Les graphes

Table des matières

Introduction

Initiée par le grand mathématicien suisse Euler, avec le célèbre problème des 7 ponts de Königsberg, les applications de la théorie des graphes et de la recherche opérationnelle sont aujourd'hui immenses tant au plan civil que militaire :

- aide à la prise de décision ;
- recherche de la meilleure stratégie ;
- optimisation (plus court chemin, GPS, coût minimal, ordonnancement des tâches...) ;
- réseaux de transports (autoroutes, chemins de fer, métro, lignes aériennes ...) ;
- transport de l'énergie (électricité, gaz ...) ;
- transport de l'information : internet, réseaux sociaux ...

I. Définition d'un graphe

Il existe deux types de graphes : les graphes *orientés* et les graphes *non-orientés*.

1) Graphes non orientés

Définition 1. *Graphes non orientés*

On appelle *graphe non orienté* tout couple $G = (S, A)$, où :

- S est un ensemble fini non vide dont les éléments sont appelés *nœuds* ou *sommets* ;
- A est un ensemble de paires $\{s, t\}$, où s et t sont deux éléments **distincts** de S. Ces paires sont appelées *arêtes*.

  On dit que deux sommets s et t sont reliés par une arête si et seulement si $\{s, t\} \in A$.

Cette représentation interdit le fait d'avoir plusieurs arêtes reliant deux sommets. Les graphes autorisant de telles arêtes sont appelés *multigraphes* et ils sont hors programme.

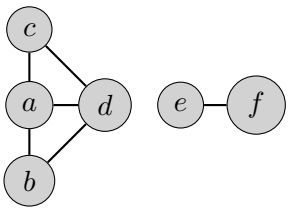
Définitions 2

- Le nombre de sommets que nous noterons n se nomme l'*ordre* du graphe.
- Deux sommets reliés par une arête sont dits *adjacents*. On appelle *voisin* d'un sommet s tout sommet adjacent à s.
- Le *degré* d'un sommet s que nous noterons $d(s)$ est le nombre de voisins de ce sommet.

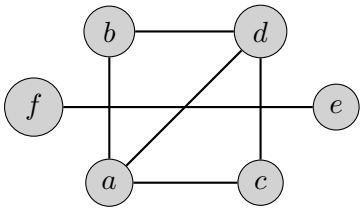
Exemple :

Supposons que l'ensemble des sommets soit $S = \{a, b, c, d, e, f\}$ et que l'ensemble des arêtes soit $A = \{\{a, b\}, \{a, c\}, \{a, d\}, \{b, d\}, \{c, d\}, \{e, f\}\}$.

On peut alors représenter le graphe de plusieurs façons, parmi toutes celles possibles, les représentations (1) et (2) conviennent.



Représentation (1)



Représentation (2)

FIGURE 1 – Représentations d'un graphe non orienté

Ainsi le graphe présenté en figure 1 est d'ordre $n = 6$ et $d(a) = d(d) = 3$, $d(b) = d(c) = 2$, $d(e) = d(f) = 1$.

Exercice : Quel est le nombre maximal d'arêtes dans un graphe d'ordre n ?

2) Graphes orientés

Définition 1. *Graphe orienté*

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 (s, t), où s et t sont deux éléments de S. Ces couples sont appelées arcs. En particulier, on admet (s, s) comme élément possible de A.

Si $(s, t) \in A$ on dit que le sommet s est *l'origine* de l'arc et que t en est *l'extrémité*. On dit aussi que t est un *successeur* s et que s est un *prédécesseur* de t.

Exemple :

Supposons que l'ensemble des sommets soit $S = \{a, b, c, d\}$ et que l'ensemble des arêtes soit $A = \{(a, a), (a, b), (a, c), (a, d), (b, d), (d, a), (d, c)\}$. La figure 2 propose une représentation possible de ce graphe orienté sous forme sagittal (les arcs étant représentés par des flèches entre deux sommets) :

Remarques :

- On notera la différence entre un graphe non orienté pour lequel chaque arête est un ensemble de deux éléments non ordonnés et un graphe orienté pour lequel un arc est un couple dont les deux éléments sont ordonnés afin de définir l'orientation de l'arc.

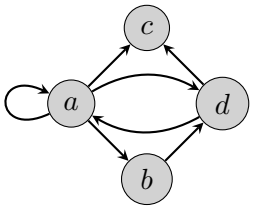


FIGURE 2 – Représentation d'un graphe orienté

- Une *boucle* d'un graphe orienté $G = (S, A)$ est un arc de la forme $(s, s) \in A$.

Définition 2. *Degrés*

Dans un graphe orienté $G = (S, A)$, on appelle :

- *degré entrant* d'un sommet s, noté $d^+(s)$, le nombre d'arcs $(t, s) \in A$ dont l'extrémité est s.
- *degré sortant* d'un sommet s, noté $d^-(s)$, le nombre d'arcs de la forme $(s, t) \in A$ dont s est l'origine.
- *degré total* d'un sommet s, noté $d(s)$, la somme de son degré entrant et de son degré sortant : $d(s) = d^-(s) + d^+(s)$.

Remarque :

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

3) Notation unifiée pour les graphes orientés et non-orientés

On a vu que pour un graphe non-orienté A est un ensemble de paires $\{s, t\}$ tandis que pour un graphe orienté, c'est un ensemble de couples (s, t).

Or un certain nombre de propriétés peuvent être énoncées pour les deux types de graphes. Afin de ne pas réécrire deux fois la propriété, une fois avec des paires et un autre avec des couples, nous allons convenir de la *notation unifiée* suivante. Si s et t sont deux sommets, alors :

$$[s, t] = \begin{cases} \{s, t\} & \text{pour un graphe non-orienté} \\ (s, t) & \text{pour un graphe orienté} \end{cases}$$

4) Graphes isomorphes. Sous-graphes

Définition 1. *Graphes isomorphes*

Deux graphes $G = (S, A)$ et $G' = (S', A')$ sont dits *isomorphes* si et seulement s'il existe une **bijection** $f : S \longrightarrow S'$ vérifiant :

$$\forall\, (s, t) \in S^2,\ [s, t] \in A \iff [\, f(s), f(t)\,] \in A'$$

Autrement dit le passage de G à G' se fait en remplaçant les sommets de S par ceux de S' selon la correspondance bijective f, mais en gardant les mêmes arêtes (arcs).

Exemples : dans les deux cas ci-dessous, les deux graphes G et G' sont-ils isomorphes ou non ?

