Chapitre (S2) 3 Notion de graphe

Objectifs

- Connaître 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.
- Savoir mettre en oeuvre des algorithmes de traitement des graphes.

Structure de données. Une *structure de données* est la description d'une structure logique destinée à organiser et à agir sur des données.

Les structures de données *linéaires* l'organisent de manière séquentielle comme dans les *tableaux*, les *listes*, les *piles* ou les *files*[1]. Chaque donnée est précédée et suivie d'une autre donnée.

Certains problèmes exigent une *organisation non-linéaire* des données. Les *arbres* sont une telle structure de données qui organise les informations de manière *hiérarchique*. Les *graphes* en sont une autre qui organise les informations suivant un schéma dit *relationnel*.

Les graphes : origine et motivations. De nombreux problèmes sont des mises en situation des graphes et expliquent l'intérêt considérable porté au sujet.

- Par exemple, les réseaux informatiques permettent l'interconnexion d'ordinateurs, la communication entre les machines, à sens unique ou bidirectionnelle, se faisant via des liens[2] établis entre elles. Chaque machine constitue un *noeud* d'un graphe dont les liens sont appelés *arêtes*. Lorsque toutes les communications possibles sont bidirectionnelles, le graphe modélisé est dit *non orienté* ; dans le cas contraire, il est qualifié de graphe *orienté*.
- Un autre exemple est celui du réseau routier modélisable par un graphe dont les sommets du graphe sont les villes et ses arêtes sont les routes. Plus localement, un réseau urbain peut être modélisé par un graphe dont les arêtes sont les rues et les sommets leurs intersections. La recherche d'un *plus court chemin*[3] dans un tel réseau est une application de certains algorithmes de graphes.
- Rechercher le chemin menant à la sortie d'un labyrinthe est également un problème que l'approche en termes de graphe permet d'étudier, en particulier à travers la thématique des algorithmes de *parcours* de graphes.
- Les jeux qui font évoluer une configuration vers une nouvelle configuration en vue d'aboutir à une situation finale particulière peuvent être traités par des algorithmes sur les graphes.

Ainsi, d'un problème historiquement mathématique[4], le thème des graphes est présent dans de nombreuses branches de l'activité humaine[5]. Il constitue un intense domaine d'activité informatique tant d'un point de vue théorique que d'un point de vue pratique. L'algorithmique des graphes fourmille de solutions astucieuses et variées que la complexité temporelle rend parfois inutilisables ! Des algorithmes d'une autre nature, menant à des solutions approchées, leur sont parfois préférés. Appelées *heuristiques*, l'une de ces solutions sera abordée dans un prochain chapitre.

L'objet de ce chapitre et des deux suivants est de formaliser la notion de graphe puis d'en donner une description informatique à l'aide de deux *implémentations*. Des *fonctions de manipulation* des graphes sont ensuite définies pour pouvoir travailler sur ces derniers. En particulier, la question du *parcours* des graphes fait l'objet des Chapitres (S2) 5 et (S2) 4.

1. Les listes et les tableaux ont été vus en première partie d'année. Les piles et les files sont introduites dans les prochains chapitres.
2. Physiques avec les câbles ou immatériels avec les communications hertziennes.
3. En pratique, cela signifie utiliser un GPS !
4. On attribue à Euler et à sa résolution du problème des ponts de Koenisberg le début de la théorie des graphes.
5. Réseaux électriques, réseaux biologiques, réseaux sociaux, etc.

1. GÉNÉRALITÉS

1.1. Graphes orientés et non orientés

Définition 1 | Graphe

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

- $S = \{v_1, \dots, v_n\}$ est l'ensemble des *sommets* (ou *noeuds*) du graphe.
- $A = \{e_1, \dots, e_m\}$ 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 Card S.

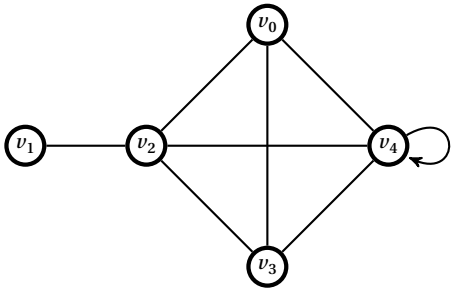
Remarque 1 On rencontre souvent la notation G = (V, E) pour désigner un graphe. L'ensemble des sommets est noté V, pour *vertices* en anglais ; l'ensemble des arêtes est noté E, pour *edges* en anglais.

Les arêtes d'un graphe peuvent être orientées de sorte que le graphe est dit *orienté*. Un graphe pour lequel les arêtes ne sont pas orientée est dit *non orienté*.[6]

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.

