CHAPITRE 3

Introduction à la théorie des graphes

Plan du chapitre

Introduction

La notion de graphe est omniprésente en informatique moderne : ces derniers permettent de modéliser une structure composée d'objets et de relations entre ces objets.

Historiquement, les graphes font leur première apparition dans un article de Leonhard Euler de 1735 sur le célèbre problème des sept ponts de Königsberg : il s'agit de proposer un circuit touristique dans la ville de Königsberg en empruntant une et une seule fois tous ses ponts et en retournant au point de départ.

La théorie des graphes a connu un essor au cours du XIXème par l'intermédiaire du problème des quatre couleurs : il s'agit de déterminer le nombre minimal de couleurs différentes nécessaires pour colorier une carte sans que deux zones limitrophes n'aient la même couleur. Le théorème des quatre couleurs affirme que seulement quatre sont nécessaires. Le résultat fut conjecturé en 1852 par Francis Guthrie, mais il ne fut démontré qu'en 1976 par deux Américains Kenneth Appel et Wolfgang Haken. Les deux chercheurs ont ramené la résolution du problème à celles de 1478 cas particuliers, puis ils ont utilisé un ordinateur pour étudier ces différents cas. C'est la première fois qu'une démonstration exige l'utilisation d'un ordinateur.

De nombreux problèmes dans des domaines variés utilisent une représentation de structures par des graphes. Voici quelques exemples :

- les réseaux de transport (routiers, maritimes, aériens ou ferroviaires) où les objets sont certaines infrastructures (villes, ports, aéroports ou gares) et les relations sont les liaisons entre ces infrastructures ;
- le réseau internet dont les objets sont les différentes pages web existantes et les relations sont les liens hypertextes permettant d'aller d'une page web vers une autre ;
- les molécules en chimie peuvent se représenter avec des graphes dont les objets sont les atomes et les relations sont les liaisons entre ces atomes ;
- les réseaux sociaux dans lesquels les objets sont des personnes et les relations sont les relations d'amitié entre ces personnes ;
- certains jeux peuvent se représenter par un graphe où les sommets sont les configurations du jeu et les relations sont les coups valides qui permettent de passer d'une configuration à une autre.

Un des objectifs de l'étude des graphes en informatique est la conception d'algorithmes efficaces pour résoudre de manière effective des problèmes modélisés par des graphes (recherche du plus court chemin, problème du voyageur de commerce, etc.).

Partie I Généralités sur les graphes

I.A - Graphes non orientés

Définition (Graphe non orienté) : Un graphe non orienté G est un couple $G = (S, A)$ où S est un ensemble fini non vide et A une partie de $\mathscr{P}(S)$ dont les éléments sont de la forme $\{s_1, s_2\}$ avec $(s_1, s_2) \in S^2$. On dit que :

- les éléments de S sont appelés les sommets (ou les nœuds) du graphe G ;
- les éléments de A sont appelés les arêtes du graphe G.

Remarques 1 :

a) Si $a = \{s_1, s_2\}$ est une arête d'un graphe $G = (S, A)$, on dit que s_1 et s_2 sont les extrémités de a.

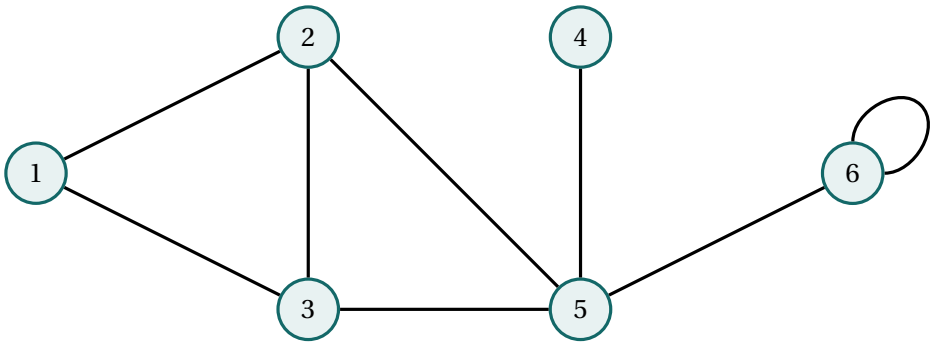
b) Si s_1 et s_2 sont les extrémités d'une arête a d'un graphe $G = (S, A)$, alors on dit que les sommets s_1 et s_2 sont adjacents ou que s_1 et s_2 sont incidents à l'arête a.

c) La définition retenue pour ce cours se limite aux graphes admettant au plus une arête entre deux sommets. Dans le cas contraire, on parle de multigraphe, mais nous n'étudierons pas ces derniers dans ce chapitre.

