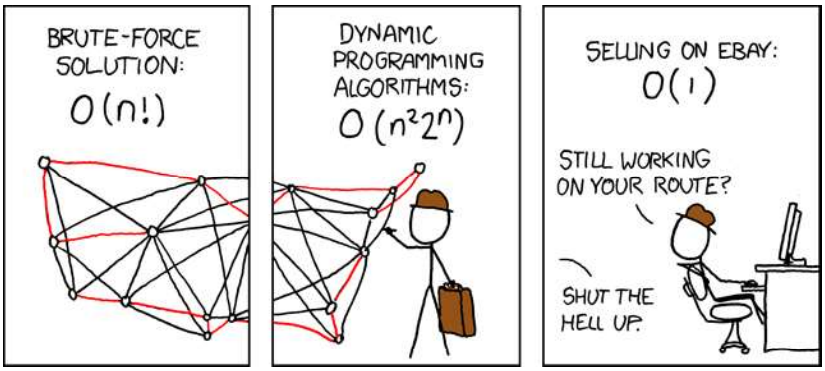
Chapitre 8

Graphe - Informatique Commune



8.1 Graphe

8.1.1 Graphe non orienté

Définition 8.1.1

On appelle *graphe non orienté* tout couple $G := (S, A)$ où
- S est un ensemble fini non vide dont les éléments sont appelés *noeuds*, ou *sommets*.
- A est un ensemble de paires $\{x, y\}$, où x et y sont deux éléments distincts de S. Ces paires sont appelées *arêtes*, ou *arcs*.

Remarques

⇨ En pratique, l'arête $\{x, y\}$ sera notée $x - y$ ou $y - x$. On utilisera aussi les notations xy ou yx. Intuitivement, une arête $x - y$ permet de passer du sommet x au sommet y et du sommet y au sommet x. Deux sommets reliés par une arête sont dits *adjacents*. On appelle *voisin* d'un sommet x tout sommet adjacent à x.

⇨ Dans notre définition, nous nous sommes interdit les *boucles*, c'est-à-dire les arêtes reliant un sommet à lui-même. Nous n'autorisons pas non plus le fait d'avoir plusieurs arêtes entre deux sommets. Les graphes s'autorisant de telles arêtes sont appelés *multigraphes*.

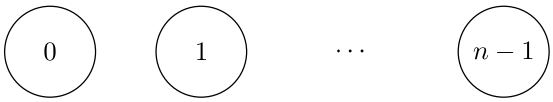
⇨ Dans un graphe non orienté

$$|A| \leqslant \frac{|S|\,(|S|-1)}{2}.$$

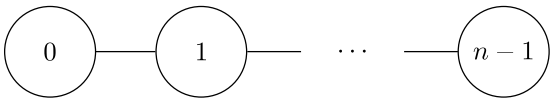
⇨ Lorsqu'on parle d'un graphe à n sommets sans préciser S, on prend $S := [\![0, n[\![$.

Exemples

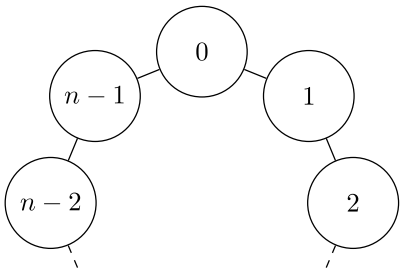
⇨ Le graphe *entièrement déconnecté* possède n sommets et aucune arête.



⇨ Le graphe *chemin* à n sommets $\mathcal{K}_n$ possède une arête entre i et $j \in [\![0, n[\![$ si et seulement si $j = i + 1$.

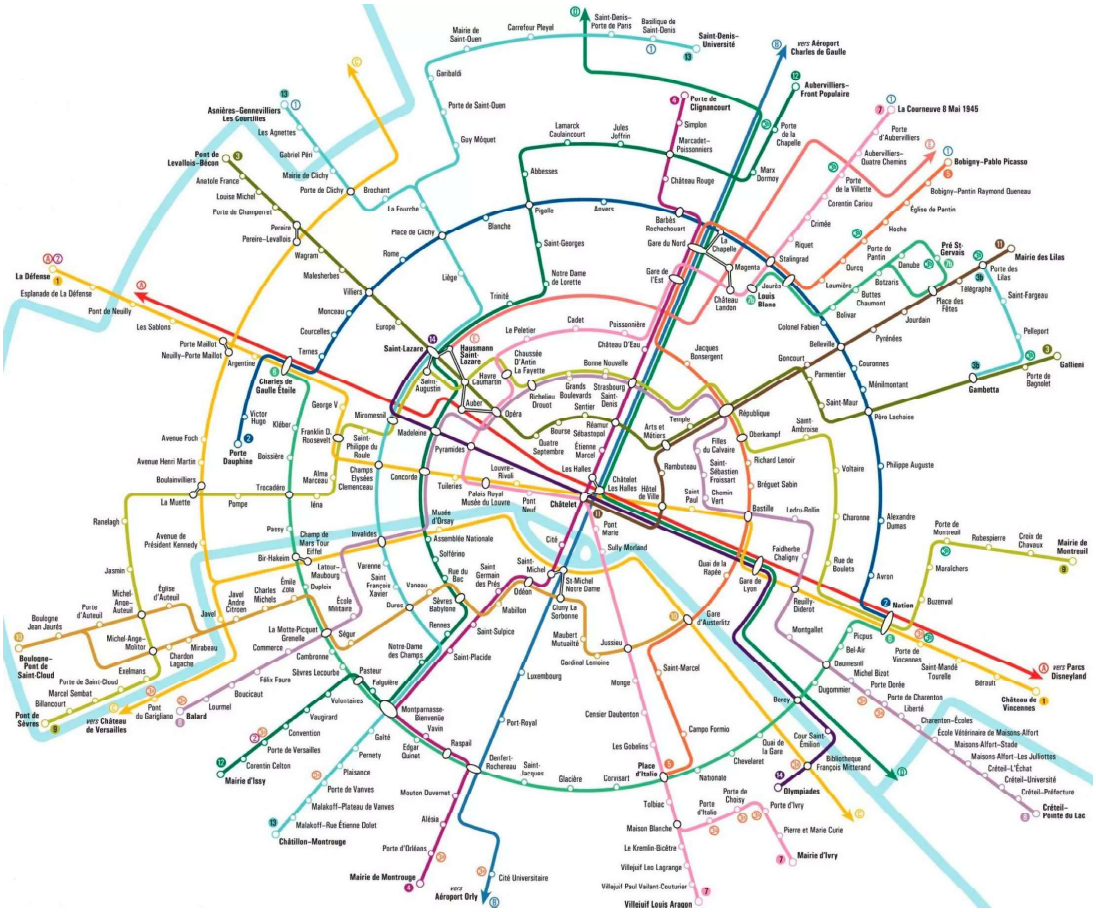


⇨ Le graphe *cycle* à n sommets $\mathcal{C}_n$ (pour $n \geqslant 3$) possède une arête entre i et $j \in [\![0, n[\![$ si et seulement si $j \equiv i+1\ [n]$.



⇨ Le graphe des utilisateurs de Facebook a un sommet pour chaque utilisateur et une arête entre deux sommets lorsque deux utilisateurs sont *amis*. Notons que $|S|$ est de l'ordre de 10^9 et que $|A|$ est de l'ordre de 10^{11}, ce qui pose quelques difficultés algorithmiques.

⇨ Dans le graphe du métro parisien, les sommets représentent les stations de métro et les arêtes, les liaisons entre ces stations.



Définition 8.1.2

On appelle *degré* d'un sommet x le nombre d'arêtes de la forme $x - y$.

Remarque

⇨ Le degré d'un sommet est son nombre de voisins.

Définition 8.1.3

On appelle *chemin* de longueur n toute suite $c \coloneqq z_0, z_1, \ldots, z_n$ de $n+1$ sommets telle que

$$\forall k \in [\![0, n[\![, \quad z_k - z_{k+1} \in A.$$

Les sommets z_0 et z_n sont appelés *extrémités* du chemin et on dit que c *relie* z_0 à z_n.

Remarques

⇨ On peut aussi voir un chemin de longueur n comme une suite de n arêtes consécutives. Un chemin de longueur n est constitué de $n+1$ sommets et de n arêtes.

⇨ On accepte les chemins de longueur nulle, reliant un sommet à lui-même, sans arête.