EXEMPLE DE GRAPHE NON ORIENTÉ. Considérons le *graphe non oriénté* $G_1 = (S_1, A_1)$ d'ordre 5 dont les ensembles S_1 et A_1 dont définis ci-dessous en termes des sommets v_0, v_1, v_2, v_3, v_4. La Figure 1 est une représentation graphique de G_1. Lorsque deux lignes se croisent, elles n'établissent pas pour autant de lien entre elles. Cette représentation graphique n'est pas unique. Une infinité de représentation topologiquement *(C'est-à-dire qui peuvent être obtenues en déformant le graphe sans rompre de liens entre ses sommets)* équivalentes sont possibles. La Figure 2 donne deux représentations graphiques équivalentes du graphe G_1 précédent.

6. En toute rigueur, un graphe orienté est muni d'un ensemble A de *couples*, ces derniers, par leur nature, induisant une orientation. Si $i \neq j$, les couples (i, j) et (j, i) sont différents. Un graphe non orienté est muni d'un ensemble de *paires*. Si $i \neq j$, les paires $\{i, j\}$ et $\{j, i\}$ sont identiques. Mais là encore, les sujets de concours ont l'habitude d'utiliser la notation des couples pour décrire aussi bien des graphes orientés que des graphes non orientés. Il faut donc être vigilant !



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

FIGURE 1. – Représentation graphique de G_1.

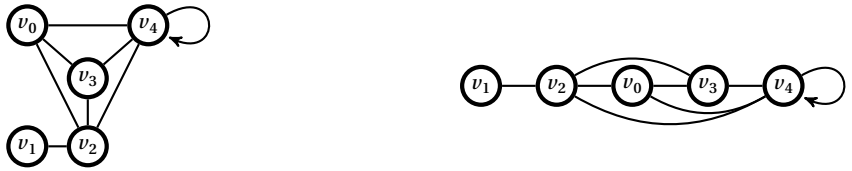
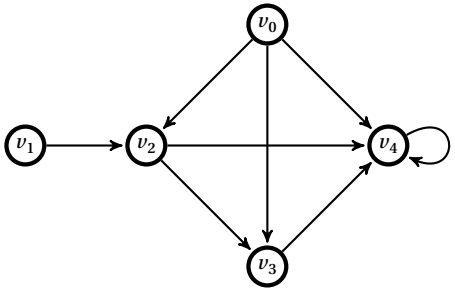


FIGURE 2. – Deux représentations topologiquement équivalentes de G_1.

EXEMPLE DE GRAPHE ORIENTÉ. Considérons le *graphe orienté* $G_2 = (S_2, A_2)$ d'ordre 5 obtenu à partir du graphe G_1 en orientant certaines arêtes. Certaines arêtes de G_1 sont maintenant absentes de l'ensemble des arêtes et A_2 ne contient que les couples de sommets des arêtes orientées.



$$
\begin{aligned}
G_2 &= (S_2, A_2)\\
S_2 &= \{v_0, v_1, v_2, v_3, v_4\}\\
A_2 &= \{(v_0, v_2), (v_0, v_3), (v_0, v_4), (v_1, v_2),\\
&\quad (v_2, v_3), (v_2, v_4), (v_3, v_4), (v_4, v_4)\}
\end{aligned}
$$

FIGURE 3. – Représentation du graphe orienté G_2.

Exercice 1 [Sol 1]

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

1.2. Degrés

Définition 2 | Degré, cas non orienté

Dans un graphe *non orienté*, on appelle *degré* d'un sommet s le nombre d'arêtes noté $d(s)$ dont ce sommet est une extrémité. Ce degré vaut 0 si le sommet est isolé (figure 4a).

Dans un graphe *orienté*, le sens de l'arête doit être pris en compte. C'est pourquoi on distingue le « degré sortant » du « degré entrant ».

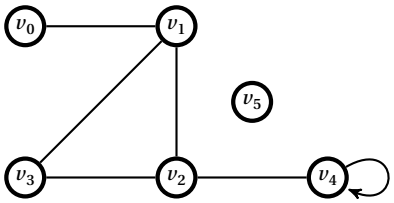
Définition 3 | Degré, cas orienté

Dans un graphe *orienté*, on appelle :

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

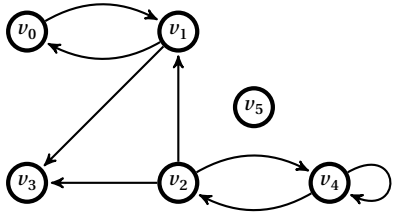
$$d(s) = d_+(s) + d_-(s).$$

Remarque 2 Un graphe non-orienté peut être vu comme un graphe orienté où chaque arête simple est remplacée par deux arêtes orientées. Le degré $d(s)$ (précédemment défini pour les graphes orientés pour chaque sommet s) correspond alors au double de $d(s)$ défini pour les graphes non orientés.



s	v_0	v_1	v_2	v_3	v_4	v_5
$d(s)$	1	3	3	2	3	0

(a)



s	v_0	v_1	v_2	v_3	v_4	v_5
$d_+(s)$	1	2	3	0	2	0
$d_-(s)$	1	2	1	2	2	0
$d(s)$	2	4	4	2	4	0

(b)

FIGURE 4. – Deux exemples de graphes

Exercice 2 [Sol 2] Trois pays envoient chacun à une conférence deux espions ; chaque espion doit espionner tous les espions des autres pays (mais pas son propre collègue !).