Exemple 1 : Considérons le graphe non orienté $G = (S, A)$ défini par

$$S = \{1,2,3,4,5,6\} \quad A = \big\{\{1,2\},\{1,3\},\{2,3\},\{2,5\},\{3,5\},\{4,5\},\{5,6\},\{6,6\}\big\}.$$

On peut représenter un graphe orienté de la manière suivante : on associe à chaque sommet de G un point du plan et on relie les points correspondant aux extrémités de chaque arête de G.



Définition (Ordre et taille d'un graphe non orienté) : Soit $G = (S, A)$ un graphe non orienté.

(i) L'ordre du graphe G est le nombre de sommets de G, i.e. $\text{Card}(S)$.

(ii) La taille du graphe G est le nombre d'arêtes de G, i.e. $\text{Card}(A)$.

Exemples 2 :

a) Le graphe non orienté de l'exemple 1 est d'ordre 6 et de taille 8.

b) On a l'encadrement $0 \leqslant \text{Card}(A) \leqslant \dfrac{\text{Card}(S)(\text{Card}(S)+1)}{2}$.

c) L'ordre du graphe associé au réseau ferroviaire français est supérieur à 2800.

d) Le graphe associé à certains réseaux sociaux ont un ordre en milliards et une taille en centaine de milliards.

Définition (Boucle) : Une boucle dans un graphe $G = (S, A)$ est une arête de la forme $a = \{s, s\}$ avec $s \in S$.

Remarque 2 : On dit qu'un graphe non orienté $G = (S, A)$ est simple s'il ne contient aucune boucle.

Exemple 3 : Le graphe non orienté de l'exemple 1 contient une boucle sur le sommet 6.

Définition (Degré d'un sommet dans un graphe non orienté) : Soit $G = (S, A)$ un graphe non orienté. Le degré d'un sommet $s \in S$, noté $d(s)$, est le nombre d'arêtes reliant s aux sommets de G, i.e.

$$d(s) = \text{Card}\big(\{a \in A \mid \exists s' \in S,\ a = \{s, s'\}\}\big) + \begin{cases} 0 & \text{si} \quad \{s,s\} \notin A \\ 1 & \text{si} \quad \{s,s\} \in A. \end{cases}$$

Remarques 3 :

a) Si la boucle $\{s, s\}$ est une arête de $G = (S, A)$, alors elle compte double dans le calcul du degré de s.

b) Si $G = (S, A)$ est un graphe non orienté, alors on a $\displaystyle\sum_{s \in S} d(s) = 2\,\text{Card}(A)$.

Exemple 4 : Le degré des sommets du graphe non orienté de l'exemple 1 sont respectivement

$$d(1) = 2, \qquad d(2) = 3, \qquad d(3) = 3, \qquad d(4) = 1, \qquad d(5) = 4, \qquad d(6) = 3.$$

Définition (Chaîne et cycle dans un graphe non orienté) : Soit $G = (S, A)$ un graphe non orienté.

(i) Une chaîne dans le graphe non orienté G est un $(n+1)$-uplet $c = (s_0, \dots, s_n)$ de sommets de G avec $n \in \mathbb{N}^*$ tel que pour tout $k \in [\![0, n]\!]$, l'élément $\{s_k, s_{k+1}\}$ est une arête de G. On dit que c est une chaîne de longueur n qui relie les sommets s_0 et s_n.

(ii) La chaîne $c = (s_0, \dots, s_n)$ est dite simple si les arêtes $\{s_0, s_1\}, \dots, \{s_{n-1}, s_n\}$ sont deux à deux distinctes.

(iii) La chaîne $c = (s_0, \dots, s_n)$ est dite élémentaire si les sommets $s_0, \dots, s_n$ sont deux à deux distincts.

(iv) On dit que la chaîne $c = (s_0, \dots, s_n)$ est un cycle si la chaîne c est simple et si $s_0 = s_n$.

