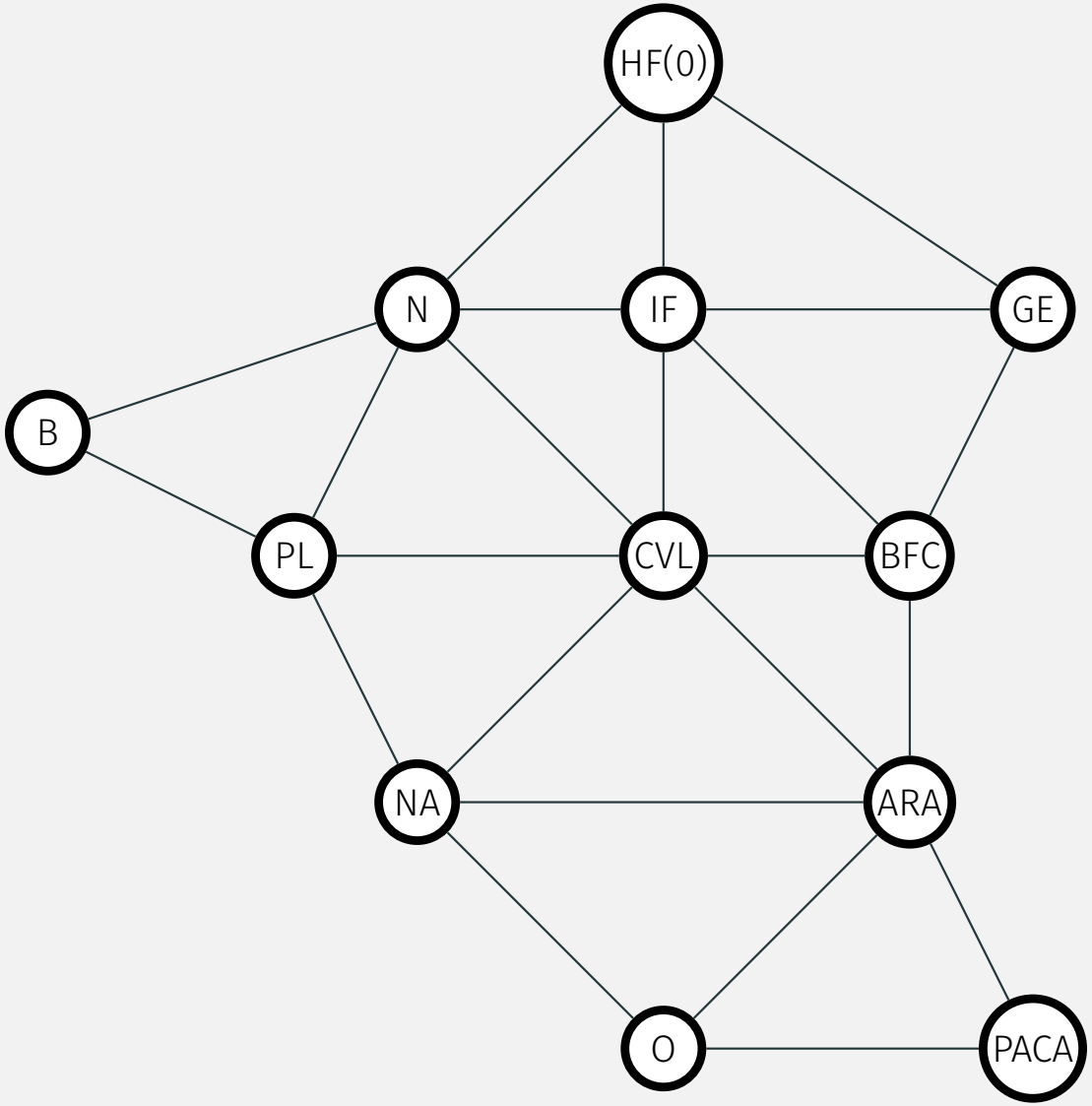
Voici le graphe d'adjacence des régions de France métropolitaine continentale. (HF : hauts-de-France, N : Normandie, IF :île-de-France, etc).