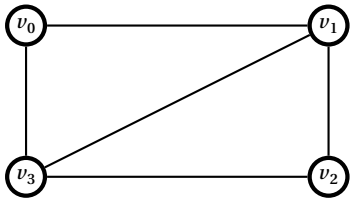
1. Représenter cette situation par un graphe orienté d'ordre 6 dans lequel la présence d'un arc du sommet i vers le sommet j signifie que i espionne j.

2. Calculer le nombre d'arcs de ce graphe ainsi que le degré de chaque sommet.

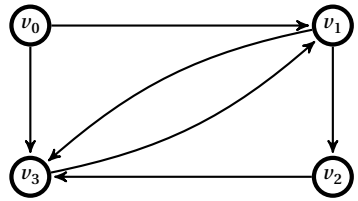
1.3. Chemin et cycle

Définition 4 | Chemin, cycle

- 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*.



(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.



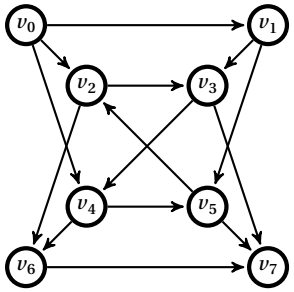
(v_0, v_1, v_2) et (v_0, v_3, v_1, v_2) sont des chemins du sommet v_0 vers le sommet v_2 de longueurs respectives 2 et 3. En revanche, il n'existe pas de chemin de v_2 vers v_0. Le chemin (v_1, v_2, v_3, v_1) est un cycle.

Exercice 3 [Sol 3] On considère le graphe orienté ci-dessous.

1. Pour tout entier $k \in [\![1, 5]\!]$, existe-t-il un chemin de longueur k reliant v_0 à v_7 ?

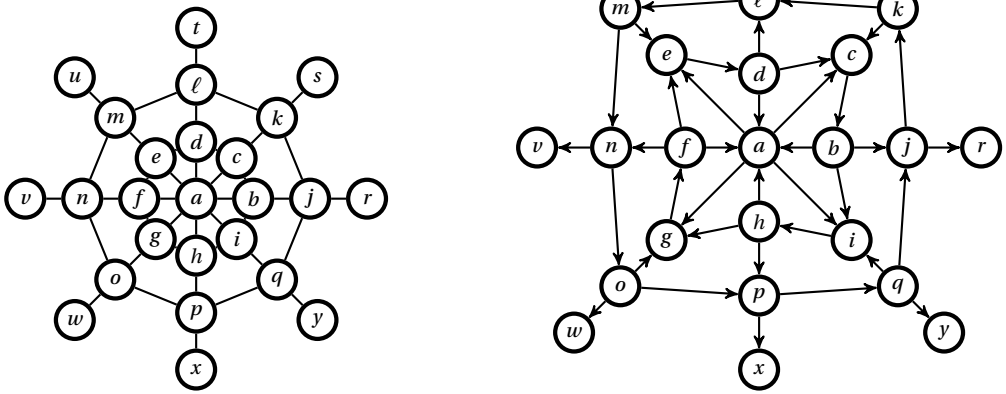
2. Peut-on trouver un chemin reliant v_6 à v_1 ?

3. Déterminer un cycle.



1.4. Longueur d'un chemin & Distance entre sommets

Considérons les deux graphes ci-dessous.



(a) Graphe G_1 non orienté et non pondéré (b) Graphe G_{1b} orienté et non pondéré

FIGURE 6. – Deux exemples de graphes

Dans la section précédente, on a défini un *chemin* dans un graphe comme une suite de sommets adjacents. Par exemple, $\gamma_1 = (a,b,j,r)$ ou $\gamma_2 = (a,e,d,\ell,k,j,r)$ sont deux chemins dans le graphe G_1 de la Figure 1. Ces deux chemins ont en commun d'avoir les mêmes extrémités. Ils permettent tous les deux de relier les sommets a et r. Mais leurs *longueurs*, exprimées en nombres d'arêtes, diffèrent. Notons les :

$$\delta(\gamma_1) = 3, \qquad \delta(\gamma_2) = 6.$$

La longueur d'un chemin se calcule de la même façon dans un *graphe orienté*, en comptant le nombre d'arcs qui définissent un chemin. On a par exemple dans le G_{1b} de la Figure 6b :

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

Définition 5 | Distance
La *distance* entre deux sommets d'un graphe est la longueur d'un plus court chemin qui relie ces sommets, en cas d'existence.