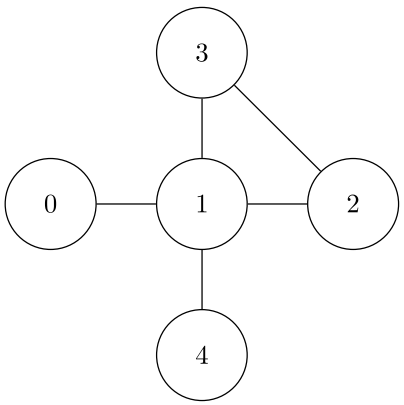
⇨ Dans la suite, on notera $l(c)$ la longueur d'un chemin c.

Définition 8.1.4

Un chemin $z_0, z_1, \ldots, z_n$ est dit
- *élémentaire* lorsqu'il ne passe pas deux fois par le même sommet, c'est-à-dire lorsque les sommets sont deux à deux distincts.
- *simple* s'il ne passe pas deux fois par la même arête, c'est-à-dire lorsque les arêtes $z_k - z_{k+1}$ sont deux à deux distinctes.

Remarque

⇨ Tout chemin élémentaire est simple. Cependant la réciproque est fausse puisque sur le graphe non orienté ci-dessous, le chemin $0 - 1 - 2 - 3 - 1 - 4$ est simple, mais pas élémentaire.



Définition 8.1.5

Un sommet y est dit *accessible* depuis un sommet x lorsqu'il existe au moins un chemin reliant x à y.

Remarque

⇨ Puisqu'on accepte les chemins de longueur nulle, tout sommet est accessible depuis lui-même.

Proposition 8.1.6

Dans un graphe non orienté, la relation d'accessibilité est une relation d'équivalence sur S.

Remarques

⇨ Si y est accessible depuis x, alors x est accessible depuis y. On dit alors que x et y sont *connectés*.

⇨ On appelle *composante connexe* toute classe d'équivalence pour cette relation.

Définition 8.1.7

On dit qu'un graphe non orienté est *connexe* lorsqu'il ne possède qu'une seule composante connexe, c'est-à-dire lorsque tous ses sommets sont connectés.

Définition 8.1.8

Si y est un sommet accessible depuis un sommet x, on appelle distance entre x et y l'entier

$$\mathrm{d}(x,y) \coloneqq \inf\{l(c) : c \text{ est un chemin de } x \text{ à } y\}.$$

Un *chemin de longueur minimale* de x à y est un chemin c de x à y tel que $l(c) = \mathrm{d}(x,y)$.

Remarques

- Lorsque y n'est pas accessible depuis x, la convention est de poser $\mathrm{d}(x,y) \coloneqq +\infty$.
- Conformément à ce qu'on attend d'une distance

$$\begin{aligned} \forall x \in S, &\quad \mathrm{d}(x,x) = 0, \\ \forall x,y \in S, &\quad \mathrm{d}(y,x) = \mathrm{d}(x,y), \\ \forall x,y,z \in S, &\quad \mathrm{d}(x,z) \leqslant \mathrm{d}(x,y) + \mathrm{d}(y,z). \end{aligned}$$

Définition 8.1.9

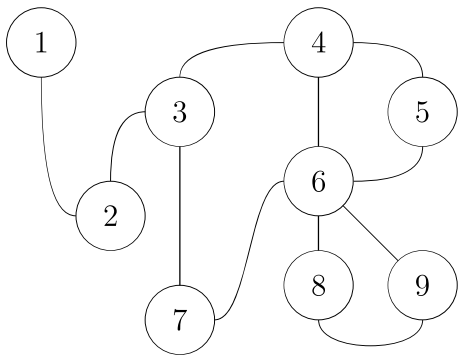
On appelle *cycle* tout chemin simple de longueur non nulle dont les deux extrémités sont identiques.

Remarques

- Dans la définition, il est nécessaire de se limiter aux chemins simples, sinon on pourrait construire des « cycles » dans les graphes en parcourant une même arête dans un sens puis dans l'autre.
- Dans un graphe non orienté, la longueur d'un cycle est supérieure ou égale à 3.
- Un cycle sera dit *élémentaire* lorsque la seule répétition de sommets est celle de ses extrémités. Un cycle est élémentaire si et seulement si il ne contient pas d'autre cycle.
- On dit qu'un graphe est *acyclique* lorsqu'il ne possède pas de cycle.

Exemple

- Dans le graphe suivant, le chemin $4-5-6-9-8-6-4$ est un cycle. Ce graphe possède 4 cycles élémentaires.

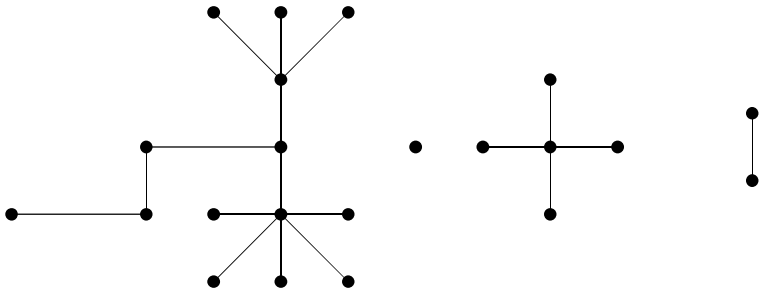


Définition 8.1.10

On appelle *arbre* tout graphe connexe acyclique.

Remarques

⇨ Les graphes suivants sont des arbres.



⇨ Les arbres que nous avons manipulés dans les chapitres précédents sont ce qu'on appelle des *arbres enracinés*. Ce sont des arbres pour lesquels on a choisi un sommet appelé *racine*. La distance d'un sommet à la racine est appelée *profondeur*. Ces arbres sont conventionnellement dessinés de façon à ce que les sommets de même profondeur soient à même hauteur.

Exercice 1

⇨ En choisissant successivement trois racines pour l'arbre de gauche ci-dessus, dessiner l'arbre enraciné ainsi obtenu de manière conventionnelle.

8.1.2 Graphe orienté

Définition 8.1.11

On appelle *graphe orienté* tout couple $G := (S, A)$ où
- S est un ensemble fini non vide dont les éléments sont appelés *sommets*.
- A est un ensemble de couples (x, y), où x et y sont deux éléments distincts de S. Ces paires sont appelées *arcs*.

Remarques

⇨ En pratique, l'arc (x, y) sera noté $x \to y$ ou xy. Intuitivement, un arc $x \to y$ permet de passer du sommet x au sommet y mais pas du sommet y au sommet x. S'il y a un arc $x \to y$, on dit que x est un *prédécesseur* de y et que y est un *successeur* de x.

⇨ Dans un graphe orienté

$$|A| \leqslant |S|\,(|S| - 1)\,.$$

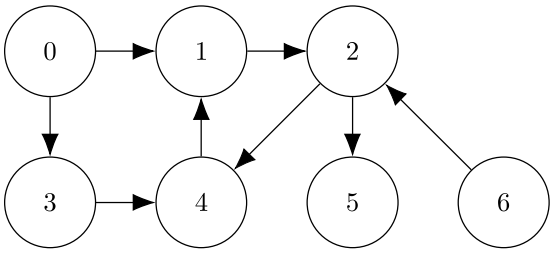
⇨ En pratique, on confondra souvent un graphe non orienté $G := (S, A)$ avec son graphe orienté associé G_{o}. Ce dernier possède les mêmes sommets que G. De plus, $x \to y$ est un arc de G_{o} si et seulement si $x - y \in A$. En particulier, dans G_{o}, dès que $x \to y$ est un arc, $y \to x$ en est un autre.

⇨ Les arbres enracinés s'orientent naturellement depuis leur racine : lorsqu'on les dessine de manière conventionnelle, on les oriente du haut vers le bas.

⇨ À un graphe orienté $G := (S, A)$, on associe le graphe non orienté G_{no} obtenu en « oubliant » l'orientation des arcs.

Exemples

⇨ Voici un exemple de graphe orienté à 7 sommets. C'est sur cet exemple que nous détaillerons l'exécution de nos algorithmes dans la seconde partie de ce chapitre.



⇨ Le graphe du web possède un sommet pour chaque page web et un arc de x vers y lorsque la page x contient un lien vers la page y. C'est ce graphe que les moteurs de recherche parcourent pour construire leur index. La taille du graphe du web est inconnue mais Google indexe plus de 50 milliards de pages.

⇨ Le graphe des utilisateurs d'Instagram a un sommet pour chaque utilisateur et un arc de x vers y lorsque x est un *follower* de y. Contrairement au graphe des utilisateurs de Facebook qui est non orienté, celui d'Instagram l'est.

Définition 8.1.12

Dans un graphe orienté, on appelle
- *degré entrant* d'un sommet x, le nombre d'arcs de la forme $y \to x$.
- *degré sortant* d'un sommet x, le nombre d'arcs de la forme $x \to y$.
- *degré total* d'un sommet, la somme de son degré entrant et de son degré sortant.