Exemple

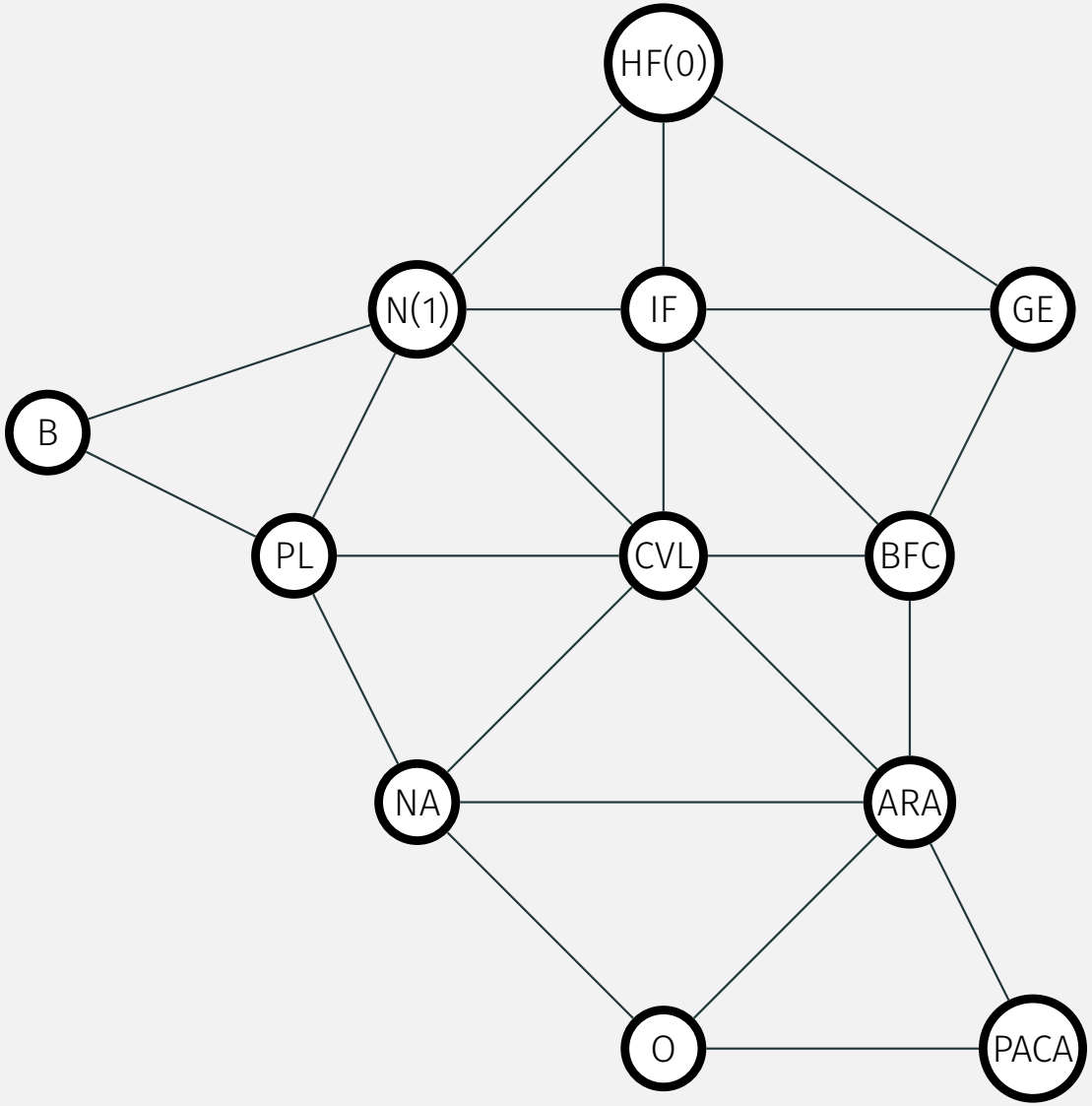


Dans un sens de parcours nord-sud et ouest-est (HF, N, IF, GE, B, etc) :