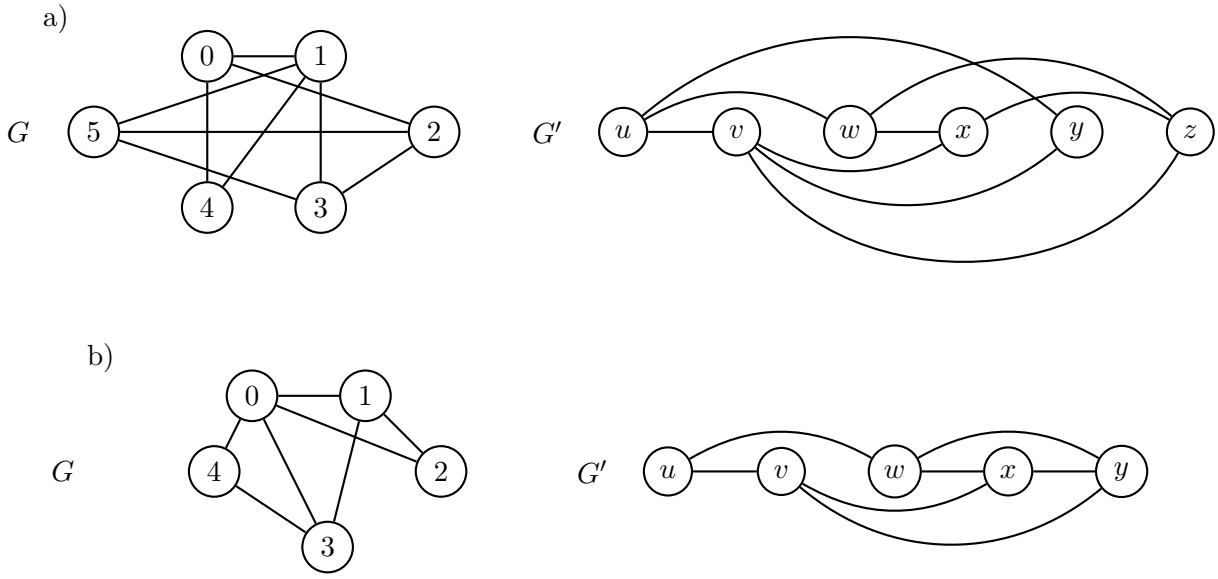


FIGURE 3 – Graphes isomorphes ou non ?

Définition. *Sous - graphe*

Soit $G = (S, A)$ un graphe orienté ou non. On dit que $G' = (S', A')$ est un *sous-graphe* de G si et seulement si $S' \subset S$ et $A' \subset A$.

En particulier on appelle sous-graphe de $G = (S, A)$ **engendré** par $S' \subset S$ le graphe $G' = (S', A')$ tel que :

$$A' = \{\, (s,t) \in A \mid (s,t) \in S' \,\}$$

5) Pondération d'un graphe

Dans le cas d'une carte routière, les arêtes (arcs) représentent les routes reliant deux sommets routiers. Il peut être intéressant, comme sur une carte, d'associer à chaque route sa longueur. Cette démarche fait appel à la notion de graphe pondéré.

Définition. *Graphe pondéré*

Un graphe *pondéré* est un triplet $G = (S, A, w)$, où (S, A) est un graphe (orienté ou non) et $w : A \longrightarrow \mathbb{R}$ est une application associant un *poids* $w([s,t])$ à chacune des arêtes ou arcs du graphe.

Notation :

Dans toute la suite et dans le but de simplifier les notations, on écrira systématiquement $w(s,t)$ à la place de $w([s,t])$.

II. Implémentation d'un graphe

En informatique, il y a deux façons de représenter un graphe : une représentation par *matrice d'adjacence* et une représentation par *liste d'adjacence*.

Soit $G = (S, A)$ un graphe d'ordre n ($|S| = n$), orienté ou non. Comme l'ensemble des sommets est non vide et fini, il existe une bijection $f : S \longrightarrow S' = \{0, \dots, n-1\}$ (ce processus revient à numéroter chaque sommet). Il s'ensuit que les graphes $G = (S, A)$ et $G' = (S', A')$ tel que :

$$A' = \{\, [f(s), f(t)] \mid [s, t] \in A \,\}$$

sont **isomorphes**.

On peut ainsi se ramener systématiquement à l'étude et à l'implémentation informatique d'un graphe G' dont les sommets sont des entiers naturels et c'est ce que nous allons faire dans toute la suite en convenant que l'ensemble S des sommets est :

$$S = \{\, 0, \dots, n-1 \,\}$$

et que les arêtes (arcs) sont de la forme $[i, j]$, avec $0 \leqslant i \leqslant n-1$ et $0 \leqslant j \leqslant n-1$.

Dans le cas d'un graphe pondéré $G = (S, A, w)$ avec $w : A \longmapsto \mathbb{R}$, on posera : $w_{ij} = w(\,[i, j]\,)$.

1) Matrice d'adjacence

Définition 1. *Matrice d'adjacence d'un graphe orienté ou non*

Soit $G = (S, A)$ un graphe orienté ou non, avec $S = \{0, \dots, n-1\}$. La *matrice d'adjacence* de G est la matrice carrée $M = (m_{ij})$ d'ordre n dont les coefficients m_{ij} tels que $0 \leqslant i \leqslant n-1$ et $0 \leqslant j \leqslant n-1$ sont définis par :

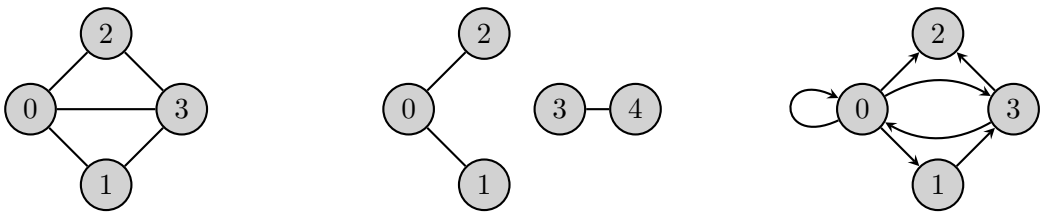
$$m_{ij} = \begin{cases} 1 \text{ si } [i, j] \in A \\ 0 \text{ sinon} \end{cases}$$

Quelle propriété vérifie la matrice d'adacence d'un graphe non orienté ?

Dans le cas d'un graphe pondéré $G = (S, A, w)$, on modifie légèrement la définition de M en attribuant à m_{ij} une valeur différentes de toutes les autres (par exemple la valeur `None` en python) lorsque $[i, j] \notin A$ et la valeur w_{ij} si $[i, j] \in A$.

Exemples :

Écrire les matrices d'adjacence des trois graphes (non pondérés) donnés ci-dessous.



Propriétés :

Soit $G = (S, A)$ un graphe **orienté** où $S = \{0, \dots, n-1\}$ et $M = (m_{ij})_{0 \leqslant i,\, j \leqslant n-1}$ sa matrice d'adjacence. Soit $i \in \{0, \dots, n-1\}$:

1. Comment calculer le degré sortant $d^-(i)$?
2. Comment calculer le degré entrant $d^+(i)$?
3. Que représente la trace de M ?
4. Que représente la somme des coefficients de M (i.e. $\displaystyle\sum_{0 \leqslant i,\, j \leqslant n-1} m_{ij}$) ?