Remarque

⇨ Le degré entrant d'un sommet est son nombre de prédécesseurs et le degré sortant, son nombre de successeurs.

Définition 8.1.13

On appelle chemin de longueur n toute suite $c := z_0, z_1, \dots, z_n$ de $n+1$ sommets telle que

$$\forall k \in [\![0, n[\![\,, \quad z_k \to z_{k+1} \in A.$$

Les sommets z_0 et z_n sont appelés *extrémités* du chemin et on dit que c *relie* z_0 à z_n.

Remarques

⇨ Comme dans les graphes non orientés, on définit la notion de chemin *élémentaire* et de chemin *simple*. Les chemins élémentaires sont simples, mais la réciproque est fausse.

⇨ La notion d'*accessibilité* se définit aussi de la même manière. Cependant, dans un graphe orienté, ce n'est plus une relation d'équivalence sur S. Les notions de connexité et de composante connexe n'ont plus de sens pour ces graphes.

⇨ La notion de distance entre un sommet x et un sommet y est toujours définie. L'inégalité triangulaire reste vraie, mais la distance n'est plus symétrique.

⇨ La notion de cycle se définit toujours de la même manière dans un graphe orienté. Cependant, contrairement à ce qui se passe dans le cas des graphes non orientés, il existe des cycles de longueur 2.

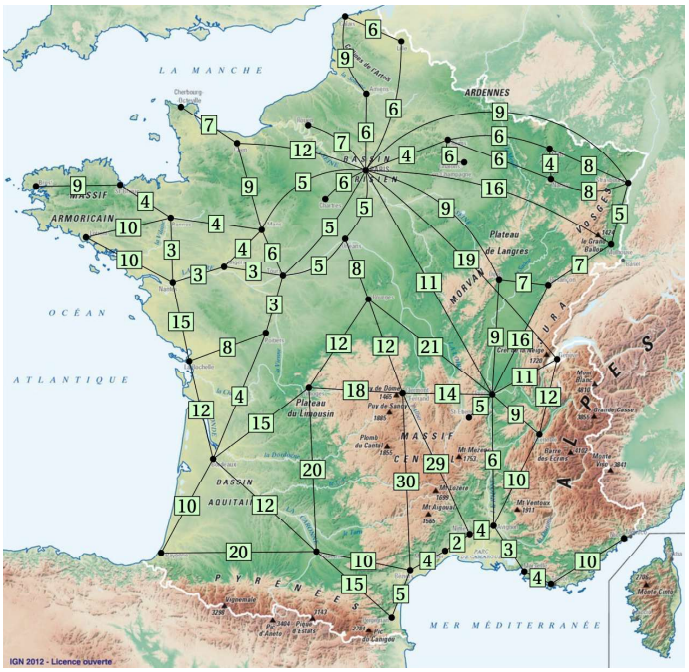
8.1.3 Graphe pondéré

Définition 8.1.14

On appelle *graphe pondéré* la donnée d'un graphe $G := (S, A)$ et d'une application $\rho : A \to \mathbb{R}_+$ appelée *poids*.

Exemple

⇨ Voici le graphe non orienté des connexions ferroviaires françaises, le poids représentant les temps de trajet en dizaines de minutes.



Remarques

⇨ Un graphe pondéré peut être orienté ou non orienté.

⇨ Il est possible de considérer des graphes pondérés avec des fonctions de poids prenant des valeurs négatives. Mais dans ce cours, puisque c'est une condition pour pouvoir appliquer l'algorithme de Dijkstra, nous nous limiterons à des fonctions de poids positives.

⇨ On appelle *poids du chemin* $c \coloneqq z_0, z_1, \dots, z_n$ le réel positif

$$\rho(c) \coloneqq \sum_{k=0}^{n-1} \rho(z_k z_{k+1}).$$

⇨ On appelle *poids d'un graphe* la somme des poids de ses arêtes.

Définition 8.1.15

Si y est un sommet accessible depuis un sommet x, on définit

$$\delta(x, y) \coloneqq \inf \{\rho(c) : c \text{ est un chemin de } x \text{ à } y\}.$$

Un *chemin de poids minimal* de x à y est un chemin c de x à y tel que $\rho(c) = \delta(x, y)$.

Remarque

⇨ Dans le cas où les poids sont des distances, par exemple si G est le graphe d'un réseau routier, on pourra parler de distance et de plus court chemin. Il ne faut cependant pas confondre le réel $\delta(x, y)$ avec l'entier $\mathrm{d}(x, y)$ représentant le nombre minimal d'arcs entre x et y.

8.1.4 Représentation d'un graphe

Définition 8.1.16

Soit $G \coloneqq (S, A)$ un graphe où $S = [\![0, n[\![$. On appelle matrice d'adjacence la matrice $M \in \mathcal{M}_n(\mathbb{Z})$ définie par

$$\forall i, j \in [\![0, n[\![, \quad m_{i,j} \coloneqq \begin{cases} 1 & \text{si } j \text{ est un successeur de } i, \\ 0 & \text{sinon.} \end{cases}$$

Remarques

⇨ Un graphe est « non orienté » si et seulement si sa matrice d'adjacence est symétrique.

⇨ Puisqu'on interdit les boucles, une matrice d'adjacence n'a que des 0 sur la diagonale.

⇨ On peut représenter un graphe pondéré par une matrice d'adjacence $M \in \mathcal{M}_n(\mathbb{R})$. On prend pour coefficient $m_{i,j}$ la valeur `None` lorsqu'il n'existe pas d'arc de i à j, et le poids $\rho_{i,j}$ de l'arc allant de i à j lorsqu'un tel arc existe.

Définition 8.1.17

Soit $G \coloneqq (S, A)$ un graphe où $S = [\![0, n[\![$. On appelle liste d'adjacence le tableau g de longueur n tel que pour tout $i \in [\![0, n[\![$, g_i est la liste des successeurs $j \in [\![0, n[\![$ de i.

Remarque

⇨ On peut représenter un graphe pondéré par une liste d'adjacence dans laquelle, pour tout sommet i, g_i est la liste des couples $(\rho_{i,j}, j)$ où j est un successeur de i.

⇨ Dans la suite de ce cours, nous utiliserons des listes d'adjacence pour stocker les graphes pondérés.

8.2 Algorithmes sur les graphes

8.2.1 Parcours générique d'un graphe

Supposons que vous êtes enfermé dans un labyrinthe de salles, connectées entre elles par des portes. Nous représentons ce labyrinthe par un graphe dont les sommets sont les salles et les arêtes sont les portes reliant ces salles entre elles. Si l'on souhaite sortir de ce labyrinthe, un réflexe naturel est d'emprunter au hasard les portes que l'on croise. Cependant, cette stratégie possède deux défauts importants : elle ne nous dit pas ce qu'on doit faire lorsqu'on tombe

dans un cul-de-sac et elle ne nous empêche pas de tourner en rond.

Pour résoudre ces deux problèmes, la solution la plus simple est de marquer les salles. Au cours de notre exploration, nous choisirons donc de les placer successivement dans 3 états différents :

— *Inconnu* : C'est l'état dans lequel est une salle qui n'a pas encore été découverte.
— *Découvert* : C'est l'état dans lequel on place une salle lorsqu'on l'a aperçue par une porte.
— *Visité* : C'est l'état dans lequel est une salle dans laquelle nous sommes déjà entrés.

Pour ne pas tourner en rond, il suffit de ne pas entrer dans une salle qui a déjà été visitée. Pour ces salles, on peut choisir de marquer leur sol d'une croix blanche. Et pour savoir que faire lorsqu'on est dans un cul-de-sac, il suffit de garder une trace des salles que l'on a découvertes, mais qui n'ont pas encore été visitées. Pour cela, on conserve avec nous un sac contenant une marque pour chacune d'elles.

Revenons au vocabulaire des graphes. À l'aide de ces deux outils, notre sac ainsi que le marquage des sommets, nous sommes armés pour parcourir l'ensemble des sommets accessibles depuis notre sommet de départ. Ce sommet est appelé *source*. Pour cela, il nous suffit de suivre l'algorithme suivant, que nous appelons « *parcours générique* ».

Algorithme Parcours générique

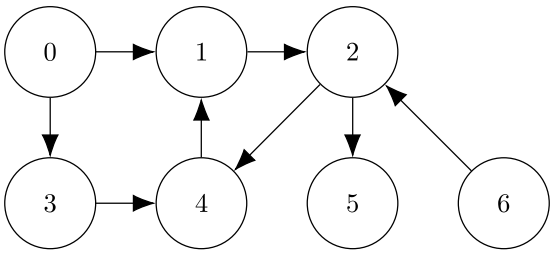
```
mettre le sommet source dans le sac
tant que le sac n'est pas vide faire
    prendre un sommet x dans le sac
    si x n'a pas été visité alors
        marquer le sommet x comme visité
        pour chaque arc xy faire
            mettre le sommet y dans le sac
```

La seule propriété dont notre sac a besoin est qu'on puisse y mettre des sommets pour les extraire plus tard. L'ordre dans lesquels ces sommets sont extraits n'a pas d'importance pour le moment.