Sur le graphe G_1, la distance de a à r est 3. Il n'existe d'ailleurs qu'un seul chemin de a et r de distance 3 : le chemin $\gamma_1 = (a,b,j,r)$. Tous les autres chemins de a à r ont des longueurs au moins égales à 4.

Notation
Dans la suite, la distance d'un sommet u à un sommet v est notée $d_u[v]$. On note $\Gamma_u(v)$ l'ensemble des chemins de u à v.

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

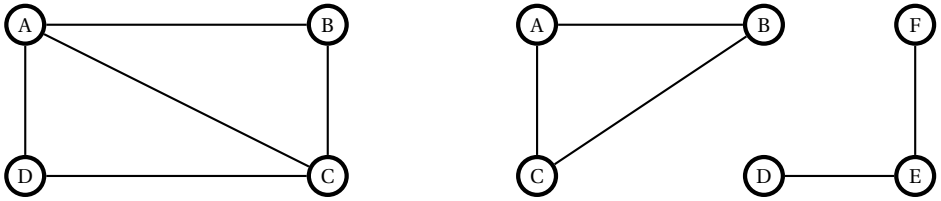
Dans la situation précédente, la détermination de $d_a[r]$ est relativement aisée et la topologie du graphe aide grandement à ce calcul! Se pose la question d'une méthode systématique de calcul de la distance entre deux sommets qui permettrait de résoudre ce problème de manière algorithmique (nous y répondrons dans les Chapitres (S2) 5 et (S2) 4 après avoir avoir appris à parcourir des graphes).

Remarque 3 La notion de distance telle que définie ci-dessous n'est pas exactement celle d'une distance au sens mathématique du terme.
- La propriété de séparation est vérifiée : $d_u[v] = 0 \iff u = v$.
- L'inégalité triangulaire est vérifiée : $d_u[v] \leqslant d_u[w] + d_w[v]$.
- La propriété de symétrie est bien vérifiée pour les graphes non-orientés, elle ne l'est pas toujours dans les graphes orientés. Par exemple dans le graphe de la Figure 6b, on a : $d_p[q] = 1$ et $d_q[p] = 3$.

1.5. Connexité

Définition 6 | Connexité
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.

Cela revient à dire que le graphe est « d'un seul tenant ».



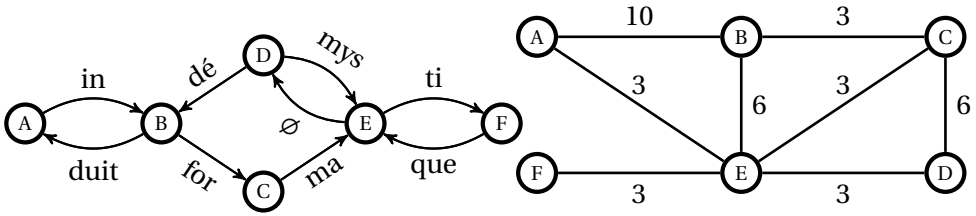
(a) Graphe connexe (b) Graphe non connexe

1.6. Étiquette et pondération

Une information (mot, lettre, symbole, chiffre, liste, image, etc) peut être associée à chaque sommet et/ou arête d'un graphe. On parle de *graphe étiqueté*.

Note | *En ce qui concerne le formalisme* G = (S,A)*, cela reviendrait à ajouter une deuxième coordonnée aux éléments de* S *et/ou une troisième coordonnées aux éléments de* A

- Si l'*étiquette d'un sommet* est unique, elle peut être confondue avec le sommet. On parle alors indifféremment du sommet A comme du sommet d'étiquette A.
- Si l'*étiquette d'une arête* est un nombre, elle définit une *pondération* de l'arête. Ce qui permet également la définition du *poids* ou du *coût* d'un chemin comme la somme des pondérations des arêtes. Un chapitre exploitera cette information pour rechercher un plus court chemin dans un graphe.



(a) Graphe orienté et étiqueté. (b) Graphe non orienté et pondéré.

2. IMPLÉMENTATIONS EN PYTHON

2.1. Notion d'implémentation

Le passage de la description formelle des graphes à leur mise en oeuvre informatique constitue une étape fondamentale appelée *implémentation*. Celle-ci mène à la construction d'une ou plusieurs *structures de données concrètes* qui définissent un *type* associé aux objets manipulés et des *fonctions de manipulation* de ces objets. Préalablement à cette phase de *concrétisation* dont les choix conditionnent très largement l'efficacité des traitements informatiques, une réflexion approfondie doit être menée pour exprimer et anticiper les besoins répondant à des objectifs attendus.