Exemple 5 : Dans le graphe non orienté de l'exemple 1, l'élément $(1,2,5,6,6)$ est une chaîne et $(1,3,5,2,1)$ est un cycle.

Remarque 4 : S'il existe une chaîne c reliant les sommets s et s', alors il existe une chaîne élémentaire reliant les sommets s et s' : il suffit de retirer les cycles présents dans la chaîne c tant qu'il y en a.

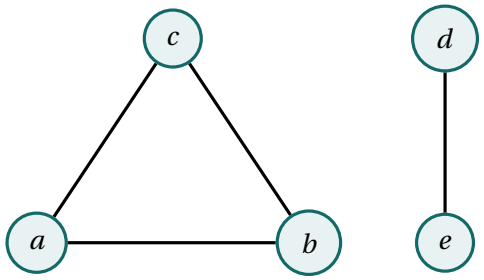
Exemple 6 : Dans le graphe non orienté de l'exemple 1, la chaîne $(1,2,3,5,2,3,5,6,6)$ permet de relier les sommets 1 et 6, mais on peut aussi le faire avec la chaîne élémentaire $(1,2,3,5,6)$.

Définition (Graphe non orienté connexe) : Soit $G = (S, A)$ un graphe non orienté. On dit que G est connexe si pour tout couple $(s_1, s_2) \in S^2$ avec $s_1 \neq s_2$, il existe une chaîne reliant s_1 à s_2.

Exemples 7 :

a) Le graphe non orienté de l'exemple 1 est connexe.

b) Le graphe non orienté $G = (S, A)$ avec $S = \{a, b, c, d, e\}$ et $A = \big\{\{a, b\}, \{a, c\}, \{b, c\}, \{d, e\}\big\}$ n'est pas connexe.



Exercice 1 : On considère les deux graphes non orientés ci-dessous.

(i) $G_1 = (S_1, A_1)$ avec $S_1 = \{a, b, c, d, e, f\}$ et $A_1 = \big\{\{a, d\}, \{a, e\}, \{b, d\}, \{b, f\}, \{c, d\}, \{c, e\}, \{d, e\}, \{d, f\}\big\}$.

(ii) $G_2 = (S_2, A_2)$ avec $S_2 = [\![1, 9]\!]$ et $A_2 = \big\{\{1, 5\}, \{2, 2\}, \{2, 4\}, \{2, 8\}, \{2, 9\}, \{3, 3\}, \{3, 5\}, \{5, 7\}\big\}$.

1. Représenter les graphes non orientés ci-dessus.
2. Pour chacun des graphes non orientés ci-dessus, préciser son ordre et sa taille, puis donner les degrés de ses sommets.
3. Les graphes ci-dessus sont-ils connexes ?

I.B - Graphes orientés

Définition (Graphe orienté) : Un graphe orienté G est un couple $G = (S, A)$ où S est un ensemble fini non vide et A une partie de S^2. On dit que :

- les éléments de S sont appelés les sommets (ou les nœuds) du graphe G;
- les éléments de A sont appelés les arcs du graphe G.

Remarques 5 :

a) Si $a = (s_1, s_2)$ est un arc d'un graphe $G = (S, A)$, on dit que s_1 est l'origine (ou le départ) de a et que s_2 est la cible (ou l'arrivée) de a.

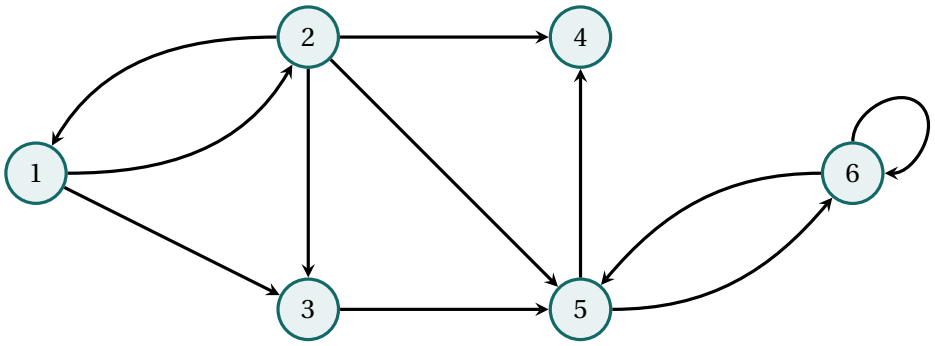
b) Si $a = (s_1, s_2)$ est un arc d'un graphe $G = (S, A)$, on dit également que s_1 est un prédécesseur de s_2 et que s_2 est un successeur de s_1.

c) La définition retenue pour ce cours se limite aux graphes admettant au plus un arc entre deux sommets. Dans le cas contraire, on parle de multigraphe, mais nous n'étudierons pas ces derniers dans ce chapitre.