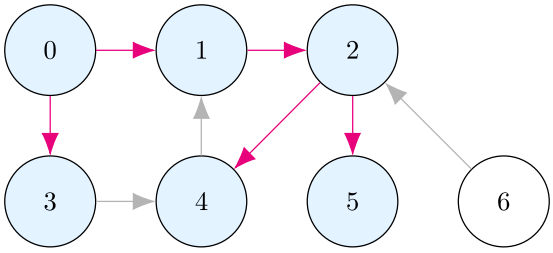
Proposition 8.2.1

L'algorithme de parcours générique visite tous les sommets accessibles depuis la source et uniquement ceux-là.

Supposons que l'on garde en mémoire le sommet depuis lequel on visite chaque sommet, en ne mettant pas le sommet y dans notre sac, mais plutôt le couple (x, y). Un parcours générique mettra ainsi en valeur un arbre, enraciné en la source, couvrant l'ensemble des sommets accessibles. Par exemple, en considérant le graphe ci-dessous,



si les couples que l'on sort du sac sont successivement $(\varnothing, 0)$, $(0, 1)$, $(1, 2)$, $(0, 3)$, $(2, 4)$ et $(2, 5)$, on obtient l'arbre enraciné suivant :



Par la suite, nous utiliserons principalement nos sacs pour y placer des sommets. Cependant, lors de certains raisonnements, il sera parfois utile de faire comme si on y avait placé un couple.

La structure de données que nous allons utiliser pour implémenter notre sac va déterminer l'ordre dans lequel les sommets en sont extraits.

— *Pile* : Si nous utilisons une pile, pour laquelle c'est le dernier sommet qui a été placé dans le sac qui en est extrait, nous obtiendrons ce qu'on appelle un *parcours en profondeur*. Bien que tous les parcours nous permettent d'obtenir l'ensemble des sommets accessibles depuis un sommet, la simplicité de la structure de pile fait que c'est souvent ce parcours que nous utiliserons pour cela. Nous verrons aussi que le parcours en profondeur nous permet de détecter les cycles dans un graphe.
— *File* : Si nous utilisons une file, pour laquelle c'est le premier sommet qui a été placé dans le sac qui en est extrait, nous obtiendrons ce qu'on appelle un *parcours en largeur*. Ce parcours nous sera utile pour trouver le chemin de longueur minimale entre la source et les sommets accessibles depuis cette dernière.
— *File de priorité* : L'utilisation d'une file de priorité nous permettra de découvrir une famille d'algorithmes fonctionnant avec les graphes pondérés. Ils se distinguent par les différentes priorités qu'ils utilisent.
  — *Dijkstra* : L'algorithme de Dijkstra nous permet de trouver le chemin de poids minimal entre la source et les sommets accessibles depuis cette dernière. Pour cela, la priorité utilisée est le poids du chemin qui nous a permis de découvrir le sommet.
  — *Prim* : L'algorithme de Prim nous permet de trouver un arbre de poids minimal couvrant l'ensemble des sommets accessibles. Pour cela, la priorité que nous utiliserons est le poids de l'arc qui nous a permis de découvrir le sommet.

Commençons par étudier le plus simple de ces parcours : le parcours en profondeur.

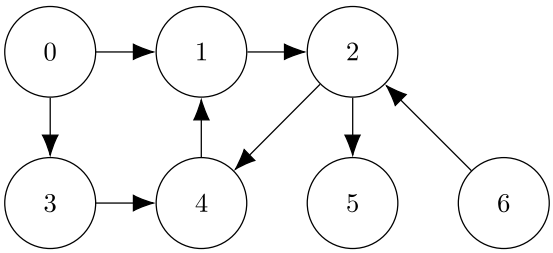
8.2.2 Parcours en profondeur

Version itérative

L'implémentation Python du parcours en profondeur se fait naturellement. On suppose ici que $\mathcal{S} \coloneqq [\![0, n[\![$ et que le graphe est représenté par sa liste d'adjacence. Pour marquer les sommets, nous utilisons le tableau *visité*, de longueur n, dont tous les éléments sont initialisés à `False`. Pour le sac, nous utilisons la liste *pile* que nous utilisons, comme son nom l'indique, comme une pile.

```
1  def profondeur(g, s):
2      """profondeur(g: list[list[int]], s: int) -> NoneType"""
3      n = len(g)
4      visite = [False for _ in range(n)]
5      pile = [s]
6      while len(pile) != 0:
7          x = pile.pop()
8          if not visite[x]:
9              visite[x] = True
10             for y in g[x]:
11                 pile.append(y)
```

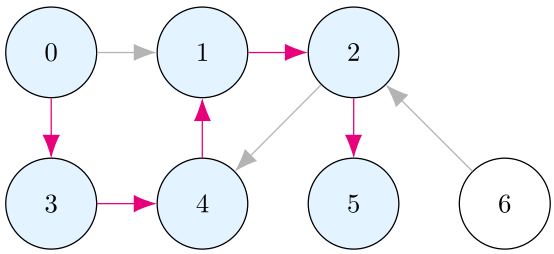
Si l'on souhaite effectuer une action pour chaque sommet, il suffit de définir une fonction `f(x: int) -> NoneType` que l'on appelle juste avant l'instruction `visite[x] = True`. Par exemple, si l'on souhaite afficher à l'écran les sommets visités dans l'ordre de notre parcours, il suffit d'insérer `print(x)` ligne 9.



Illustrons son fonctionnement en détail sur un exemple. Le tableau ci-dessous détaille les différentes étapes du parcours en profondeur du présent graphe à partir du sommet 0. Le contenu de la *pile* est détaillé lors du passage ligne 6, tout comme l'ensemble des sommets marqués dans *visité*.

pile	*visité*	action
`[0]`	$\{\}$	Dépiler 0, le marquer et empiler ses successeurs 1 et 3.
`[1, 3]`	$\{0\}$	Dépiler 3, le marquer et empiler son successeur 4.
`[1, 4]`	$\{0, 3\}$	Dépiler 4, le marquer et empiler son successeur 1.
`[1, 1]`	$\{0, 3, 4\}$	Dépiler 1, le marquer et empiler son successeur 2.
`[1, 2]`	$\{0, 1, 3, 4\}$	Dépiler 2, le marquer et empiler ses successeurs 4 et 5.
`[1, 4, 5]`	$\{0, 1, 2, 3, 4\}$	Dépiler 5 et le marquer. Il n'a pas de successeur.
`[1, 4]`	$\{0, 1, 2, 3, 4, 5\}$	Dépiler 4. Il a déjà été marqué.
`[1]`	$\{0, 1, 2, 3, 4, 5\}$	Dépiler 1. Il a déjà été marqué.
`[]`	$\{0, 1, 2, 3, 4, 5\}$	*pile* est vide donc l'algorithme se termine.

Une fois le parcours terminé, tous les sommets atteignables à partir du sommet 0 ont été marqués, à savoir $0, 1, 2, 3, 4$ et 5. Inversement, le sommet 6, qui n'est pas atteignable à partir de 0 n'a pas été marqué. C'est là une propriété fondamentale du parcours en profondeur. Le graphe ci-dessous met en valeur les sommets visités ainsi que les arcs empruntés lors de ce parcours.