Complexité :

Pour un graphe d'ordre n la complexité spatiale (càd la taille mémoire nécessaire) de l'implémentation de la matrice d'adjacence est en $\mathcal{O}(n^2)$.

La recherche d'un arc s'effectue en temps constant, cependant, pour **déterminer si un sommet possède un voisin ou un successeur**, il est nécessaire, dans le pire des cas, d'effectuer un parcours de la ligne correspondante de la matrice et la **complexité est donc linéaire** en $\mathcal{O}(n)$.

2) Liste d'adjacence

Pour un graphe peu dense (grand nombre de sommets et peu d'arêtes), la matrice d'adjacence contient beaucoup de 0 ce qui occupe de l'espace mémoire inutilement. On préfère alors représenter un graphe par une *liste d'adjacence*.

Le langage Python nous donne la structure de liste. Cela permet de construire des représentations efficaces des graphes peu denses.

Définition. *Liste d'adjacence d'un sommet et d'un graphe*

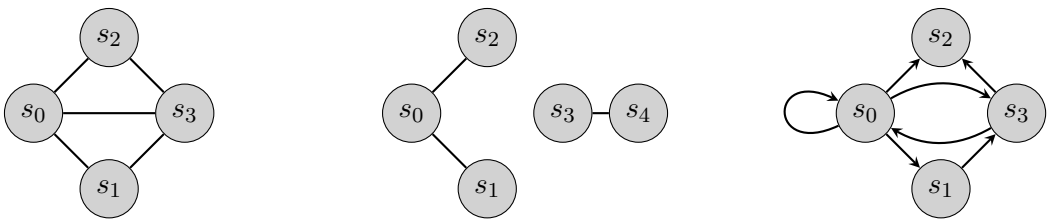
Dans un graphe non orienté (resp. orienté), la *liste d'adjacence d'un sommet* est la liste de ses voisins (resp. successeurs).

La liste d'adjacence d'un graphe `Ladj` est une **liste de listes** telle que `Ladj[i]` soit la liste d'adjacence du sommet i avec $0 \leqslant i \leqslant n-1$.

On peut étendre la notion de liste d'adjacence d'un sommet i pour un graphe pondéré en la construisant comme une liste de tuples (j, w_{ij}) où j est un voisin (resp. un successeur) de i et où w_{ij} est le poids de l'arête (resp. de l'arc) $[i, j]$.

Exemples :

Trouver les listes d'adjacence des trois graphes donnés ci-dessous :



Exercices :

1. Écrire une fonction `matrice_to_liste_adj(M)` qui prend la matrice d'adjacence `M` d'un graphe (matrice implémentée comme une liste de listes) et qui renvoie la liste d'adjacence du graphe.
2. Écrire une fonction `liste_to_matrice_adj(Ladj:list)` qui prend la liste d'adjacence `Ladj` d'un graphe et qui renvoie la matrice d'adjacence du graphe (implémentée comme une liste de listes).
3. Écrire une fonction `deg_sortant(i:int,Ladj:list)` qui renvoie le degré sortant d'un sommet `i` dans un graphe orienté (non pondéré) implémenté par une liste d'adjacence `Ladj`.
4. Écrire une fonction `pred(i:int,Ladj:list)` qui renvoie la liste des sommets prédécesseurs du sommet i.
5. Écrire une fonction `nbre_arcs(i:int,Ladj:list)` qui renvoie le nombre d'arcs d'un graphe orienté (non pondéré) implémenté par une liste d'adjacence `Ladj`.

III. Chemin dans un graphe

1) Définition

Définition. *Chemin*

Soient $G = (S, A)$ un graphe orienté ou non, s et t deux sommets de G. Un *chemin* c de longueur k allant de s à t est une **suite finie** de sommets $c =< s_0, \dots, s_k >$ vérifiant :

$$s_0 = s \ ; \ s_k = t \quad \text{et} \quad \forall\, i \in \{1, \dots, k\},\ [s_{i-1}, s_i] \in A$$

Le sommet s est alors *l'origine* du chemin et t en est *l'extrémité*.

On accepte les chemins du type $< s_0 >$ ne contenant qu'un seul sommet s_0. Dans ce cas, $s = t = s_0$. Par définition, sa longueur est nulle.

Remarques et notations :

- On dit que le chemin c **contient** les sommets $s_0, s_1, \dots, s_k$ et que, pour $i \in \{0, ..., k\}$, s_i est le sommet de rang i contenu dans le chemin c.
- De même on dit que le chemin c **contient** les arêtes (resp. les arcs) $[s_0, s_1], [s_1, s_2], \dots, [s_{k-1}, s_k]$. Un chemin de longueur k contient donc exactement k arêtes ou arcs.
- Dans un graphe non orienté un chemin du type $< s_0 >$ ne contient aucune arête. Si le graphe est orienté alors le chemin $< s_0 >$ peut soit contenir 0 arc, soit un seul arc, nécessairement de la forme (s_0, s_0).
- Si $c =< s_0, \dots, s_k >$ est un chemin allant de $s = s_0$ à $t = s_k$, il nous arrivera par commodité de le noter :
$$s \xrightarrow{c} t$$

Définitions supplémentaires :

- Un chemin est dit *simple* lorsqu'il ne passe pas deux fois par le même sommet, c'est-à -dire lorsque tous les sommets s_i contenus dans c sont deux à deux distincts.
- Un *cycle* est un chemin **de longueur non nulle** dont les sommets de départ et d'arrivée sont identiques : $s_0 = s_k$. On dit qu'un graphe est *acyclique* lorsqu'il ne possède pas de cycle.
- Dans un **graphe non orienté**, si $c =< s_0, \dots, s_k >$ est un chemin allant de s à t, alors $c' =< s_k, \dots, s_0 >$ est un chemin allant de t à s, appelé *chemin inverse de* c.

2) Relation d'accessibilité

Définition 1. *Accessibilité*

Soit $G = (S, A)$ un graphe orienté ou non. Un sommet t est dit *accessible* depuis un sommet s lorsqu'il existe au moins un chemin c allant de s à t. On définit alors la *relation d'accessibilité* :

$$\forall\, (s, t) \in S^2,\ s \mathscr{R}_A\, t \iff t \text{ est accessible depuis } s$$

Exercice : montrer que si G est un graphe non orienté la relation d'accessibilité est une relation d'équivalence. Que se passe-t-il si le graphe est orienté ?

Définition 2. *Connexité dans un graphe non-orienté*

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

- Toute classe d'équivalence $\mathscr{C}$ de la relation d'accessibilité $\mathscr{R}_A$ est appelée *composante connexe* de G.
- On dit que G est *connexe* si et seulement s'il ne possède qu'une seule composante connexe, c'est-à-dire lorsqu'il existe (au moins) un chemin reliant n'importe lequel de ses sommets à n'importe quel autre sommet du graphe.

Exemple : quelles sont les composantes connexes du graphe représenté sur la Figure 1 ?