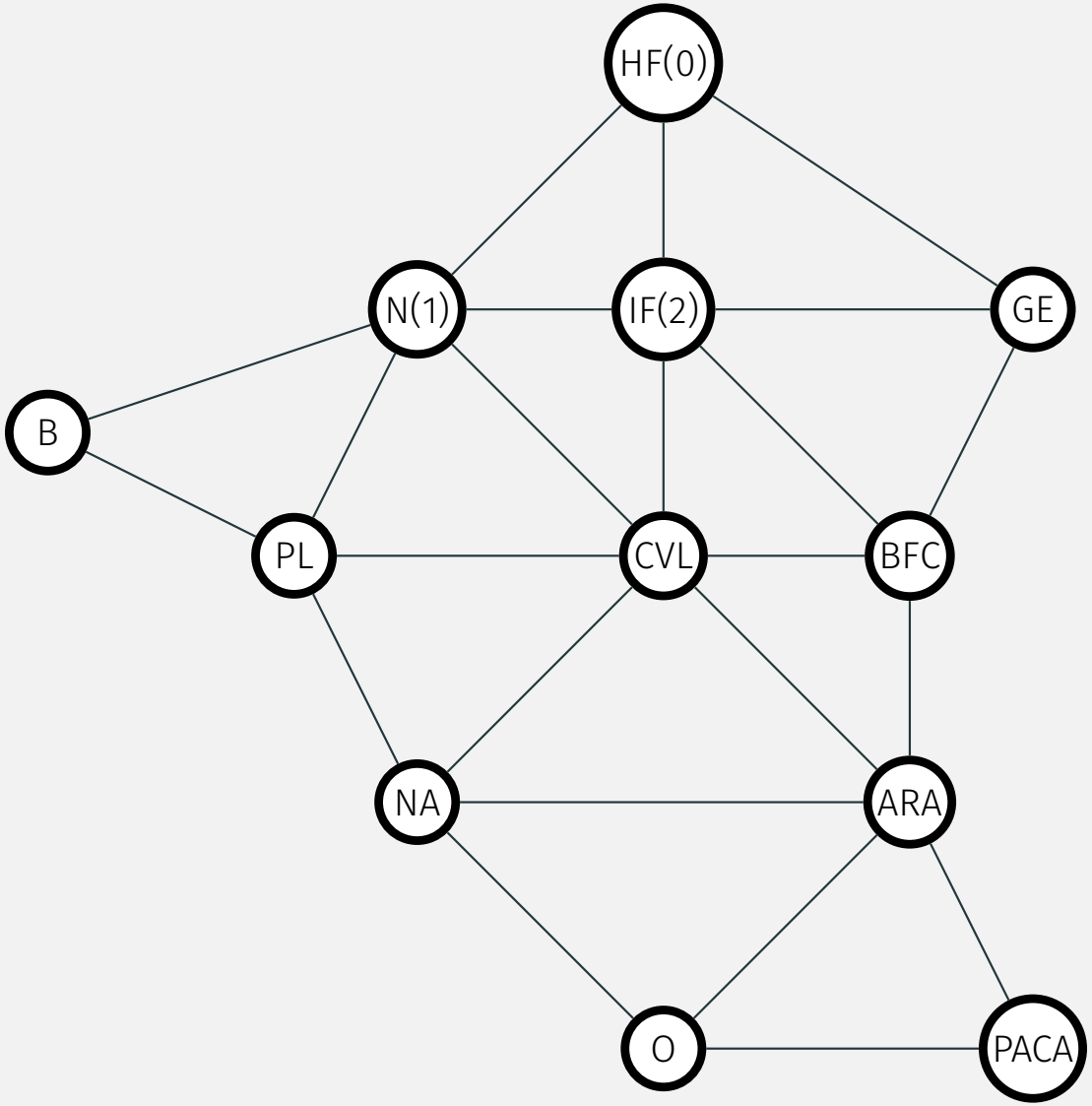
Exemple



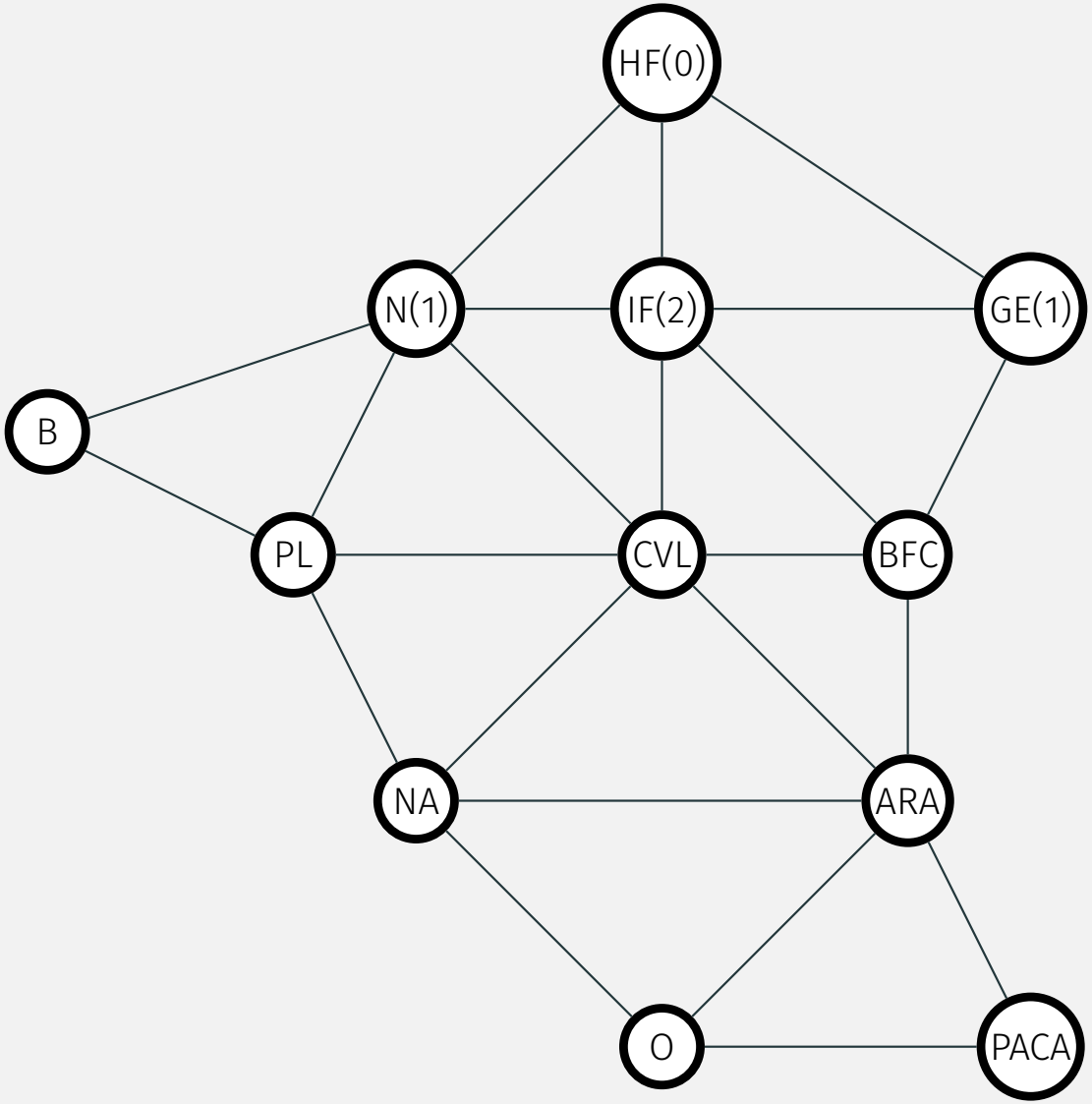
Exemple



Exemple