Exemple 8 : Considérons le graphe orienté $G = (S, A)$ défini par

$$S = \{1,2,3,4,5,6\} \quad A = \big\{(1,2),(1,3),(2,1),(2,3),(2,4),(2,5),(3,5),(5,4),(5,6),(6,5),(6,6)\big\}.$$

On peut représenter un graphe orienté de la manière suivante : on associe à chaque sommet de G un point du plan et pour chaque arc $a = (s_1, s_2) \in A$, on trace une flèche du point représentant s_1 au point représentant s_2.



Définition (Ordre et taille d'un graphe orienté) : Soit $G = (S, A)$ un graphe orienté.

(i) L'ordre du graphe G est le nombre de sommets de G, i.e. $\text{Card}(S)$.

(ii) La taille du graphe G est le nombre d'arcs de G, i.e. $\text{Card}(A)$.

Exemples 9 :

a) Le graphe orienté de l'exemple 8 est d'ordre 6 et de taille 11.

b) On a l'encadrement $0 \leqslant \text{Card}(A) \leqslant \text{Card}(S)^2$.

c) L'ordre du graphe associé au réseau internet mondial a un ordre en milliards et une taille au moins en dizaine de milliards.

Définition (Boucle) : Une boucle dans un graphe $G = (S, A)$ est un arc de la forme $a = (s, s)$ avec $s \in S$.

Remarque 6 : On dit qu'un graphe orienté $G = (S, A)$ est simple s'il ne contient aucune boucle.

Exemple 10 : Le graphe orienté de l'exemple 8 contient une boucle sur le sommet 6.

Définition (Degré sortant et entrant d'un sommet dans un graphe orienté) : Soit $G = (S, A)$ un graphe orienté.

(i) Le degré sortant d'un sommet $s \in S$, noté $d_+(s)$, est le nombre d'arcs de G partant de s, i.e.

$$d_+(s) = \text{Card}\left(\left\{a \in A \mid \exists s' \in S,\ \ a = (s, s')\right\}\right).$$

(ii) Le degré entrant d'un sommet $s \in S$, noté $d_-(s)$, est le nombre d'arcs de G arrivant en s, i.e.

$$d_-(s) = \text{Card}\left(\left\{a \in A \mid \exists s' \in S,\ \ a = (s', s)\right\}\right).$$

Remarque 7 : Si $G = (S, A)$ est un graphe orienté, alors on a $\sum_{s\in S} d_+(s) = \sum_{s\in S} d_-(s) = \text{Card}(A)$.

Exemple 11 : Les degrés sortants et entrants des sommets du graphe orienté de l'exemple 8 sont respectivement

$$\begin{array}{llllll} d_+(1) = 2, & d_+(2) = 4, & d_+(3) = 1, & d_+(4) = 0, & d_+(5) = 2, & d_+(6) = 2, \\ d_-(1) = 1, & d_-(2) = 1, & d_-(3) = 2, & d_-(4) = 2, & d_-(5) = 3, & d_-(6) = 2. \end{array}$$

Définition (Chemin et circuit dans un graphe orienté) : Soit $G = (S, A)$ un graphe orienté.

(i) Un chemin dans le graphe orienté G est un $(n+1)$-uplet $c = (s_0, \ldots, s_n)$ de sommets de G avec $n \in \mathbb{N}^*$ tel que pour tout $k \in [\![0, n]\!]$, l'élément (s_k, s_{k+1}) est un arc de G. On dit que c est un chemin de longueur n allant de s_0 à s_n.

(ii) On dit que le chemin $c = (s_0, \ldots, s_n)$ est un circuit si $s_0 = s_n$.

Exemple 12 : Dans le graphe orienté de l'exemple 8, l'élément $(1, 2, 5, 6)$ est un chemin et $(5, 6, 5)$ est un circuit.