Définition 3. *Graphe orienté fortement connexe*

On dit qu'un graphe orienté est *fortement connexe* si et seulement si, entre deux sommets quelconques du graphe, il existe toujours (au moins) un chemin.

Remarque :

Dans un graphe orienté, la relation d'accessibilité n'est pas une relation d'équivalence. En conséquence, la notion de composante connexe n'a plus de sens pour ce graphe.

3) Distance entre deux sommets

Définition. *Distance entre deux sommets*

Soit $G = (S, A)$ un graphe orienté ou non. Si t est un sommet accessible depuis un sommet s alors l'ensemble $C(s, t)$ des chemins allant de s à t n'est pas vide. Dans ce cas on appelle *distance* entre s et t l'entier défini par :

$$d(s, t) = \min_{c \in C(s,t)} \ell(c)$$

où $\ell(c)$ est la longueur du chemin c.

Remarques :

- d n'est pas véritablement une distance au sens mathématique du terme. Quelle est la propriété caractéristique des distances qui n'est pas nécessairement vérifiée ? Que se passe-t-il dans le cas d'un graphe non orienté ?
- Un chemin c reliant deux sommets s et t est dit de *longueur minimale* si et seulement si $\ell(c) = d(s, t)$.
- Si deux sommets s et t ne sont reliés par aucun chemin on convient de poser $d(s, t) = +\infty$.

Exercice* :

À l'aide d'un raisonnement par récurrence sur l'ordre n du graphe, montrer que dans un graphe non orienté et connexe, le nombre de sommets et le nombre d'arêtes vérifient : $|A| \geqslant |S| - 1$.

IV. Arbres

Les arbres sont des structures très utiles en informatique.

1) Définitions

Définition 1. *Arbre libre*

Un *arbre libre* est un graphe $G_A = (S, A)$ non orienté, connexe et ne contenant pas de cycles. Les éléments de S sont appelés *nœuds* de l'arbre (plutôt que sommets).

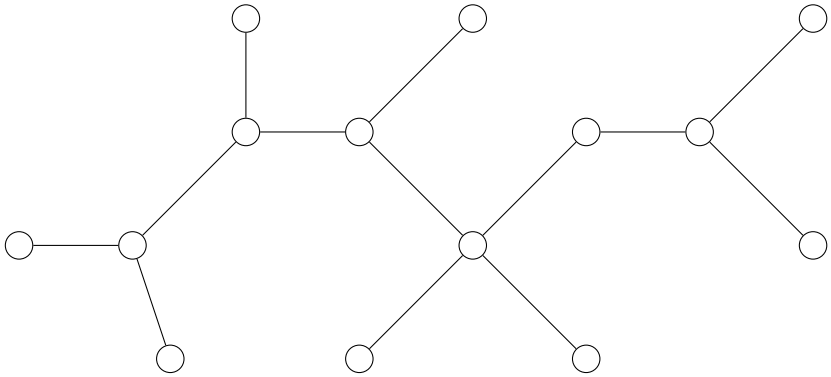


FIGURE 4 – Un arbre libre

Rappelons qu'on dit qu'un chemin est simple lorsqu'il ne contient pas deux fois le même sommet.

Théorème fondamental :

Soit $G = (S, A)$ un graphe non orienté. Les propriétés suivantes sont équivalentes :

1. G est un arbre libre.
2. Entre deux nœuds quelconques s et t de S, il existe un et un seul chemin simple.
3. G est connexe mais si on enlève une arête de A alors le graphe résultant n'est plus connexe.
4. G est connexe et $|A| = |S| - 1$.
5. G est acyclique et $|A| = |S| - 1$.
6. G est acyclique mais si on ajoute une arête à A on crée nécessairement un cycle.

Demonstration sur feuille à part.

Définition 2. *Arbre enraciné*

Un *arbre enraciné* est un arbre libre dont on a particularisé un nœud appelé racine de l'arbre.

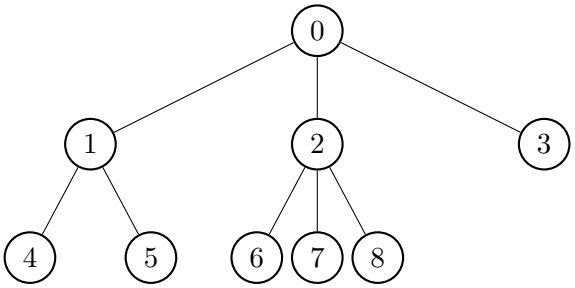


FIGURE 5 – Un arbre enraciné

Définition 3. *Relations de parenté dans un arbre enraciné*

Pour tout nœud x d'un arbre enraciné, il existe un chemin simple unique de la racine r à x.

- Tout nœud y sur ce chemin est un *ancêtre* de x et x est un *descendant* de y.
- L'avant-dernier nœud y sur l'unique chemin simple reliant r à x est le parent de x, et x est un enfant de y. L'arité d'un nœud est le nombre de ses enfants.
- Un nœud sans enfant est une *feuille*. Un nœud d'arité strictement positive est appelé *nœud interne*.
- La *hauteur* d'un arbre est la longueur maximale (comptée en nombre d'arêtes) d'un chemin simple reliant sa racine r à une feuille. Un arbre réduit à un seul nœud est de hauteur 0.

Définition 4. *Étiquettes d'un arbre enraciné*

Un arbre enraciné est dit *étiqueté* lorsqu'à chaque nœud on associe une information appelée *étiquette*.

Pratique en python :

Pour étiqueter un arbre, on convient que l'ensemble S de ses nœuds sont les entiers $\{0, 1, ..., n-1\}$ (arbre d'ordre n) et on définit :

- soit une liste `Letiquettes` telle que `Letiquette[i]` contient l'information concernant le nœud i ;
- soit un dictionnaire `Detiquettes` dont les clés sont les entiers i correpsondant aux nœuds et tel que `Detiquette[i]` contient l'information concernant le nœud i ;

2) Implémentation d'un arbre en Python

Dans tout ce qui suit, les nœuds d'un arbre d'ordre n seront des entiers appartenant à l'ensemble $[\![0, n-1]\!]$.

Pour implémenter un arbre enraciné on peut commencer par utiliser une liste d'adjacence `Ladj` (comme pour un graphe). Cependant, pour augmenter l'efficacité, on préfère disposer :

- d'une liste `parent` telle que `parent[i]` soit le parent du nœud i. Par convention pour la racine `parent[r] = None` ;
- d'une liste de listes `enfants` où `enfants[i]` est la liste des enfants du nœud i ;
- éventuellement d'une liste `etiquettes` telle que `etiquettes[i]` contient des informations relatives au nœud i.

Exercices :

1. Écrire une fonction `construire_arbre(r:int,Ladj:list)` qui prend comme paramètre la racine d'un arbre et sa liste d'adjacence et qui renvoie les listes `parents` et `enfants`
2. On suppose qu'on dispose de la liste `enfants` pour un arbre donné. Écrire une fonction `est_feuille(p:int,enfants:list)` qui prend comme paramètres un nœud p et la liste `enfants` d'un arbre et qui renvoie `True` si le nœud p est une feuille et `False` dans le cas contraire.