Version récursive

Le parcours en profondeur est un algorithme fondamentalement récursif dont voici une implémentation Python :

```
def profondeur_rec(g, visite, x):
    """profondeur_rec(g: list[list[int]], visite: list[bool],
                      x: int) -> NoneType"""
    if not visite[x]:
        visite[x] = True
        for y in g[x]:
            profondeur_rec(g, visite, y)
```

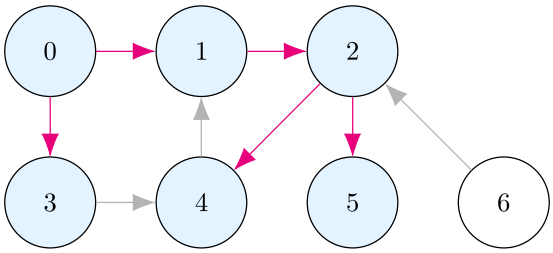
Pour lancer un parcours en profondeur depuis un sommet s, on utilisera la fonction suivante :

```
def profondeur(g, s):
    """profondeur(g: list[list[int]], s: int) -> NoneType"""
    n = len(g)
    visite = [False for _ in range(n)]
    profondeur_rec(g, visite, s)
```

Dans cette version, une pile est toujours présente par l'intermédiaire de la pile d'appels. Contrairement à ce qui se passe dans la version itérative, les sommets sont visités dès qu'ils sont découverts ; en récursif, ces deux états sont donc confondus. Le tableau ci-dessous détaille l'ensemble des sommets marqués dans *visité* au moment de l'appel de `profondeur_rec`. La pile d'appel est aussi représentée dans la colonne « chemin emprunté ».

visité	chemin emprunté	action
$\{\}$	0	Marquer 0, emprunter l'arc $0 \to 1$.
$\{0\}$	$0 \to 1$	Marquer 1, emprunter l'arc $1 \to 2$.
$\{0,1\}$	$0 \to 1 \to 2$	Marquer 2, emprunter l'arc $2 \to 4$.
$\{0,1,2\}$	$0 \to 1 \to 2 \to 4$	Marquer 4, emprunter l'arc $4 \to 1$.
$\{0,1,2,4\}$	$0 \to 1 \to 2 \to 4 \to 1$	Déjà découvert.
$\{0,1,2,4\}$	$0 \to 1 \to 2 \to 4$	Pas d'autre arc, terminé.
$\{0,1,2,4\}$	$0 \to 1 \to 2$	Emprunter l'arc $2 \to 5$.
$\{0,1,2,4,5\}$	$0 \to 1 \to 2 \to 5$	Marquer 5, aucun arc, terminé.
$\{0,1,2,4,5\}$	$0 \to 1 \to 2$	Pas d'autre arc, terminé.
$\{0,1,2,4,5\}$	$0 \to 1$	Pas d'autre arc, terminé.
$\{0,1,2,4,5\}$	0	Emprunter l'arc $0 \to 3$.
$\{0,1,2,4,5\}$	$0 \to 3$	Marquer 3, emprunter l'arc $3 \to 4$.
$\{0,1,2,3,4,5\}$	$0 \to 3 \to 4$	Déjà découvert.
$\{0,1,2,3,4,5\}$	$0 \to 3$	Pas d'autre arc, terminé.
$\{0,1,2,3,4,5\}$	0	Pas d'autre arc, terminé.

On remarque que même si les sommets visités sont les mêmes que dans la version itérative, les arcs empruntés diffèrent : le parcours que l'on vient d'effectuer est un autre parcours en profondeur.



Accessibilité, connexité

Une application immédiate du parcours en profondeur consiste à déterminer s'il existe un chemin entre deux sommets x et y. Pour cela, il suffit de lancer un parcours en profondeur à partir du sommet x puis, une fois qu'il est terminé, de regarder si le sommet y fait partie des sommets visités. Le programme suivant réalise cet algorithme :

```
def existe_chemin(g, x, y):
    """existe_chemin(g: list[list[int]], x: int, y: int) -> bool"""
    n = len(g)
    visite = [False for _ in range(n)]
    profondeur_rec(g, visite, x)
    return visite[y]
```

Le parcours en profondeur est un algorithme très efficace, dont la complexité temporelle est de l'ordre du nombre de sommets du graphe (pour l'initialisation de *visité*) auquel on ajoute le nombre d'arcs qui sont examinés pendant ce parcours. On a donc une complexité temporelle en $O(|S|+|A|)$. En effet, chaque arc $x \to y$ est examiné au plus une fois, à savoir la première fois que la fonction `profondeur_rec` est appelée sur le sommet x : si la fonction `profondeur_rec` est rappelée plus tard sur ce même sommet x, alors il sera trouvé dans *visité* et la fonction se terminera immédiatement. Dans le pire des cas, tous les sommets sont atteignables et le graphe est entièrement parcouru. Le cout est moindre lorsque certains sommets ne sont pas atteignables depuis le sommet de départ.

La complexité spatiale de la version récursive est en $\Theta(|S|)$: cette complexité provient du tableau *visité* dont la taille est $|S|$, auquel on ajoute la taille de la pile d'appels qui reste toujours inférieure au nombre de sommets puisque la succession de sommets passés en argument des appels actifs forme à chaque instant un chemin élémentaire.

On peut aussi tester la connexité d'un graphe non orienté de la même manière. On utilise pour cela la fonction `est_connexe` :

```
def tous_vrai(t):
    """tous_vrai(t: list[bool]) -> bool"""
    for v in t:
        if not v:
            return False
    return True
```

```
7
8  def est_connexe(g):
9      """est_connexe(g: list[list[int]]) -> bool"""
10     n = len(g)
11     visite = [False for _ in range(n)]
12     profondeur_rec(g, visite, 0)
13     return tous_vrai(visite)
```

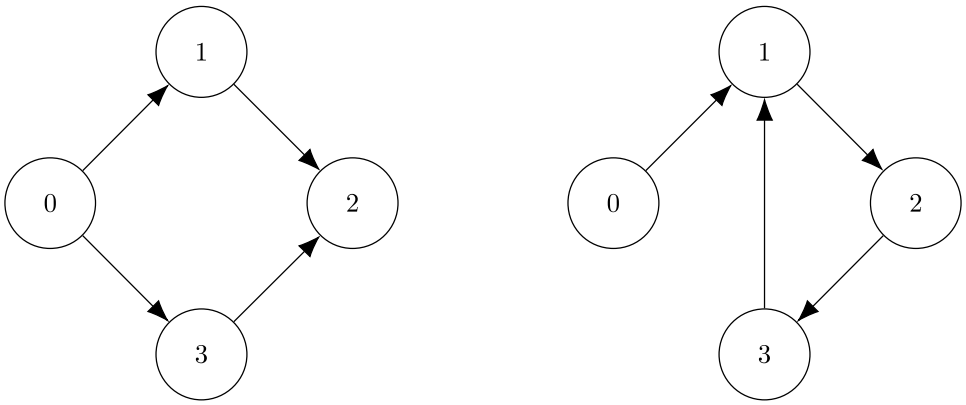
Cet algorithme a de nouveau une complexité temporelle en $O(|S| + |A|)$ et une complexité spatiale en $\Theta(|S|)$.

Exercice 2

⇨ Écrire une fonction qui compte le nombre de composantes connexes d'un graphe non orienté.

Détection de cycle

Le parcours en profondeur permet également de détecter la présence d'un cycle dans un graphe orienté. En effet, puisque l'on marque les sommets avant de considérer leurs voisins, pour justement éviter de tourner en rond dans un cycle, alors on doit pouvoir être à même de détecter leur présence. Il y a cependant une subtilité, car lorsqu'on retombe sur un sommet déjà marqué, on ne sait pas pour autant si l'on vient de découvrir un cycle. Considérons par exemple le parcours en profondeur des deux graphes suivants, à partir du sommet 0 à chaque fois.



Dans le graphe de gauche, on retombe sur le sommet 2. Il n'y a pas de cycle pour autant, mais seulement un chemin parallèle. Dans le graphe de droite, on retombe sur le sommet 1, cette fois à cause d'un cycle. Tel qu'il est écrit, notre parcours en profondeur ne nous permet de pas distinguer ces deux situations. Dans les deux cas, on constate que le sommet est déjà visité sans pouvoir en tirer de conclusion quant à l'existence d'un cycle.

Pour y remédier, on va distinguer dans notre marquage trois sortes de sommets : ceux que l'on n'a pas encore découverts qui seront marqués comme *inconnu*, ceux que l'on a *découvert* mais qui sont toujours présents dans le chemin déterminé par la pile d'appels (ces sommets sont donc *visités* puisque nous utilisons une implémentation récursive dans laquelle ces deux états sont confondus), et ceux qui ne sont plus présents dans la pile d'appels, qu'on marquera comme *fermé*. Le parcours en profondeur est modifié de la manière suivante : lorsqu'on visite un sommet x

- S'il est marqué comme « découvert », c'est qu'on vient de découvrir un cycle.
- S'il est marqué comme « fermé », on ne fait rien.
- S'il est marqué comme « inconnu », on procède ainsi :
  - On marque le sommet x comme « découvert ».
  - On visite tous ses successeurs, récursivement.
  - Enfin, on le marque comme « fermé ».

Comme on le voit, les successeurs du sommet x sont examinés après le moment où x est marqué comme « découvert » et avant le moment où il est marqué comme « fermé ». Ainsi, s'il existe un cycle nous ramenant sur x, on le trouvera comme étant « découvert » et le cycle sera signalé.