Remarque 8 : Dans un graphe orienté, l'existence d'un chemin de $s \in S$ à $s' \in S$ n'assure pas l'existence d'un chemin de s' à s en général.

Exemple 13 : Dans le graphe orienté de l'exemple 8, il existe un chemin de 1 vers 6, mais il n'existe pas de chemin de 6 vers 1.

Exercice 2 : On considère les deux graphes orientés ci-dessous.

(i) $G_1 = (S_1, A_1)$ avec $S_1 = \{1, 2, 3, 4\}$ et $A_1 = \{(2, 1), (2, 2), (3, 1), (3, 2), (3, 3), (3, 4), (4, 1), (4, 2), (4, 3)\}$.

(ii) $G_2 = (S_2, A_2)$ avec $S_2 = \{1, 2, 3, 4, 5, 6\}$ et $A_2 = \{(1, 2), (1, 4), (2, 3)(2, 5), (3, 5), (3, 6), (4, 1), (4, 2), (5, 4), (6, 4), (6, 6)\}$.

1. Représenter les graphes orientés ci-dessus.
2. Pour chacun des graphes orientés ci-dessus, préciser son ordre et sa taille, puis donner les degrés entrants et sortants de chacun de ses sommets.
3. On dit qu'un graphe orienté $G = (S, A)$ est fortement connexe si pour tout couple $(s_1, s_2) \in S^2$ avec $s_1 \neq s_2$, il existe un chemin de s_1 à s_2. Les graphes ci-dessus sont-ils fortement connexes ?

I.C - Ajouter des informations supplémentaires sur un graphe

Dans de nombreuses applications, il peut s'avérer utile d'ajouter des informations supplémentaires. Par exemple, si un graphe représente le réseau ferroviaire français, il est pertinent d'ajouter la longueur des voies entre les différentes gares à chaque arête de ce graphe.

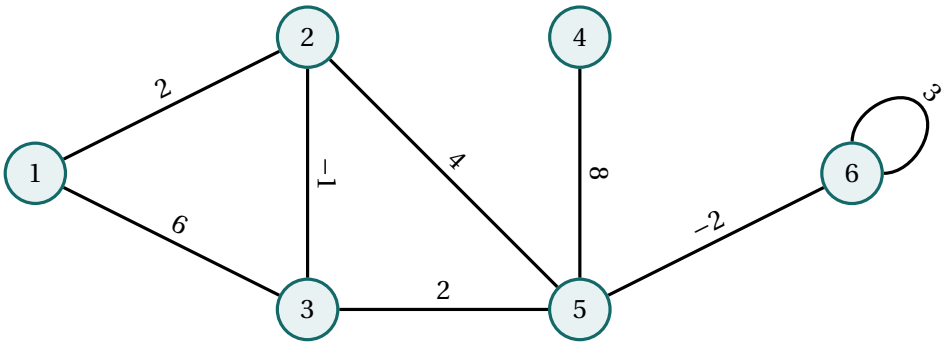
Définition (Graphe pondéré) : Un graphe pondéré G est un triplet $G = (S, A, \omega)$ où (S, A) est un graphe orienté ou non et ω est une application définie sur A à valeurs dans $\mathbb{R}$.

Remarque 9 : Pour chaque arête (ou arc) $a \in A$, l'élément $\omega(a)$ est une valeur numérique associée à a, appelée le poids de l'arête a.

Exemple 14 : Reprenons le graphe de l'exemple 1 et pondérons le avec l'application $\varphi : A \to \mathbb{R}$ définie par

$$\omega(\{1,2\}) = 2, \quad \omega(\{1,3\}) = 6, \quad \omega(\{2,3\}) = -1, \quad \omega(\{2,5\}) = 4,$$
$$\omega(\{3,5\}) = 2, \quad \omega(\{4,5\}) = 8, \quad \omega(\{5,6\}) = -2, \quad \omega(\{6,6\}) = 3.$$

Pour représenter un graphe pondéré, l'usage est d'ajouter le poids de chaque arête (ou arc) sur la représentation de (S, A).



Plus généralement, on peut ajouter une étiquette à tous sommets ou toutes arêtes d'un graphe pour indiquer des informations supplémentaires.

Définition (Graphe étiqueté) : Un graphe étiqueté G est un triplet $G = (S, A, \varphi)$ où (S, A) est un graphe orienté ou non et φ est une application définie sur $S \cup A$ à valeurs dans un ensemble E.