V. Parcours d'un graphe

Les parcours de graphes sont à la base de nombreux algorithmes sur les graphes. Ils vont nous permettre de calculer des plus courts chemins, tester la connexité, tester l'existence d'un cycle (circuit) dans un graphe, etc...

Dans le but de parcourir des graphes, il nous faut définir deux outils très utilisés en informatique : la *pile* et la *file*.

1) Structures de Pile et de File

a) Les piles

Une pile est un objet informatique qui permet de stocker des éléments (entiers, flottants, chaines de caractères, ...) en suivant le principe *LIFO* : last in, first out : dernier entré, premier sorti.

Visuellement, on peut se représenter une pile comme une pile du langage courant (figure 6). L'élément inséré en dernier se trouve tout en haut et s'appelle le *sommet de la pile*. C'est le **seul élément accessible** ; pour accéder à un élément situé en dessous du sommet, il faut commencer par sortir les éléments placés au dessu de lui.

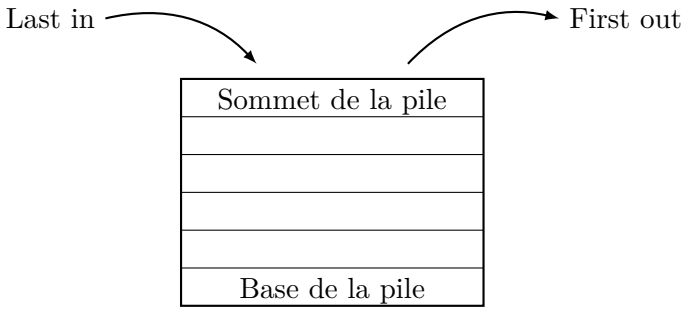


FIGURE 6 – Représentation d'une pile

Dans le langage Python, une pile est implémentée de façon efficace au moyen d'une liste. Le sommet de la pile est le dernier élément de la liste (celui dont l'indice est le plus élevé).

- Un nouvel élement est inséré dans la pile par la méthode `append` ;
- Le sommet de la pile est sorti (et récupéré) par la méthode `pop`.

b) Les files

Une file est un objet informatique qui permet de stocker des éléments (entiers, flottants, chaines de caractères, ...) en suivant le principe *FIFO* : first in, first out : premier entré, premier sorti. Ceci peut être représenté visuellement par le schéma de la figure 7.

Dans le langage Python, on utilise aussi des listes pour implémenter une file. Un nouvel élément est inséré à gauche de la liste (il devient donc l'élément d'indice 0) et les éléments sont sortis par la droite grâce à la méthode `pop`.

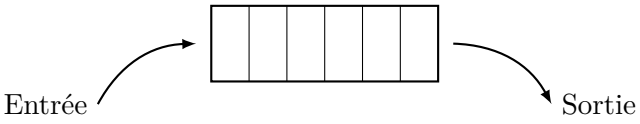


FIGURE 7 – Représentation d'une file

Exemple :

Soit F une file (liste Python). Écrire une fonction `inserer(F, e)` qui insère l'élément `e` dans `F`.

Cependant la complexité temporelle de cette fonction n'est pas excellente (l'opération de concaténation à gauche d'une liste est en $\mathcal{O}(n)$, où n est la taille de la liste).

Le langage Python possède un module `collections` qui contient une fonction `deque` qui permet de créer et de gérer une file de façon efficace. On dispose des méthodes :

- `append` et `pop` : insertion à droite et sortie à droite
- `appendleft` et `popleft` : insertion à gauche et sortie à gauche

Ce module permet de manipuler des files en n'utilisant que `appendleft` et `pop`. Ces opérations se font alors *à temps constant*.

Exemple :

2) Parcours d'un graphe

Soit $G = (S, A)$ un graphe (orienté ou non) d'ordre n. On suppose que l'ensemble des sommets du graphe est $S = \{0, \ldots, n-1\}$. Partant d'un sommet $s \in S$ appelé *sommet source* on souhaite parcourir le graphe de façon à découvrir tous les sommets accessibles à partir de s.

Avec les algorithmes de parcours d'un graphe, tous les sommets peuvent être dans un des deux états suivants (états qui s'excluent mutuellement) : "blanc" ou "gris" [1].

- Un sommet est "blanc" tant qu'il n'a pas été rencontré (découvert) par l'algorithme.
- Dès qu'un sommet est découvert, il passe à l'état "gris". Cela permet de marquer le sommet et, si l'algo. le rencontre à nouveau plus tard, il saura qu'il a déjà été découvert avant.
- Au début, tous les sommets sont "blancs", sauf le sommet source s qui est "gris".

Lorsque, étant arrivé à un sommet u, l'algo. poursuit son chemin vers les voisins (successeurs) de u, deux situations peuvent survenir :

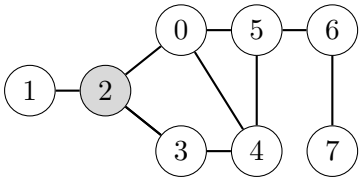
L'algo. de parcours utilise une liste `parent` de taille n telle que, dans la seconde situation, `parent`$[v] = u$. De cette manière, l'algorithme construit un arbre (informatique) ayant la structure d'un *arbre généalogique* où chaque sommet du graphe sera *l'enfant* d'un autre sommet. L'origine de cet arbre est le sommet source s ; on dit que s est la **racine** de l'arbre.

Il est clair que la structure de cet arbre dépend de la manière dont l'algo. parcourt le graphe. En pratique, deux types de parcours sont utilisés :

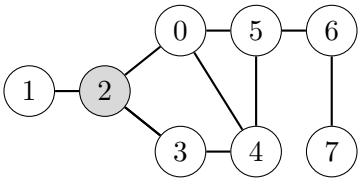
- le *parcours en largeur*. À partir d'un sommet u atteint, l'algo. explore tous les voisins (successeurs) de u pas encore découverts ; ceux-ci deviennent les enfants de u. Il poursuit ensuite son chemin en explorant les voisins (successeurs) de chacun des enfants, et ainsi de suite.
- le *parcours en profondeur*. À partir d'un sommet u atteint, dès que l'algo. touve un voisin (successeur) v pas encore découvert, il en fait l'enfant de u, puis il continue son chemin à partir v sans chercher à explorer les autres voisins (successeurs) de u. Dans ce type de parcourt, l'algo. cherche à aller le plus loin possible dans le graphe. Dès qu'il ne peut plus avancer, il "revient sur ses pas" jusqu'à trouver un chemin qu'il n'a pas encore emprunté et il recommence son parcourt en profondeur à partir de ce point.