De manière schématique, cette étape formelle doit mener à la définition d'un *type abstrait de données*, sorte de cahier des charges qui spécifie de manière rigoureuse la nature des objets manipulés et les traitements à y apporter, indépendamment de toute considération informatique. C'est la façon dont les *données* sont *stockées* et *organisées en mémoire* et la façon dont elles sont traitées par des *fonctions de manipulation* qui doivent être détaillées. Les *implémentations* ne sont alors que des déclinaisons informatiques possibles du type abstrait de données, répondant chacune, par leurs caractéristiques intrinsèques, spécifiquement à certains besoins. On les appelle alors des *structures de données concrètes*.

Insistons sur la possibilité d'implémenter *plusieurs* structures de données associées à un seul et même type abstrait. Par exemple, définir un type abstrait de données associé à un *ensemble* de données peut mener à la définition des types *listes*[7] et *tableaux* et à leurs fonctions de manipulation associées. Même si ces deux implémentations présentent de nombreux points communs, elles diffèrent de manière essentielle sur certains de leurs aspects. Par exemple, un tableau est une structure de données *statique* dans le sens où sa taille est figée au moment de sa création alors qu'une liste est une structure de données *dynamique*, son nombre d'éléments pouvant évoluer au gré des besoins. Selon les circonstances, cette distinction est un avantage ou un inconvénient. La question se pose bien évidemment de construire une structure de données qui n'offre que des avantages. Ce n'est pas toujours possible. L'informaticien doit donc faire des choix.

2.2. Implémentations des graphes

Il existe de nombreuses manières d'implémenter les graphes dans un langage de programmation. Avant toute implémentation, il convient de préciser leur description informatique sous forme d'objets typés; de définir et construire les fonctions qui agissent sur ces objets en vue de réaliser certains traitements spécifiques.

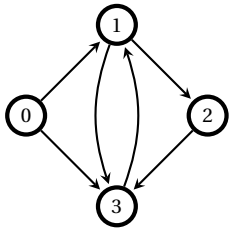
Conformément au programme, ce chapitre présente deux implémentations, l'une par *matrices d'adjacence*, l'autre par *dictionnaire d'adjacence*, ces dernières pouvant

7. Dans le cas du langage Python, les listes Python diffèrent des listes traditionnellement définies dans les autres langages de programmation. Elles devraient plus logiquement être appelées *tableau dynamique*. Mais conformément au programme, cette distinction n'est pas faite dans ce cours.

être déclinées sous plusieurs formes comme nous le verrons. L'exposé s'attache à définir certaines de leurs fonctions de manipulation et compare leur efficacité. Ces implémentations s'appuient sur des *types* pré-existants en Python : listes, tableaux, dictionnaires. Les *fonctions de manipulations* sont alors construites pour s'adapter à ces choix même si toutes répondent à un même objectif, indépendamment de leur implémentation. Par exemple, une fois l'organisation des données d'un graphe parfaitement définie et implémentée, les fonctions de manipulation devront toutes au moins permettre l'*ajout* ou la *suppression* de noeuds et d'arêtes d'un graphe, la *modification* de certaines étiquettes, la détermination de la *taille* d'un graphe, etc. D'autres fonctions devront permettre le *parcours* d'un graphe, c'est-à-dire l'accès d'une donnée à une autre en tenant compte de leurs liens. Les *parcours* de graphes font l'objet d'un deuxième chapitre sur les graphes. Enfin, certains algorithmes sur les graphes doivent permettre de répondre à des problèmes spécifiques comme la recherche d'un *plus court chemin* dans un graphe ou la recherche de *composantes connexes*. Ces points sont abordés dans un troisième chapitre.

2.3. Cas non pondéré : matrice d'adjacence

On considère dans cette partie un graphe *orienté* formé de N noeuds, et on suppose ici que ces noeuds sont désignés par des entiers consécutifs de 0 à N – 1. On peut représenter ce graphe par une matrice de N lignes et N colonnes, et dont chaque case de ligne i et colonne j contient un booléen égal à `True` si le graphe possède un arc du noeud i vers le noeud j, et `False` sinon. Notons que l'on peut aussi faire le choix de remplir la matrice par les entiers 1 et 0, où 1 code pour `True` et 0 code pour `False`. Une telle matrice est appelée *matrice d'adjacence*.