Remarque 10 : Pour chaque élément $x \in S \cup A$, l'élément $\varphi(x)$ est l'étiquette associée à x. Cette dernière permet d'ajouter des informations à certains sommets ou à certaines arêtes du graphe.

Partie II Implémentation des graphes

Dans cette partie, nous allons voir les deux méthodes principales pour représenter les graphes en machine.

II.A - Matrice d'adjacence

Définition (Matrice d'adjacence d'un graphe) : Soit $G = (S, A)$ un graphe. On choisit arbitrairement une numérotation des sommets de G de sorte que $S = \{s_1, \dots, s_n\}$ où $n = \text{Card}(S)$.

(i) Si G est un graphe non orienté, alors la matrice d'adjacence de G est la matrice $M \in \mathscr{M}_n(\mathbb{R})$ définie par

$$\forall (i,j) \in [\![1,n]\!]^2, \quad m_{i,j} = \begin{cases} 1 & \text{si les sommets } s_i \text{ et } s_j \text{ sont adjacents,} \\ 0 & \text{sinon.} \end{cases}$$

(ii) Si G est un graphe orienté, alors la matrice d'adjacence de G est la matrice $M \in \mathscr{M}_n(\mathbb{R})$ définie par

$$\forall (i,j) \in [\![1,n]\!]^2, \quad m_{i,j} = \begin{cases} 1 & \text{s'il existe un arc d'origine } s_i \text{ et de cible } s_j, \\ 0 & \text{sinon.} \end{cases}$$

Exemples 15 :

a) La matrice d'adjacence du graphe non orienté de l'exemple 1 est

$$M = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix} \in \mathscr{M}_6(\mathbb{R}).$$

b) La matrice d'adjacence du graphe orienté de l'exemple 8 est

$$M = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix} \in \mathscr{M}_6(\mathbb{R}).$$

Remarques 11 :

a) Si le graphe G est non orienté, alors sa matrice d'adjacence M est symétrique.

b) La matrice d'adjacence M caractérise le graphe G : on peut reconstruire le graphe G uniquement à partir des informations contenues dans la matrice M.

c) Pour manipuler un graphe en Python, nous pourrons donc sauvegarder sa matrice d'adjacence, par exemple comme une liste de listes.

Exemple 16 : Le graphe orienté de l'exemple 8 peut s'implémenter avec la variable ci-dessous.

```
M = [
[0, 1, 1, 0, 0, 0],
[1, 0, 1, 1, 1, 0],
[0, 0, 0, 0, 1, 0],
[0, 0, 0, 0, 0, 0],
[0, 0, 0, 1, 0, 1],
[0, 0, 0, 0, 1, 1]]
```

Bilan : La représentation d'un graphe G par sa matrice d'adjacence M présente plusieurs avantages :

- pour vérifier s'il existe ou non une arête d'extrémités s_i et s_j (ou un arc d'origine s_i et de cible s_j), il suffit d'accéder à la variable `M[i-1][j-1]`, ce qui se fait avec une complexité constante ;
- ajouter ou supprimer une arête (ou un arc) au graphe se fait également avec un coût constant.

Par contre, cette approche présente aussi des inconvénients :

- le coût spatial pour sauvegarder la matrice d'adjacence du graphe est en $O(n^2)$, ce qui peut être relativement important ;
- déterminer tous les sommets adjacents (ou tous les successeurs) d'un sommet donné a un coût en $O(n)$.

Remarque 12 : Lorsque $G = (S, A, \omega)$ est un graphe pondéré, la définition de la matrice d'adjacence est souvent modifiée pour incorporer le poids des arêtes (ou des arcs) à la matrice.