1. On pourrait bien entendu choisir True ou False, 0 ou 1, ou encore n'importe quelle paire de symboles pour désigner les deux états possibles

Exemple 1 : parcours en largeur à partir du sommet 2

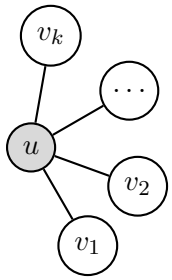


Exemple 2 : parcours en profondeur à partir du sommet 2



3) Parcours en largeur d'un graphe implémenté par une liste d'adjacence

Dans un parcours en largeur, lorsque l'algo. a atteint un sommet u il commence par découvrir tous ses enfants v_1, v_2, ..., v_k. Il repart ensuite de v_1 pour découvrir ses enfants, puis il traite v_2, ..., et ainsi de suite jusqu'à v_p.



Comment garder en mémoire la liste des sommets dans l'ordre dans lequel ils ont été découverts afin de pouvoir ensuite les traiter dans ce même ordre ?

L'algorithme utilise :

- une file `F` dans laquelle il place les différents sommets au fur et à mesure de leur découverte ;
- une liste `etat` de taille n (ordre du graphe) qui indique l'état de chaque sommet : blanc ou gris. `etat[i]` contient donc l'état du sommet s_i à un moment donné : cet état peut être une des deux chaînes de caractères : `"blanc"` ou `"gris"` ;
- une liste `parent` telle que `parent[i]` indique le parent du sommet s_i. Par convention, le parent du sommet source est `None` et il est est de même du parent de tout sommet qui n'a pas été découvert. *Cette liste est facultative et n'apparaît pas dans certaines implémentations de l'algorithme* ;
- une liste `distance` telle que `distance[i]` contient la distance (comptée en nombre d'arêtes ou d'arcs) de la source s au sommet s_i. Par convention `distance[s]` contient 0 et tant que le sommet s_i n'a pas été découvert `distance[i]` $= +\infty$. *Cette liste est facultative et peut être omise dans certaines implémentations de l'algorithme.*

Outil : propriétés et codage de $+\infty$ dans le langage python :

Algo. parcours en largeur : on suppose que le graphe G est implémenté par un liste d'adjacence `Ladj` définie comme variable globale dans le programme.

```
from collections import deque

n = len(Ladj)
etat = [ "blanc" for i in range(n) ]
parent = [ None for i in range(n) ]
distance = [ float("inf") for i in range(n) ]

def parcoursL(Ladj,s,etat,parent,distance) :
   F = deque()
   etat[s], distance[s] = "gris", 0
   F.appendleft(s)

   while len(F) != 0 :
      u = F.pop()
      for v in Ladj[u] :
         if etat[v] == "blanc" :
            etat[v] = "gris"
            parent[v] = u
            distance[v] = distance[u] + 1
            F.appendleft(v)
```

Cet algorithme est valable aussi bien pour des graphes non orientés que pour des graphes orientés.

4) Parcours en profondeur depuis un sommet source

Le parcours en profondeur se prête bien à un programmation récursive mais on peut aussi en donner une version itérative. L'algorithme utilise :

- une liste `etat` de taille n (ordre du graphe) qui indique l'état de chaque sommet : blanc ou gris (càd pas découvert ou découvert) ;
- une liste `parent` telle que `parent[i]` indique le parent du sommet s_i (cette liste est faculative) ;

La liste `distance` n'est pas utilisée ici car elle n'a pas de sens (pourquoi ?). Comme pour le parcours en largeur précédent on suppose que le graphe est représenté par une liste d'adjacence `Ladj` qui est une variable globale du programme.

Cet algorithme s'applique aussi bien aux graphes orientés qu'aux graphes non orientés.

a) Version itérative

```
n = len(Ladj)
etat = [ "blanc" for i in range(n) ]
parent = [ None for i in range(n) ]

def parcoursPP(Ladj,s,etat,parent) :
   etat[s] = "gris"
   P = []
   P.append(s)
   while len(P) != 0 :
      u = P.pop()
      for v in Ladj[u] :
         if etat[v] == "blanc" :
            etat[v] = "gris"
            parent[v] = u
            P.append(v)
```

b) Version récursive

```
n = len(Ladj)
etat = [ "blanc" for i in range(n) ]
parent = [ None for i in range(n) ]

def parcoursPP(Ladj,u,etat,parent) :
   etat[u] = "gris"
   for v in Ladj[u] :
      if etat[v] == "blanc" :
         parent[v] = u
         parcoursPP(Ladj,v,etat,parent)
```

L'algorithme récursif est lancé par un premier appel à `parcoursPP(Ladj,s,etat,parent)`

VI. Chemins de poids minimal

Dans toute cette section, les sommets d'un graphe $G = (S, A)$ (orienté ou non) d'ordre n et l'ensemble des sommets est $S = \{0, ..., n-1\}$.

1) Définitions

Soit un graphe pondéré $G = (S, A, w)$ **orienté**. On rappelle que l'application de pondération $w : A \longrightarrow \mathbb{R}$, $a \longmapsto w(a)$ où $w(a)$ est le poids de l'arc $a \in A$. En pratique $w(a)$ peut représenter une distance, une durée, un coût, un nombre de communications, etc ... entre le sommet à l'origine et le sommet à l'extrémité de l'arc a.

Définition 1. *Poids d'un chemin*

Dans un graphe pondéré et orienté $G = (S, A, w)$, on appelle *poids d'un chemin* $c =< s_0, ..., s_k >$ qui relie un sommet $s = s_0$ à un sommet $t = s_k$, la somme :

$$W(c) = \sum_{i=1}^{k} w(s_{i-1}, s_i)$$

Dans le cas d'un chemin de longueur nulle, de la forme $c =< s_0 >$ on convient que $W(c) = 0$.

Remarque :

On sait que dans un graphe orienté il peut exister des boucles, c'est à dire des arcs de la forme (s, s). Dans ce cas, on supposera toujours que l'application de pondération w leur attribue un poids nul : $w(s, s) = 0$.

Définition 2. *Chemin de poids minimal entre deux sommets d'un graphe*

Soient s et t deux sommets d'un graphe pondéré et orienté $G = (S, A, w)$. Deux cas se présentent :

1. Si t est accessible depuis s alors l'ensemble $C(s, t)$ des chemins ayant pour origine s et pour extrémité t n'est pas vide et on pose :

$$\delta(s, t) = \min_{c \in C(s,t)} W(c)$$

   Tout chemin c reliant s à t et vérifiant $W(c) = \delta(s, t)$ est appelé *chemin de poids minimal* reliant s à t.

2. Si t n'est pas accessible depuis s, alors on convient que $\delta(s, t) = +\infty$

Dans les deux cas $\delta(s, t)$ est le *poids minimal* entre l'origine s et l'extrémité t.

Cohérence de la définition du poids minimal

1. Peut-on envisager une recherche de chemins de poids minimal entre deux sommets s'il existe des cycles de poids strictement négatif ?