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

En pratique, cette matrice peut être implémentée à l'aide d'un tableau bidimensionnel du module `numpy`, la fonction `np.array` permettant de construire un tableau à partir de la liste de ses lignes. Par exemple, la matrice précédente pourra être obtenue par la commande :

```
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 suivante.

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

La fonction `zeros` construit un tableau rempli de 0 de taille spécifiée en premier paramètre – tuple donnant le nombre de lignes puis de colonnes – et le deuxième paramètre indique que le tableau est rempli de booléens – la valeur 0 étant alors convertie en le booléen `False`.

Remarque 4 On peut représenter un graphe pondéré aussi à l'aide d'une matrice ; on indique simplement les poids dans le tableau, en lieu et place des booléens (et $+\infty$ en cas d'absence d'arête). On parle alors de *matrice des poids*, mais cette notion est hors-programme.

Exercice 4 [Sol 4] La variable G est supposée contenir 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) :

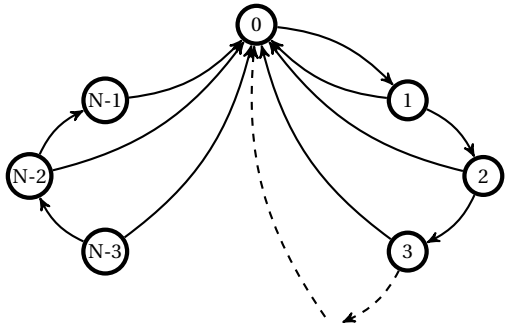


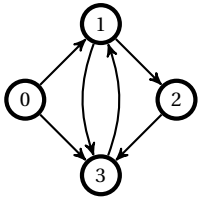
FIGURE 9. – Graphe construit par `un_graphe(N)`.

AVANTAGES ET INCONVÉNIENTS DE L'IMPLÉMENTATION Représenter un graphe par une matrice d'adjacence a l'avantage de la simplicité. Cependant, une telle implémentation occupe un espace mémoire proportionnel à N^2. Dans le cas où le graphe est peu dense, c'est-à-dire lorsqu'il comporte peu d'arcs par rapport au nombre de noeuds, ce coût spatial n'est pas toujours pertinent. Dès lors, une solution occupant un espace mémoire proportionnel au nombre d'arcs peut être plus avantageuse.

2.4. Cas pondéré ou non : dictionnaire d'adjacence

Dans cette partie, nous allons stocker le graphe en utilisant un dictionnaire, dont les clés sont les noeuds du graphe, et la valeur associée est la liste de ses noeuds voisins.

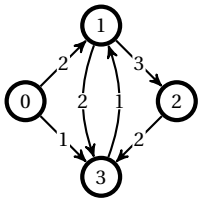
Voici pour le graphe de la partie précédente son dictionnaire d'adjacence :




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

Remarque 5 (Noeuds entiers numérotés à partir de 0) Notons que, dans le cas où les noeuds sont des entiers successifs à partir de 0, on peut choisir de stocker ces informations dans une liste au lieu d'un dictionnaire. La liste d'adjacence est alors une liste de listes, l'élément d'indice i contenant là encore la liste des noeuds voisins du noeud i. Par exemple, le graphe précédent aurait été stocké par la liste d'adjacence suivante : `[[1,3], [2,3], [3], [1]]`.

Considérons maintenant plutôt un graphe pondéré. Dans ce cas, plutôt que d'introduire une liste de sommets en valeur du dictionnaire, on indiquera à la place une liste de couples contenant le numéro de sommet voisin, ainsi que son poids. Par exemple,



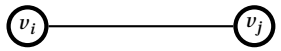