Exemple 17 : Pour représenter le graphe pondéré de l'exemple 14, on considère la matrice $M \in \mathcal{M}_6(\mathbb{R})$ définie par

$$\forall (i,j) \in [\![1,n]\!]^2, \quad m_{i,j} = \begin{cases} \omega(\{i,j\}) & \text{si les sommets } i \text{ et } j \text{ sont adjacents,} \\ +\infty & \text{sinon.} \end{cases}$$

On obtient dans ce cas la matrice d'adjacence suivante.

$$M = \begin{pmatrix} +\infty & 2 & 6 & +\infty & +\infty & +\infty \\ 2 & +\infty & -1 & +\infty & 4 & +\infty \\ 6 & -1 & +\infty & +\infty & 2 & +\infty \\ +\infty & +\infty & +\infty & +\infty & 8 & +\infty \\ +\infty & 4 & 2 & 8 & +\infty & -2 \\ +\infty & +\infty & +\infty & +\infty & -2 & 3 \end{pmatrix} \in \mathcal{M}_6(\mathbb{R}).$$

Précisons néanmoins que la définition donnée ci-dessus peut varier d'un auteur à l'autre et en fonction du problème étudié : elle est en général précisée dans l'énoncé.

Exercice 3 : Écrire la matrice d'adjacence de chacun des graphes des exercices 1 et 2.

II.B - Listes d'adjacence

Définition (Listes d'adjacence d'un graphe) : Soit $G = (S, A)$ un graphe. On choisit arbitrairement une numérotation des sommets de G de sorte que $S = \{s_1, \dots, s_n\}$ où $n = \text{Card}(S)$.

(i) Si G est un graphe non orienté, on appelle listes d'adjacence du graphe G les listes $L_1, \dots, L_n$ telles que pour tout indice $i \in [\![1,n]\!]$, les éléments de L_i sont les sommets adjacents à s_i.

(ii) Si G est un graphe orienté, on appelle listes d'adjacence du graphe G les listes $L_1, \dots, L_n$ telles que pour tout indice $i \in [\![1,n]\!]$, les éléments de L_i sont les sommets successeurs de s_i.

Remarque 13 : En pratique, les listes d'adjacence sont rassemblées dans une liste ou dans un dictionnaire.

Exemples 18 :

a) Le dictionnaire des listes d'adjacence du graphe non orienté de l'exemple 1 est

```
d = {1: [2, 3], 2: [1, 3, 5], 3: [1, 2, 5], 4: [5], 5: [2, 3, 4, 6], 6: [5, 6]}
```

b) Le dictionnaire des listes d'adjacence du graphe orienté de l'exemple 8 est

```
d = {1: [2, 3], 2: [1, 3, 4, 5], 3: [5], 4: [], 5: [4, 6], 6: [5, 6]}
```

Bilan : La représentation d'un graphe G par ses listes d'adjacence présente plusieurs avantages :

- ajouter ou supprimer une arête (ou un arc) au graphe se fait également avec un coût constant ;
- on dispose directement de tous les sommets adjacents (ou tous les successeurs) d'un sommet donné en mémoire.

Par contre, cette approche présente aussi un inconvénient :

- pour vérifier s'il existe ou non une arête d'extrémités s_i et s_j (ou un arc d'origine s_i et de cible s_j), il est nécessaire de parcourir la liste d'adjacence L_i, ce qui dans le pire des cas a un coût en $O(n)$.

Finalement, remarquons que le coût spatial pour sauvegarder les listes d'adjacence d'un graphe est en $O(n+p)$ où $p = \text{Card}(A)$ est le nombre d'arêtes (ou d'arcs). Cette complexité est meilleure que le coût spatial $O(n^2)$ pour sauvegarder la matrice d'adjacence du moment que le nombre d'arêtes (ou d'arcs) p est du même ordre de grandeur que n (ou plus généralement si p est négligeable devant n^2).

Remarque 14 : Lorsque $G = (S, A, \omega)$ est un graphe pondéré, la définition des listes d'adjacence est souvent modifiée pour incorporer le poids des arêtes (ou des arcs) à la matrice.

Exemple 19 : Pour représenter le graphe pondéré de l'exemple 14, on peut définir les listes d'adjacence $L_1, \dots, L_n$ en convenant que pour tout indice $i \in [\![1, n]\!]$, les éléments de L_i sont les couples de la forme $\big(j, \omega(i,j)\big)$ où j parcourt l'ensemble des sommets adjacents au sommet i.

```
d = {1: [(2, 2), (3, 6)], 2: [(1, 2), (3, -1), (5, 4)], 3: [(1, 6), (2, -1), (5, 2)],
4: [(5, 8)], 5: [(2, 4), (3, 2), (4, 8), (6, -2)], 6: [(5, -2), (6, 3)]}
```

Comme pour la matrice d'adjacence d'un graphe pondéré, la définition choisie ci-dessus peut varier d'un auteur à l'autre et en fonction du problème étudié : elle est en général précisée dans l'énoncé.