   S'il existe un cycle c_0 de poids strictement négatif allant d'un sommet u_0 et revenant à ce sommet, alors pour tout chemin $c =< s_0, \ldots, s_k >$ qui contient u_0 on peut, en posant $c_1 =< s_0, \ldots u_0 >$ et $c_2 =< u_0, \ldots, s_k >$, trouver des chemins alllant de s_0 à s_k ayant des poids aussi petits qu'on veut. Par exemple :

$$s_0 \xrightarrow{c_1} u_0 \xrightarrow{c_0} u_0 \xrightarrow{c_0} u_0 \ldots \xrightarrow{c_0} u_0 \xrightarrow{c_2} s_k$$

en un chemin de poids arbitrairement petit et négatif. L'ensemble $W(s,t) = \{ W(c) \mid c \in C(s,t) \}$ ne possède donc pas de minorant et la notion de chemin minimal allant de s à t n'a pas de sens.

C'est pour cela que nous ferons l'une des deux hypothèses suivantes :

Il n'existe aucun cycle de poids strictement négatif dans l'arbre $G = (S, A)$ étudié (hypothèse forte) ou bien, si s est le sommet origine, pour tout sommet t accessible depuis s, il n'existe aucun cycle de poids strictement négatif allant de s à t (hypothèse faible).

2. Peut-on étudier des chemins de poids minimal dans des graphes non orientés ?

   Un problème surgit lorsque certaines arêtes ont un poids négatif. Ces graphes contiennent alors automatiquement des cycles (il suffit de faire des aller-retours entre deux sommets adjacents), ce qui fait qu'un chemin de poids minimal n'y est pas correctement défini. *C'est pour cela qu'on restreint l'étude des chemins de poids minimal à des graphes orientés. Bien sûr, dans le cas où les poids des arcs sont tous positifs, on peut étendre cette étude aux graphes non orientés.*

3. Peut-il exister des cycles de poids strictement positifs dans un chemin de poids minimal ?

   La réponse est négative. En effet, raisonnons par l'absurde en considérant un chemin $c =< s_0, \ldots, s_k >$ de poids minimal et en supposant qu'il contient un cycle de poids strictement positif. Il existe donc un sous-chemin de c que l'on notera $c_{ij} =< s_i, \ldots, s_j >$ avec $0 \leqslant i \leqslant j \leqslant k$, $s_i = s_j$ et $W(c_{ij}) > 0$. En décomposant le chemin c en :

   $$c : s_0 \xrightarrow{c_{0i}} s_i \xrightarrow{c_{ij}} s_j \xrightarrow{c_{jk}} s_k$$

   on constate que son poids est $W(c) = W(c_{0i}) + W(c_{ij}) + W(c_{jk})$.

   Or on peut construire le chemin

   $$c' : s_0 \xrightarrow{c_{0i}} s_i = s_j \xrightarrow{c_{jk}} s_k$$

   Le poids de ce chemin est alors :

   $$W(c') = W(c) - W(c_{ij}) < W(c)$$

   ce qui montre que c n'est pas de poids minimal.

   CQFD.

4. Peut-il exister des cycles de poids nul dans un chemin de poids minimal ?

   À priori rien ne s'y oppose mais si le cas se présente, on peut éliminer ces cycles du chemin sans rien changer à son poids.

En conclusion, s'il existe un chemin c de poids minimal allant d'un sommet s à un sommet t, on peut toujours s'arranger pour que ce chemin soit simple : chaque sommet contenu dans c n'y est présent qu'une seule fois (autrement dit, c ne contient pas de cycle).

2) Propriété fondamentale d'un chemin de poids minimal

Proposition

Soit $c =< s_0, \ldots, s_k >$ un chemin allant de $s = s_0$ à un sommet $t = s_k$, de poids minimal $W(c) = \delta(s,t)$. Alors tout sous-chemin $c_{ij} =< s_i, \ldots, s_j >$ de c, avec $0 \leqslant i < j \leqslant k$ est un chemin de poids minimal allant de s_i à s_j.

Preuve :

Raisonnons par l'absurde en supposant que c_{ij} n'est pas de poids minimal. Cela signifie qu'il existe un autre chemin c'_{ij} allant de s_i à s_j, de poids strictement inférieur à celui de c_{ij}, . Alors le chemin :

$$s_0 \xrightarrow{c_{0i}} s_i \xrightarrow{c'_{ij}} s_j \xrightarrow{c_{jk}} s_k$$

a un poids $W = W(c_{0i}) + W(c'_{ij}) + W(c_{ik}) < W(c_{0i}) + W(c_{ij}) + W(c_{ik}) = W(c)$, ce qui est impossible puisque c est un chemin de poids minimal.

CQFD.

Corollaire

Soit $c =< s_0, \ldots, s_k >$ un chemin allant de $s = s_0$ à un sommet $t = s_k$, de poids minimal $W(c) = \delta(s,t)$. Alors pour tout sommet s_i de rang i, $1 \leqslant i \leqslant k$, contenu dans c, on a :

$$\delta(s, s_i) = \delta(s, s_{i-1}) + w(s_{i-1}, s_i) \quad (*)$$

Preuve :

- Si $i = 1$, alors en convenant que $\delta(s,s) = 0$, l'équation (*) devient $\delta(s, s_1) = w(s, s_1)$, ce qui est manifestement vrai puisque $< s_0, s_1 >$ est le chemin de poids minimal allant de s_0 à s_1 en tant que sous-chemin d'un chemin de poids minimal.
- Si $i > 1$ alors on décompose : $s_0 \xrightarrow{c_{0,i}} s_i$ en $s_0 \xrightarrow{c_{0,i-1}} s_{i-1}$ suivi de $s_{i-1} \xrightarrow{c_{i-1,i}} s_i$. Les deux premiers sont des chemins de poids minimaux en tant que sous-chemins d'un chemin de poids minimal. On a donc :

  $$\delta(s, s_i) = \delta(s, s_{i-1}) + w(s_{i-1}, s_i)$$

  CQFD.

3) Recherche de chemins de poids minimal à origine unique

On cherche dans cette partie à déterminer les chemins de poids minimal depuis un sommet origine $s \in S$ **donné** vers chaque sommet $t \in S$ du graphe. On appelle cela le problème de la recherche du chemin de poids minimal **à origine unique**.

On considère un graphe orienté pondéré $G = (S, A, w)$ ne contenant pas de cycle de poids strictement négatif. De l'étude précédente il ressort que :

Proposition

À tout sommet $t \in S$ on peut associer une grandeur $\delta(s,t) \in \mathbb{R} \cup \{+\infty\}$ qui est le poids minimal entre l'origine s et le sommet t. Elle vérifie les deux propriétés :

1. t est accessible depuis s si et seulement si $\delta(s,t) \in \mathbb{R}$.
2. t n'est pas accessible depuis s si et seulement si $\delta(s,t) = +\infty$

Preuve sur feuille à part.

a) Cas où w est une application constante

Commençons par le cas particulier où $w : a \longmapsto w(a)$ est une *application constante*. Dans ce cas, $\forall\, a \in A,\ w(a) = w_0 \in \mathbb{R}_+^*$. Soit c un chemin partant de s et aboutissant à t.