```
0 : [(1, 2), (3, 1)], 1 : [(2, 3), (3, 2))], 2 : [(3,
2)], 3 : [(1, 1)]
```

Exercice 5 [Sol 5] La variable G est maintenant supposée contenir le dictionnaire d'adjacence d'un graphe non pondéré. Écrire les versions correspondantes des fonctions `nb_arcs`, `voisins` et `un_graphe` de l'exercice précédent.

La représentation d'un graphe par un dictionnaire d'adjacence, moins gourmande en espace mémoire, est généralement privilégiée (sauf dans le cas particulier de graphes possédant de très nombreux arcs, où l'utilisation d'une matrice d'adjacente peut s'avérer plus pertinente). En outre, même si c'est plus anecdotique, on notera qu'avec la solution d'un dictionnaire d'adjacence, les sommets du graphe n'ont plus besoin d'être désignés par des entiers (mais peuvent par exemple être stockés en tant que chaînes de caractère).

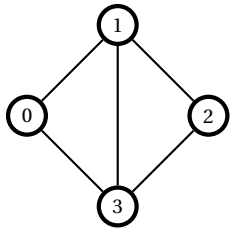
2.5. Cas d'un graphe non orienté

Pour un graphe non orienté, on utilise une représentation de graphe orienté (sous forme de matrice ou dictionnaire d'adjacence) de telle sorte que chacune des arêtes non orientée soit représentée par *deux* arcs, un pour chacune des orientations. Autrement dit, l'arête non orientée :



est représenté par les deux arcs orientés :

Voici un graphe non orienté, son dictionnaire d'adjacence, et sa matrice d'adjacence.



De matrice d'adjacence :

	0	1	2	3
0	False	True	False	True
1	True	False	True	True
2	False	False	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]}`.

Notons que dans le cas d'un graphe non orienté, la matrice d'adjacence est symétrique. En résumé,

représentation par MATRICE D'ADJACENCE = une matrice de taille le nombre de sommets indiquant la présence d'arêtes + (un dictionnaire d'étiquetage, si nécessaire)

représentation par DICTIONNAIRE DES VOISINS = un dictionnaire de { CLEFS : les sommets $s \in S$; VALEURS : les voisins de s }

2.6. Un peu de coloriage

Terminons cette partie en présentant un algorithme classique relatif au *coloriage* des graphes (l'algorithme que nous allons exposer sera programmé en TP).

On considère ici un graphe non orienté (associé à par exemple à une carte[8] de régions), pour lequel on veut associer à chacun de ses sommets une couleur de telle sorte que deux sommets voisins aient une couleur différente, et en tâchant d'utiliser 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 quatre couleurs au plus, mais il n'existe pas d'algorithme efficace permettant d'obtenir un tel coloriage optimal à tous les coups.

8. il est difficile de décrire l'ensemble des graphes tels que le théorème des 4 couleurs s'applique

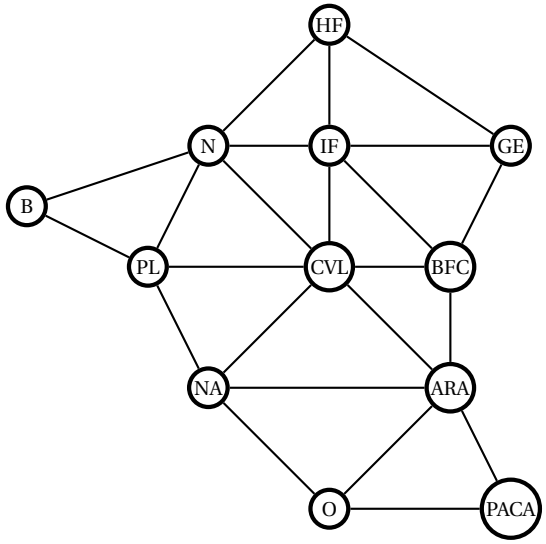


FIGURE 10. – Graphe des régions de France métropolitaine continentale adjacentes.

Voici néanmoins un algorithme conduisant généralement à une bonne solution, même si celle-ci n'est pas toujours optimale. Il repose sur un principe d'*algorithme glouton* de la manière suivante :

- les couleurs sont représentées par des entiers successifs à partir de 0 ;
- on décrit un à un tous les sommets (dans l'ordre que l'on souhaite), et on associe au sommet courant la plus petite couleur non déjà utilisée pour ses sommets voisins.