Le programme suivant réalise cette détection de cycle. La fonction `possede_cycle_rec(g, x, etat)` est toujours une fonction récursive, mais elle renvoie désormais un résultat, à savoir un booléen indiquant la présence d'un cycle. Enfin, la fonction `possede_cycle` marque tous les sommets comme « inconnu » puis lance un parcours en profondeur à partir de tous les sommets du graphe. Si l'un de ces parcours renvoie `True`, on transmet ce résultat. Sinon, on renvoie `False`.

```
1 INCONNU = 0
2 DECOUVERT = 1
```

```
3  FERME = 2
4
5  def possede_cycle_rec(g, etat, x):
6      """possede_cycle_rec(g: list[list[int]], etat: list[int], x: int) -> bool"""
7      if etat[x] == INCONNU:
8          etat[x] = DECOUVERT
9          for y in g[x]:
10             cycle = possede_cycle_rec(g, etat, y)
11             if cycle:
12                 return True
13         etat[x] = FERME
14         return False
15     else:
16         return etat[x] == DECOUVERT
```

Enfin, dans la version suivante, on cherche à détecter la présence d'un cycle n'importe où dans le graphe. C'est pourquoi on lance un parcours en profondeur à partir de tous les sommets du graphe.

```
1  def possede_cycle(g):
2      """possede_cycle(g: list[list[int]]) -> bool"""
3      n = len(g)
4      etat = [INCONNU for _ in range(n)]
5      for x in range(n):
6          cycle = possede_cycle_rec(g, etat, x)
7          if cycle:
8              return True
9      return False
```

Pour beaucoup de ces sommets, le parcours est déjà passé par là, car ils étaient accessibles depuis des sommets déjà parcourus ; la fonction `parcours_cycle_rec` se termine alors immédiatement sans rien faire. À nouveau, la complexité temporelle de cet algorithme est en $O(|S| + |A|)$, et sa complexité spatiale en $\Theta(|S|)$.

Attention à ne pas oublier que cet algorithme ne fonctionne que pour les graphes orientés. Il nécessite quelques aménagements pour fonctionner avec les graphes non orientés.

8.2.3 Parcours en largeur

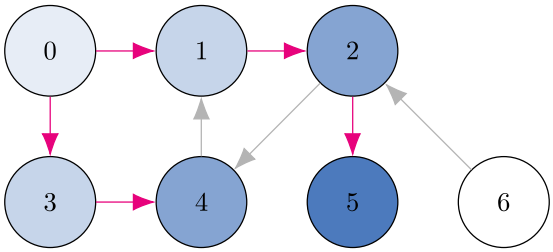
Le parcours en largeur se fait simplement en utilisant une liste pour implémenter le sac de notre parcours générique :

```
1  import collections
2
3  def largeur(g, s):
4      """largeur(g: list[list[int]], s: int) -> NoneType"""
5      n = len(g)
6      visite = [False for _ in range(n)]
7      file = collections.deque()
8      file.append(s)
9      while len(file) != 0:
10         x = file.popleft()
11         if not visite[x]:
12             visite[x] = True
13             for y in g[x]:
14                 file.append(y)
```

Illustrons son fonctionnement sur un notre exemple. Le tableau ci-dessous détaille les différentes étapes du parcours en largeur du graphe précédent à partir du sommet 0. Le contenu de la *file* est détaillé lors du passage ligne 9, tout comme l'ensemble des sommets marqués dans *visité*.

file	*visité*	action
`[0]`	$\{\}$	Défiler 0, le marquer et enfiler ses successeurs 1 et 3.
`[1, 3]`	$\{0\}$	Défiler 1, le marquer et enfiler son successeur 2.
`[3, 2]`	$\{0,1\}$	Défiler 3, le marquer et enfiler son successeur 4.
`[2, 4]`	$\{0,1,3\}$	Défiler 2, le marquer et enfiler ses successeurs 4 et 5.
`[4, 4, 5]`	$\{0,1,2,3\}$	Défiler 4, le marquer et empiler son successeurs 1.
`[4, 5, 1]`	$\{0,1,2,3,4\}$	Défiler 4. Il a déjà été marqué.
`[5, 1]`	$\{0,1,2,3,4\}$	Défiler 5, le marquer. Il n'a pas de successeur.
`[1]`	$\{0,1,2,3,4,5\}$	Défiler 1. Il a déjà été marqué donc on ne fait rien.
`[]`	$\{0,1,2,3,4,5\}$	*file* est vide donc l'algorithme se termine.

Tout comme le parcours en profondeur, le parcours en largeur a visité exactement les sommets atteignables à partir du sommet 0. Voici les sommets visités ainsi que les arcs empruntés lors de ce parcours.



On observe que les arcs ainsi mis en valeur soulignent les chemins de longueur minimale entre la source 0 et les sommets atteignables depuis cette source. On voit que les sommets 1 et 3 sont à une distance de 1 de la source, les sommets 2 et 4 sont à une distance de 2 et le sommet 5 est à une distance de 3.

Comme pour le parcours en profondeur, un même sommet peut apparaitre plusieurs fois dans notre sac, mais le fait qu'on travaille ici avec une *file* fait que les sommets sortent de la file dans le même ordre que celui dans lequel ils y sont entrés. Une fois qu'un sommet y est entré, il est donc inutile de l'enfiler de nouveau. Cette remarque nous permet l'optimisation suivante : au lieu de marquer les sommets lorsqu'ils sont *visités*, c'est-à-dire lorsqu'ils sortent de la file, nous allons les marquer lorsqu'ils sont *découverts*, c'est-à-dire lorsqu'ils y entrent.

```
1  def largeur(g, s):
2      """largeur(g: list[list[int]], s: int) -> NoneType"""
3      n = len(g)
4      decouvert = [False for _ in range(n)]
5      file = collections.deque()
6      decouvert[s] = True
7      file.append(s)
8      while len(file) != 0:
9          x = file.popleft()
10         for y in g[x]:
11             if not decouvert[y]:
12                 decouvert[y] = True
13                 file.append(y)
```

Contrairement à l'algorithme initial qu'on appelle parfois parcours en largeur à marquage *tardif*, cette nouvelle version est appelée parcours en largeur à marquage *précoce*. Avec cette optimisation, les valeurs successives de *file* sont : `[0]`, `[1, 3]`, `[3,2]`, `[2, 4]`, `[4, 5]`, `[5]` et enfin `[]`. On remarque qu'à chaque étape, la file est composée d'une succession de sommets dont la distance à la source est d suivie d'une succession (éventuellement vide) de sommets dont la distance à la source est $d+1$. On observe donc que le parcours en largeur explore le graphe en « cercles concentriques » à partir de la source.

Cette idée de cercles concentriques nous permet une dernière transformation de notre programme dans lequel on va travailler non pas avec un sac fonctionnant comme une file, mais avec deux sacs. À chaque instant, un des sacs, qu'on appelle *courant*, contient des sommets situés à une distance d de la source, tandis que l'autre sac, qu'on appelle *suivant*, contient des sommets à une distance $d+1$ de la source. On examinera ces derniers une fois que le sac *courant* est vide. Par soucis de simplicité, nous utiliserons une pile pour implémenter ces deux sacs, mais nous aurions pu utiliser n'importe quelle structure de donnée séquentielle. À côté de ces deux piles, on utilise un tableau *découvert* qui marque les sommets déjà découverts. Le parcours en largeur procède ainsi :

— Initialement, la source est empilée dans *courant* et on la marque comme *découverte*.
— Tant que la pile *courant* n'est pas vide

— On dépile le sommet x de *courant*.
— Pour chaque successeur y de x, s'il n'a pas été marqué comme *découvert*, on le marque et on l'empile dans *suivant*.
— Si la pile *courant* est vide, on l'échange avec la pile *suivant*.

Si l'on reprend l'exemple de notre graphe exemple et qu'on effectue un parcours en largeur à partir du sommet 0, on obtient le parcours résumé dans le tableau suivant.

vu	*courant*	*suivant*	action
$\varnothing$	`[]`	`[]`	Le sommet 0 est marqué puis empilé dans *courant*.
$\{0\}$	`[0]`	`[]`	On dépile le sommet 0 ; 1 et 3 sont marqués puis empilés dans *suivant*.
$\{0,1,3\}$	`[]`	`[1, 3]`	La pile *courant* est vide ; on échange.
$\{0,1,3\}$	`[1, 3]`	`[]`	On dépile le sommet 3 ; 4 est marqué puis empilé dans *suivant*.
$\{0,1,3,4\}$	`[1]`	`[4]`	On dépile le sommet 1 ; 2 est marqué puis empilé dans *suivant*.
$\{0,1,2,3,4\}$	`[]`	`[4, 2]`	La pile *courant* est vide ; on échange.
$\{0,1,2,3,4\}$	`[4, 2]`	`[]`	On dépile le sommet 2 ; 5 est marqué puis empilé dans *suivant*.
$\{0,1,2,3,4,5\}$	`[4]`	`[5]`	On dépile le sommet 4.
$\{0,1,2,3,4,5\}$	`[]`	`[5]`	La pile *courant* est vide ; on échange.
$\{0,1,2,3,4,5\}$	`[5]`	`[]`	On dépile le sommet 5.
$\{0,1,2,3,4,5\}$	`[]`	`[]`	*courant* est vide donc l'algorithme se termine.