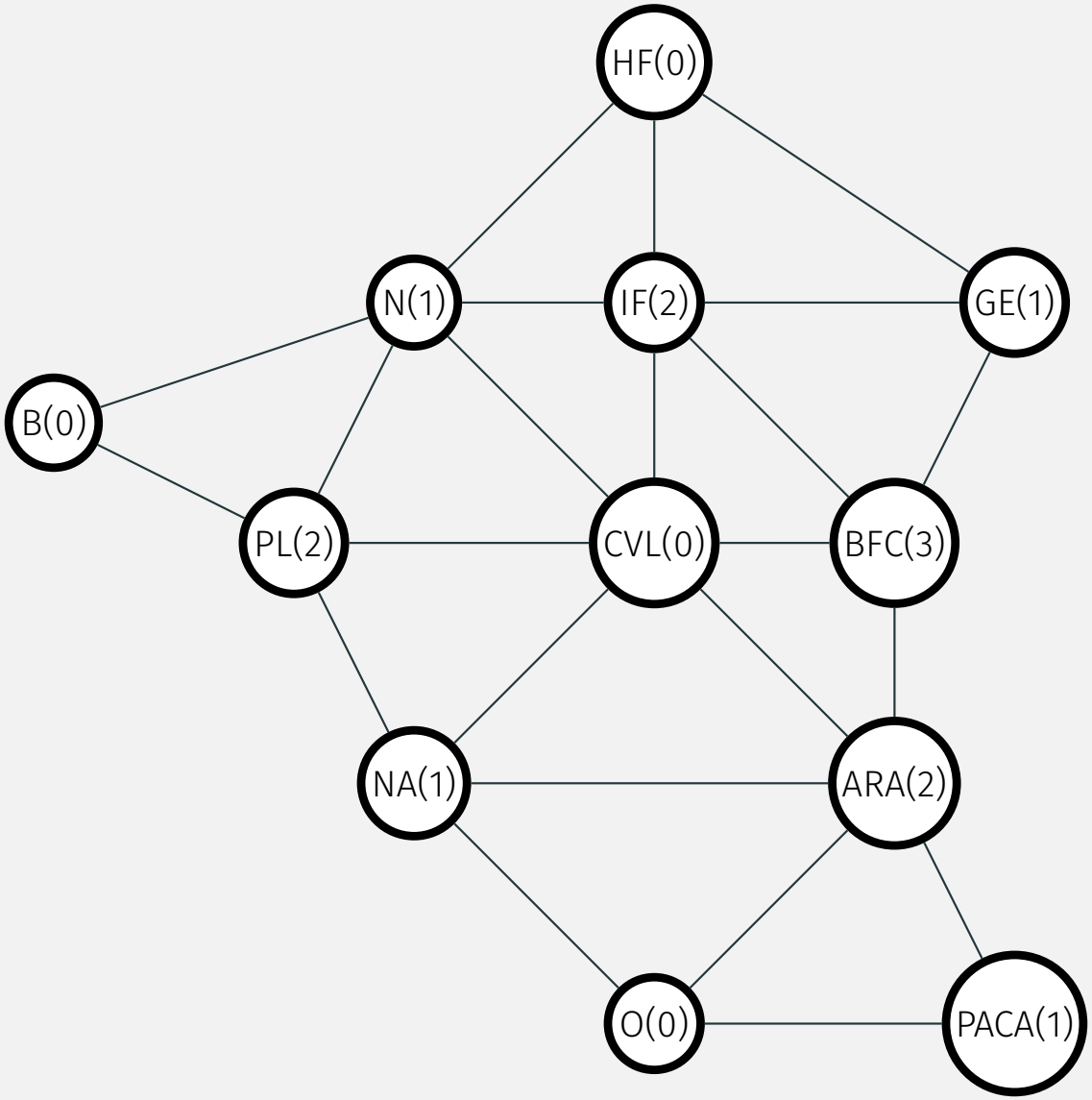


Exemple



Et ainsi de suite, jusqu'à obtenir

Exemple



Exemple

Le coloriage obtenu est **optimal** :

Exemple

Le coloriage obtenu est **optimal** :

- il utilise seulement 4 couleurs,

Exemple

Le coloriage obtenu est **optimal** :