Dans ce cas : $W(c) = w_0 \times k = w_0 \times \ell(c)$. Le poids de c est tout simplement proportionnel à la longueur de c. Le poids minimal $\delta(s,t)$ est donc proportionnel à la distance $d(s,t)$ entre s et t.

Afin de déterminer ce poids minimal, il suffit d'effectuer un **parcours en largeur** du graphe en démarrant depuis le sommet s. En effet, lors de ce type de parcours la liste `distance` donne la longueur minimale entre s et n'importe quel sommet t accessible depuis s. On a :

$$\texttt{distance[i]} = \begin{cases} d(s,s_i) & \text{si} \quad s_i \text{ accessible depuis } s \\ +\infty & \text{si} \quad s_i \text{ n'est pas accessible depuis } s \end{cases}$$

On aura donc :

$$\delta(s,t) = \begin{cases} w_0 \times \texttt{distance[i]} & \text{si} \quad s_i \text{ accessible depuis } s \\ +\infty & \text{si} \quad s_i \text{ n'est pas accessible depuis } s \end{cases}$$

b) Algorithme de Dijkstra

L'algorithme de Dijkstra [2] (publié par E. Dijkstra en 1959) et que nous noterons plus simplement algo. D par la suite, permet de calculer le poids minimal d'un chemin entre deux sommets d'un graphe tel que $\forall\, a \in A,\ w(a) \in \mathbb{R}_+$, c'est à dire dans le cas où les **poids sont tous positifs**. C'est un algorithme glouton.

L'ensemble des sommets est $S = \{0, \ldots, n-1\}$.

On pourrait implémenter le graphe uniquement par une liste d'adjacence `Ladj`, mais, par souci de simplicité, on va créer en plus une matrice `w` (implémentée comme une liste de listes) qui contient l'information des poids des arcs de sorte que :

$$\texttt{w[i][j]} = \begin{cases} w(i,j) & \text{si} \quad (i,j) \in A \quad \text{et} \quad i \neq j \\ +\infty & \text{si} \quad (i,j) \notin A \end{cases}$$

- L'algo. D parcourt le graphe à partir d'un sommet source s et utilise une liste `poidsChemin`. Le but est que lorsque l'algorithme se termine, `poidsChemin[i]` contienne le poids minimal entre s et i (éventuellement égal à $+\infty$ si i n'est pas accessible depuis s).
  - Au début, la liste `poidsChemin` est initialisée comme ci-dessous :

    ```
    poidsChemin = [ float("inf") for i in range(n) ]
    poidsChemin[s] = 0
    ```

  - dès que l'algorithme trouve un chemin entre s et i de poids inférieur à celui stocké dans `poidsChemin[i]` il remplace l'ancienne valeur par la nouvelle.

- L'algo. D gère aussi une liste `parent`. Au début cette liste est initialisée comme ci-dessous :

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

  Si un sommet j est découvert comme successeur de i alors `parent[j]` contient `i`. Cependant, cette liste peut encore varier au cours du déroulement de l'algo. D si on trouve un chemin menant de s à j dont le poids est plus petit que celui qui passe par i.

2. E. Dijkstra (1930-2002) : informaticien néerlandais.

- L'algo. D utilise une liste `sommetsTraites` qui n'est pas indispensable pour son fonctionnement mais qui va nous servir à prouver la justesse de cet algorithme.
- Contrairement à l'algorithme de parcours en largeur, l'algo D n'utilise pas de file pour gérer l'exploration des sommets ; à la place, il utilise une structure appelée *file de priorité* `F` où on stocke initialement tous les sommets du graphe.

  À chaque étape de l'algo. D on n'extrait que le sommet t_0 tel que `poidsChemin[t0]` soit le plus petit parmi tous les $t \in F$ et on supprime t_0 de F.

  Il faut donc écrire une fonction auxilliaire `extraire_min(F,poidsChemin)` qui détermine le sommet $t_0 \in F$ ayant la plus petite valeur de `poidsChemin[t0]`, élimine ce sommet de `F` et renvoie `t0`.

  Exercice : écrire cette fonction `extraire_min(F,poidsChemin)`. On pourra utiliser la méthode de liste `L.remove(x)` qui élimine l'élément `x` de la liste `L`.

- Enfin, l'algo. D utilise une fonction `recalculer(i,j,w,poidsChemin, parent)` qui réactualise les listes `poidsChemin` et `parent`.

```
def recalculer(i,j,w,poidsChemin,parent) :
   if poidsChemin[j] > poidsChemin[i] + w[i][j] :
      poidsChemin[j] = poidsChemin[i] + w[i][j]
      parent[j] = i
```

Initialisation : on suppose qu'une liste d'adjacence `Ladj` implémentant le graphe a été définie comme variable globale.

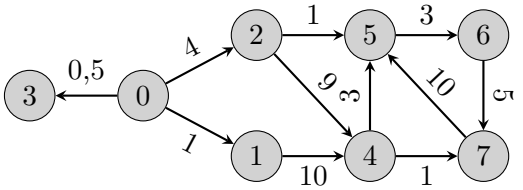
```
n = len(Ladj)
s = ...  # choisir s entre 0 et n-1
poidsChemin = [ float("inf") for i in range(n) ]
poidsChemin[s] = 0
parent = [ None for i in range(n) ]
sommetsTraites = []
F = [ i for i in range(n) ]

def extraire_min(F,poidsChemin) :
   ...
```

```
def recalculer(i,j,w,poidsChemin,parent) :
    ...
```

Déroulement :

```
while F != [] :
  i = extraire_min(F,poidsChemin)
  for j in Ladj[i] :
    recalculer(i,j,w,poidsChemin,parent)
  sommetsTraites.append(i)
```

Exemple : considérons le graphe orienté pondéré suivant. On cherche à déterminer la liste des poids minimaux entre le sommet 0 (source) et chacun des autres sommets du graphe.



Compléter le tableau ci-après. La première colonne représente les itérations de la boucle `while`. On indiquera le contenu des listes `F`, `sommetsTraites` et `poidsChemin` à la fin de chaque itération (les numéros 0 ... 7 qui figurent dans le tableau sont les indices des éléments de la liste `poidsChemin`).

	`F`	`sommetsTraites`	`poidsChemin`							
			0	1	2	3	4	5	6	7
1										
2										
3										
4										
5										
6										
7										
8										

Preuves de la terminaison et de la justesse de l'algo. D sur feuille à part.

L'algorithme de Dijkstra marche avec des graphes orientés ou non (comme les poids sont positifs, cela ne pose pas de problème pour les cycles éventuels). Pour qu'il soit correct, il est important que la fonction de pondération w soit à valeurs positives. Dans le cas de valeurs négatives, l'algorithme de Bellman-Ford (hors programme ITC) permet de répondre à la question.