L'implémentation Python de cet algorithme se fait alors naturellement.

```
1  def largeur(g, s):
2      """largeur(g: list[list[int]], s: int) -> NoneType"""
3      n = len(g)
4      decouvert = [False for _ in range(n)]
5      decouvert[s] = True
6      courant = [s]
7      suivant = []
8      while len(courant) != 0:
9          x = courant.pop()
10         for y in g[x]:
11             if not decouvert[y]:
12                 decouvert[y] = True
13                 suivant.append(y)
14         if len(courant) == 0:
15             courant = suivant
16             suivant = []
```

Si l'on souhaite effectuer une action pour chaque sommet à l'aide d'une fonction `f(x: int) -> NoneType`, on l'appellera juste avant d'avoir marqué le sommet avec l'instruction `decouvert[x] = True`, c'est-à-dire aux lignes 5 et 12.

Comme pour le parcours en profondeur, le parcours en largeur a une complexité temporelle en $O(|S|+|A|)$ et une complexité spatiale en $\Theta(|S|)$. En effet, chaque sommet est placé au plus une fois dans la pile *suivant*, la première fois qu'il est rencontré. Donc chaque arc $x \to y$ est examiné au plus une fois, lorsque le sommet x est retiré de l'ensemble courant.

Exercice 3

⇨ Écrire une fonction prenant entrée un graphe et une source et renvoyant le tableau des distances de chaque sommet à la source. Le tableau contiendra `None` pour un sommet qui n'est pas accessible.

8.2.4 Plus court chemin

Dans cette section, on se donne un graphe pondéré $G := (S, A, \rho)$ ainsi qu'une source s. On rappelle que le fonction de poids avec laquelle on travaille est positive. On cherche à déterminer pour chaque sommet x, le poids minimal d'un chemin reliant s à x. Afin d'utiliser une terminologie plus conventionnelle, on imaginera que les poids représentent des distances et on utilisera les termes de distance et de plus court chemin.

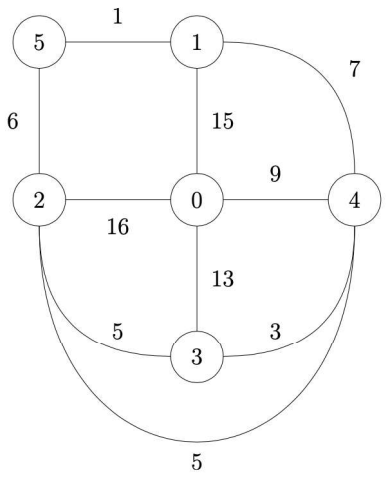
Algorithme de Dijkstra

L'implémentation de l'algorithme de Dijkstra se fait simplement en utilisant une file de priorité pour notre sac dans notre parcours générique. Les priorités sont des distances et c'est ainsi que nous les appelerons dans notre description.
— On commence par insérer la source dans la file avec une distance de 0.

— Tant que la file de priorité n'est pas vide :
  — On récupère l'élément x ayant la plus faible distance δ.
  — Si x n'a pas encore été visité :
    — On le marque comme visité. La distance δ est alors la distance entre la source et x.
    — Pour tous ses successeurs y, on les insère dans la file de priorité avec la distance $\delta + \rho(x \to y)$.

Remarquons qu'un même sommet pourra se retrouver plusieurs fois dans la file de priorité, avec des distances différentes. L'algorithme de parcours ne traitant que les sommets de la file qui n'en sont pas encore sortis, si l'on insère un sommet x avec une distance δ' et qu'il est déjà présent dans la file avec une distance $\delta \leqslant \delta'$, cette insertion n'affecte pas le déroulement de notre programme. Si par contre $\delta' < \delta$, c'est l'ancien élément présent dans la file qui est ignoré. Tout se passe donc comme si la distance δ du sommet x était mise à jour à $\min(\delta, \delta')$.

Afin de se familiariser avec cet algorithme, nous allons l'exécuter sur le graphe suivant, en utilisant le sommet 0 pour source.



— On commence à placer le sommet 0 dans la file avec la distance 0.
— Le seul sommet de la file est le sommet 0. C'est donc celui dont la distance est minimale. On marque ce sommet comme visité puis on regarde ses voisins : 1, 2, 3 et 4. On les place dans la file de priorité avec les distances respectives de 15, 16, 13 et 9.
— Le sommet de la file ayant une distance minimale est 4. Cette distance est de 9. On le marque comme visité, puis on regarde ses voisins : 1, 3 et 2. Le chemin passant par 4 et allant à 1 a une distance de 16 qui n'est pas inférieure à la distance actuelle pour 1. Par contre, le chemin passant par 4 et allant à 3 a une distance de 12 qui est inférieure à la distance actuelle de 13 pour 3. C'est aussi le cas du chemin passant par 4 et allant à 2 dont la distance est 14. On met donc ces distances à jour dans notre file. Les sommets de la file sont donc désormais les sommets 1, 2, et 3 de distances respectives 15, 14 et 12.
— Le sommet de la file ayant une distance minimale est 3. Cette distance est de 12. Aucun des chemins passant par 3 et menant à ses voisins ne permet d'obtenir une meilleure distance que celle que nous avons actuellement.
— Le sommet de la file ayant une distance minimale est 2. On marque ce sommet comme visité, puis on regarde ses voisins : 4, 3, 0 et 5. La distance du sommet 2 étant 14, on insère donc le sommet 5 avec une distance de 20. Les sommets de la file sont désormais les sommets 1 et 5 de distances respectives 15 et 20.
— Le sommet de la file ayant une distance minimale est 1. Cette distance est de 15. On marque ce sommet comme visité, puis on regarde ses voisins : 0, 4 et 5. Le chemin passant par 1 et allant à 5 a une distance de 16 qui est inférieure à la distance temporaire de 20. On met donc à jour cette distance.
— Le sommet de la file ayant une distance minimale est 5. Cette distance est de 16. On marque ce sommet comme visité. L'étude de ses voisins ne donne lieu à aucune mise à jour, car ils ont déjà tous été traités.
— À la boucle suivante, il n'y a plus de sommet dans notre file et l'algorithme s'arrête. Les distances du sommet 0 aux autres sommets du graphe sont donc 0 pour 0, 9 pour 4, 12 pour 3, 14 pour 2, 15 pour 1 et 16 pour 5.

Pour l'implémentation, nous utilisons le module `heapq` de Python.

```python
import heapq

def dijkstra(g, s):
    """dijkstra(g: list[list[tuple[float, int]]], s: int) -> list[float]"""
    n = len(g)
    visite = [False for _ in range(n)]
```

```
    dist = [None for _ in range(n)]
    filep = []
    heapq.heappush(filep, (0.0, s))
    while len(filep) != 0:
        delta, x = heapq.heappop(filep)
        if not visite[x]:
            dist[x] = d
            visite[x] = True
            for rho, y in g[x]:
                heapq.heappush(filep, (delta + rho, y))
    return dist
```

Si nous ne disposons pas de file de priorité, nous pouvons en faire une implémentation à la main (qui ne sera malheureusement pas très efficace). Pour cela, nous allons utiliser deux tableaux *visité* et *dist* dont la longueur est le nombre n de sommets du graphe. Lorsqu'un sommet x est dans l'état *inconnu* c'est-à-dire lorsqu'il n'a pas encore été inséré dans la liste, `dist[x]` est égal à `None` et `visite[x]` est égal à `False`. Lorsqu'un sommet x est dans la file, `dist[x]` contient sa priorité, et `visite[x]` est égal à `False`. Enfin, lorsque x est sorti de la file de priorité, `dist[x]` contient la distance de la source au sommet x et `visite[x]` est égal à `True`. On commence par écrire une fonction `prochain_sommet` qui renvoie le sommet de la file dont la distance est minimale.