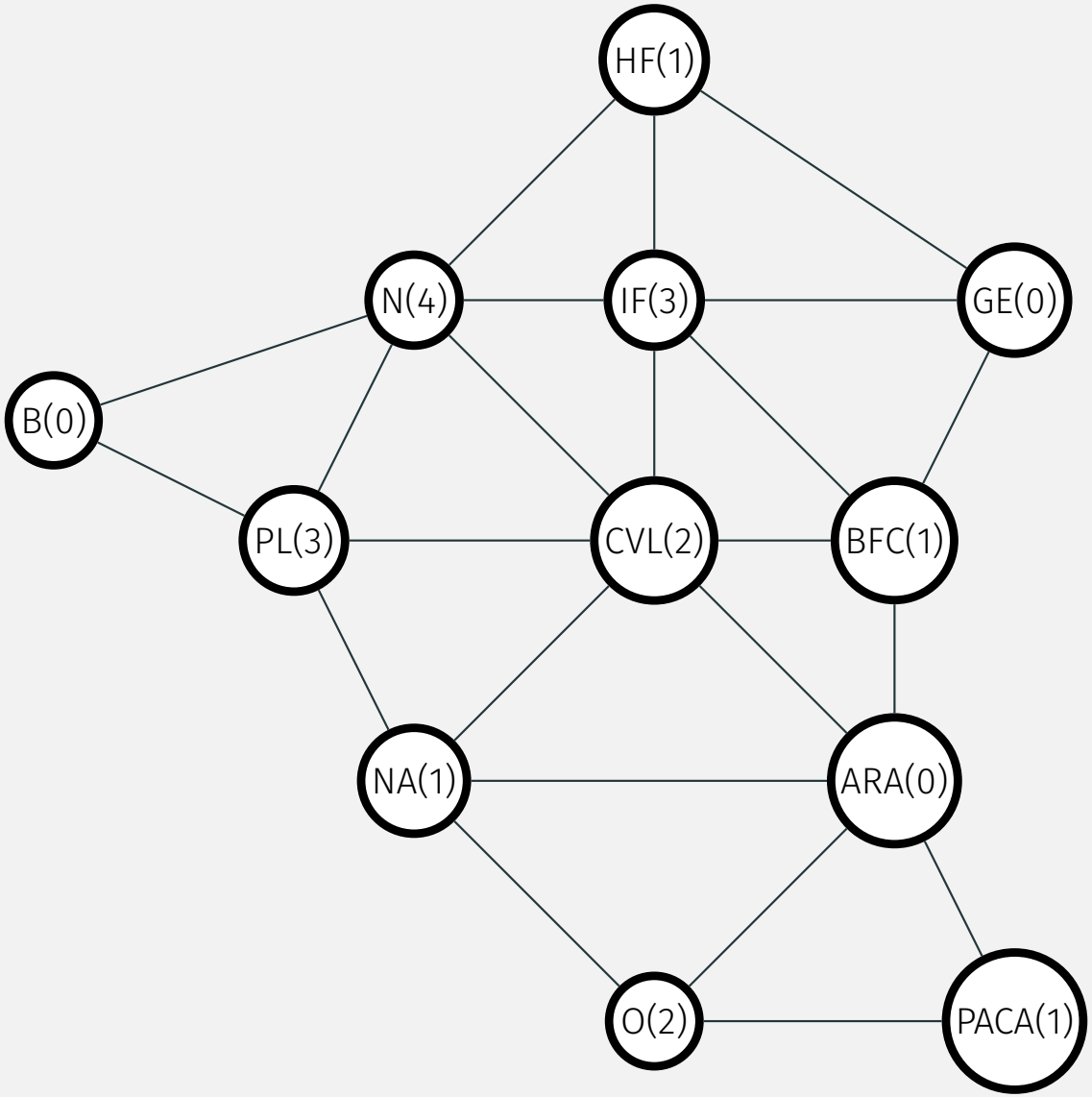
- il utilise seulement 4 couleurs,
- et aucun coloriage à 3 couleurs serait possible.

Exemple

Un autre coloriage pour le sens de parcours :

$$ARA, BFC, B, CVL, GE, HF, IF, N, NA, O, PL, PACA$$

Exemple



Exemple

Le coloriage obtenu **n'est pas optimal** puisque 5 couleurs ont été utilisées.