Illustrons son principe sur le graphe suivant, représentant les régions de France métropolitaine continentale adjacentes (HF désigne les hauts-de-France, N la Normandie, IF l'île-de-France, etc) :

Utilisons l'algorithme dans un sens de parcours nord-sud et ouest-est (HF, N, IF, GE, B, etc) :

- HF n'a aucun voisin colorié, on lui affecte la couleur 0 ;
- N a pour seul voisin colorié HF avec la couleur 0, on lui affecte la couleur 1 ;
- IF a pour voisins coloriés HF avec la couleur 0 et N avec la couleur 1, on lui affecte la couleur 2 ;
- GE a pour voisins coloriés HF avec la couleur 0 et IF avec la couleur 2, on lui affecte la couleur 1 ;
- B a pour seul voisin colorié N avec la couleur 1, on lui affecte la couleur 0 ;

et ainsi de suite, jusqu'à obtenir le coloriage suivant :

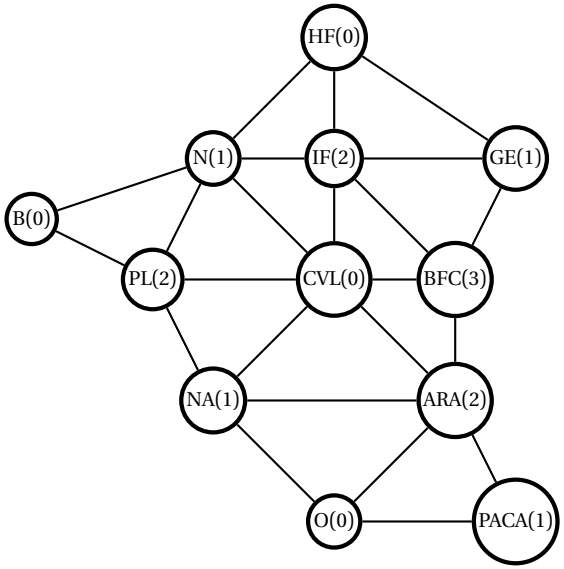


FIGURE 11. – Le même graphe colorié pour un sens de parcours nord-sud et ouest-est.

Notons que dans ce cas une solution optimale à quatre couleurs a bien été trouvée par l'algorithme. Notons aussi que, bien sûr, le coloriage obtenu dépend de l'ordre de parcours choisi sur les sommets, et que le nombre de couleurs utilisées peut alors être différent. Par exemple, si les sommets du même graphe sont parcourus dans l'ordre ARA, BFC, B, CVL, GE, HF, IF, N, NA, O, PL, PACA, vous vérifierez que l'on obtient le nouveau coloriage suivant (où cinq couleurs ont dû être utilisées!).

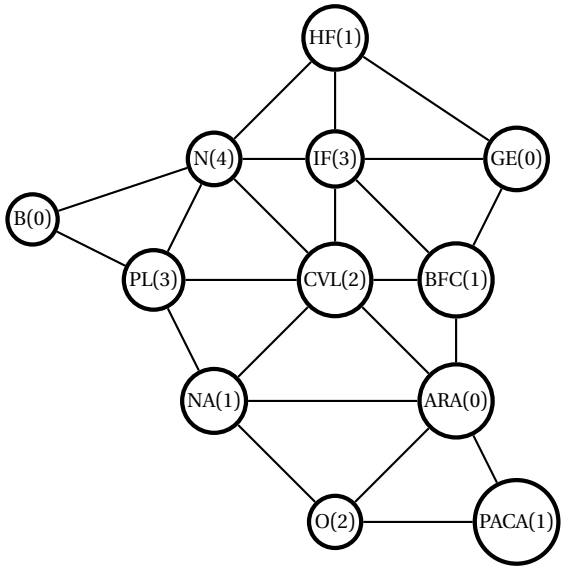
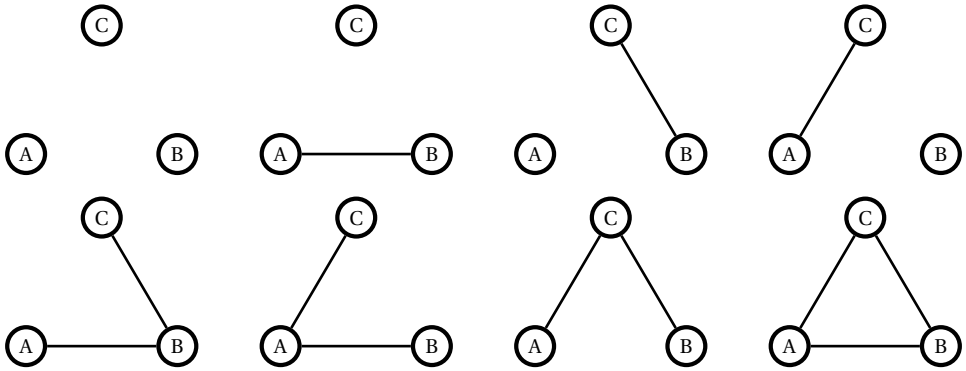


FIGURE 12. – Le même graphe colorié pour un autre sens de parcours.

SOLUTIONS DES EXERCICES

Solution 1

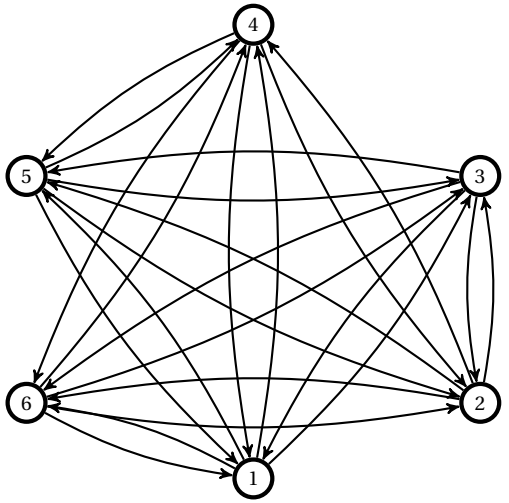
1. Il y en a exactement 8 :



2. Chaque couple (u, v) de sommets peut ne pas être relié par un arc, être relié par une 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.

Solution 2

1. 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 :



2. 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.

Solution 3

1. Il n'y a pas d'arc reliant v_0 à v_7, donc, ce n'est pas possible pour $k = 1$.

Il n'y a pas non plus de chemin de longueur 2 reliant v_0 à v_7.

Le chemin (v_0, v_1, v_3, v_7) est de longueur 3.

Il n'existe pas de chemin de longueur 4 reliant v_0 à v_7 car il n'existe pas de chemin de longueur 2 reliant v_1, v_2 ou v_4 à v_3, v_5 ou v_6.

Le chemin $(v_0, v_2, v_3, v_4, v_5, v_7)$ est de longueur 5 reliant v_0 à v_7.

2. La réponse est non : en effet, un tel chemin possèderait nécessairement un arc dont v_0 serait une extrémité, ce qui n'est pas le cas.

3. $(v_2, v_3, v_4, v_5, v_2)$ est un cycle.

Solution 4

1.
```python
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
```

2.
```python
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
```

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
```

Solution 5

1.
```python
def nb_arcs(G : np.array)->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
```

2.
```python
def voisins(G : np.array , 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]
```

3. Une solution itérative :

```python
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
```

et 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
```