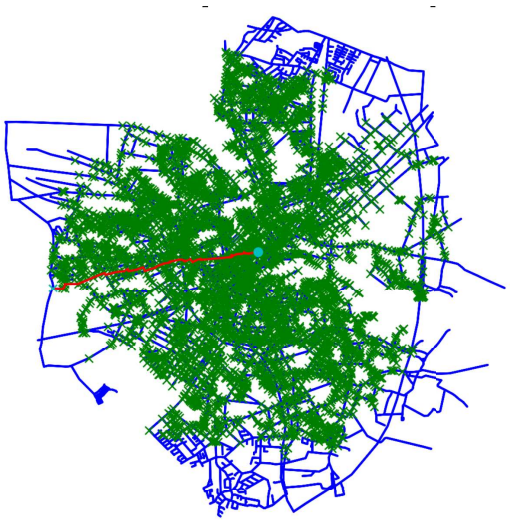
```
def prochain_sommet(dist, visite):
    """prochain_sommet(dist: list[float], visite: list[bool]) -> int"""
    n = len(dist)
    min_value = None
    min_x = None
    for x in range(n):
        if (not visite[x]) and (dist[x] != None) \
                and (min_value == None or dist[x] < min_value):
            min_value = dist[x]
            min_x = x
    return min_x
```

L'algorithme de Dijkstra s'écrit alors naturellement.

```
def dijkstra(g, s):
    """dijkstra(g: list[list[tuple[float, int]]], s: int) -> list[float]"""
    n = len(g)
    visite = [False for _ in range(n)]
    dist = [None for _ in range(n)]
    dist[s] = 0.0
    while True:
        x = prochain_sommet(dist, visite)
        if x == None:
            break
        visite[x] = True
        for rho, y in g[x]:
            delta = dist[x] + rho
            if dist[y] == None or delta < dist[y]:
                dist[y] = delta
    return dist
```

Algorithme A*

L'algorithme de Dijkstra permet de déterminer la distance d'une source s à l'ensemble des sommets accessibles depuis s. Si l'on s'intéresse uniquement à la distance entre la source et un but b, il est possible d'arrêter l'algorithme dès que le sommet b est marqué comme visité. Sur la figure ci-dessous, on a réalisé une recherche du meilleur chemin dans la ville d'Oldenburg, en Allemagne, en partant d'une source située en centre-ville et pour aller vers un but situé en périphérie, à l'ouest de la ville. Le chemin optimal trouvé est en rouge.



Nous avons colorié en vert l'ensemble des sommets « visités » par l'algorithme de Dijkstra. La zone couverte par l'algorithme est formée de tous les points dont la distance à la source est inférieure à la distance entre s et b. Cependant, notre intuition nous dit qu'il n'est surement pas utile d'aller traiter des sommets qui se trouvent tout à l'est de la ville alors que notre but est à l'ouest. Cette intuition se fonde sur le fait que la distance entre deux sommets x et y du graphe est supérieure à la distance à vol d'oiseau entre ces deux sommets.

Pour exploiter cette idée, nous allons définir la fonction $h : S \to \mathbb{R}$, appelée *heuristique*, par

$$\forall x \in S, \quad h(x) \coloneqq \|x - b\|$$

et définir une nouvelle distance ρ' sur A en définissant, pour tout arc $x \to y$

$$\rho'(x \to y) = \rho(x \to y) + h(y) - h(x).$$

Remarquons tout d'abord que ρ' est bien à valeurs positives puisque si $x \to y$ est un arc

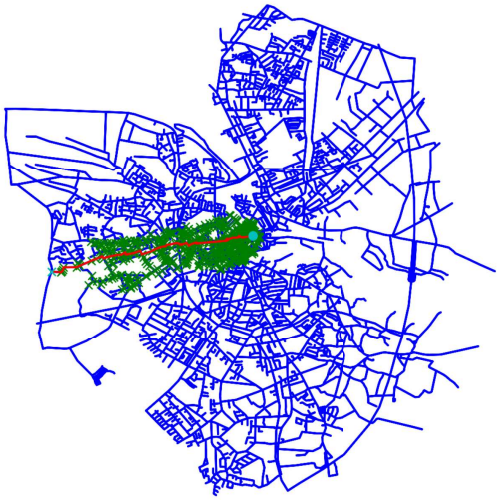
$$\rho(x \to y) \geqslant \|x - y\| \geqslant |\|x - b\| - \|y - b\|| \geqslant \|x - b\| - \|y - b\| = h(x) - h(y),$$

donc $\rho'(x \to y) = \rho(x \to y) + h(y) - h(x) \geqslant 0$. Remarquons enfin que, quel que soit le chemin $z_0 \to z_1 \to \cdots \to z_n$, on a $\rho'(z_0 \to z_1 \to \cdots \to z_n) = \rho(z_0 \to z_1 \to \cdots \to z_n) + h(z_n) - h(z_0)$. En particulier

$$\rho'(s \to z_1 \to \cdots \to z_{n-1} \to b) = \rho(s \to z_1 \to \cdots \to z_{n-1} \to b) + h(b) - h(s).$$

Puisque $h(b) - h(s)$ est indépendant du chemin allant de s à b, tout chemin entre ces deux sommets de distance minimale pour ρ' est minimal pour ρ. L'algorithme de Dijkstra appliqué sur le même graphe avec la distance ρ' au lieu de la distance ρ donnera donc un même chemin minimal entre s et b. Remarquons que cette nouvelle distance va privilégier les sommets se situant en direction du but. En effet, dans le cas extrême où il existe une ligne droite entre la source et le but et plusieurs sommets du graphe sont alignés, la distance entre ces sommets pour la nouvelle distance ρ' va être nulle et ce sont bien ces sommets qui vont être traités en premier.

En reprenant la recherche du meilleur chemin entre le centre et un point de la périphérie d'Oldenburg, on voit que l'algorithme de Dijkstra appliqué à nouvelle distance traite beaucoup moins de sommets avant d'arriver sur le but, tout en garantissant le fait que le chemin trouvé a une distance minimale pour la distance d'origine.



Chapitre 9

Langage Python

Cette annexe liste limitativement les éléments du langage Python (version 3 ou supérieure) dont la connaissance est exigible des étudiants. Aucun concept sous-jacent n'est exigible au titre de la présente annexe.

Aucune connaissance sur un module particulier n'est exigible des étudiants.

Toute utilisation d'autres éléments du langage que ceux que liste cette annexe, ou d'une fonction d'un module, doit obligatoirement être accompagnée de la documentation utile, sans que puisse être attendue une quelconque maîtrise par les étudiants de ces éléments.

Traits généraux

- Typage dynamique : l'interpréteur détermine le type à la volée lors de l'exécution du code.
- Principe d'indentation.
- Portée lexicale : lorsqu'une expression fait référence à une variable à l'intérieur d'une fonction, Python cherche la valeur définie à l'intérieur de la fonction et à défaut la valeur dans l'espace global du module.
- Appel de fonction par valeur : l'exécution de `f(x)` évalue d'abord x puis exécute f avec la valeur calculée.

Types de base

- Opérations sur les entiers (`int`) : `+`, `-`, `*`, `//`, `**`, `%` avec des opérandes positifs.
- Opérations sur les flottants (`float`) : `+`, `-`, `*`, `/`, `**`.
- Opérations sur les booléens (`bool`) : `not`, `or`, `and` (et leur caractère paresseux).
- Comparaisons `==`, `!=`, `<`, `>`, `<=`, `>=`.

Types structurés

- Structures indicées immuables (chaînes, tuples) : `len`, accès par indice positif valide, concaténation `+`, répétition `*`, tranche.
- Listes : création par compréhension `[e for x in s]`, par `[e] * n`, par `append` successifs ; `len`, accès par indice positif valide ; concaténation `+`, extraction de tranche, copie (y compris son caractère superficiel) ; `pop` en dernière position.
- Dictionnaires : création `{c_1 : v_1, ..., c_n : v_n}`, accès, insertion, présence d'une clé `k in d`, `len`, `copy`.

Structures de contrôle

- Instruction d'affectation avec `=`. Dépaquetage de tuples.
- Instruction conditionnelle : `if`, `elif`, `else`.
- Boucle `while` (sans `else`). `break`, `return` dans un corps de boucle.
- Boucle `for` (sans `else`) et itération sur `range(a, b)`, une chaîne, un tuple, une liste, un dictionnaire au travers des méthodes `keys` et `items`.
- Définition d'une fonction `def f(p_1, ..., p_n)`, `return`.

Divers

- Introduction d'un commentaire avec `#`.
- Utilisation simple de `print`, sans paramètre facultatif.
- Importation de modules avec `import module`, `import module as alias`, `from module import f, g, ...`

— Manipulation de fichiers texte (la documentation utile de ces fonctions doit être rappelée ; tout problème relatif aux encodages est éludé) : `open`, `read`, `readline`, `readlines`, `split`, `write`, `close`.
— Assertion : `assert` (sans message d'erreur).