Exercice 4 : Écrire le dictionnaire des listes d'adjacence de chacun des graphes des exercices 1 et 2.

Exercice 5 : Écrire une fonction`matrice_vers_liste(M)` prenant en entrée une matrice d'adjacence `M` d'un graphe G et renvoyant la liste des listes d'adjacence du graphe G.

Exercice 6 : Écrire une fonction `liste_vers_matrice(lst)` prenant en entrée une liste `lst` contenant les listes d'adjacence d'un graphe G et renvoyant la matrice d'adjacence du graphe G.

Partie III Les piles et les files

Pour décrire les algorithmes de base sur les graphes, nous allons introduire deux nouvelles structures de données.

III.A - Les piles

Une pile est une structure de données séquentielle (ou linéaire) dans laquelle les éléments sont accessibles selon le principe LIFO (« last in, first out ») : le dernier élément ajouté à la pile est le premier à en sortir. Nous avons déjà rencontré une pile lorsque nous avons étudié la récursivité : la pile d'exécution.

Les opérations élémentaires sur les piles sont les suivantes.

- **générer** une nouvelle pile vide ;
- **tester** si une pile donnée est vide ;
- **empiler** un élément sur la pile, i.e. ajouter un élément à son sommet ;
- **dépiler** un élément, i.e retirer et renvoyer l'élément au sommet de la pile (si la pile n'est pas vide).

On peut implémenter une structure de pile en s'appuyant sur la classe `list`.

```
1  def nouvelle_pile():
2      return([])
3
4  def est_pile_vide(p):
5      return(p == [])
6
7  def empiler(p, x):
8      p.append(x)
9
10 def depiler(p):
11     assert not est_pile_vide(p)
12     return(p.pop())
```

Une fois ces fonctions définies, on peut facilement manipuler une pile.

```
>>> p = nouvelle_pile()
>>> p
[]
>>> est_pile_vide(p)
True
>>> empiler(p, 5)
>>> p
[5]
>>> est_pile_vide(p)
False
>>> empiler(p, 9)
>>> p
[5, 9]
```

```
>>> empiler(p, 9)
>>> empiler(p, 3)
>>> empiler(p, 7)
>>> p
[5, 9, 9, 3, 7]
>>> depiler(p)
7
>>> p
[5, 9, 9, 3]
>>> depiler(p)
3
>>> p
[5, 9, 9]
```

Remarque 15 : On considère que les quatre opérations élémentaires sur les piles ont une complexité constante.

Exercice 7 : Écrire une fonction `echange(p)` prenant en argument une pile `p` et qui échange les deux éléments au sommet de la pile. Si la pile a strictement moins de deux éléments, la fonction ne la modifiera pas.

III.B - Les files

Une pile est une structure de données séquentielle (ou linéaire) dans laquelle les éléments sont accessibles selon le principe FIFO (« first in, first out ») : le premier élément ajouté à la file est le premier à en sortir.

Les opérations élémentaires sur les files sont les suivantes.

- **générer** une nouvelle file vide ;
- **enfiler** un élément sur la file, i.e. ajouter un élément à la fin de la file ;
- **défiler** un élément, i.e. retirer et renvoyer l'élément au début de la file ;
- **tester** si une file donnée est vide.

Comme pour les piles, on pourrait implémenter une structure de file en s'appuyant sur la classe `list`. Cependant, cette approche n'est pas satisfaisante, car la fonction `defiler` utiliserait la commande `list.pop(0)` qui a pour effet de retirer l'élément d'indice 0 et de décaler tous les éléments suivants dans la liste, ce qui induit un coût linéaire. Pour contourner cette difficulté, nous allons utiliser la classe `deque` du module `collections` dont l'implémentation permet de retirer l'élément d'indice 0 en temps constant.

```
1  from collections import deque
2
3  def nouvelle_file():
4      return(deque())
5
6  def est_file_vide(f):
7      return(f == deque([]))
8
9  def enfiler(f, x):
10     f.append(x)
11
12 def defiler(f):
13     assert not est_file_vide(p)
14     return(f.popleft())
```

Une fois ces fonctions définies, on peut facilement manipuler une pile.

```
>>> f = nouvelle_pile()
>>> f
deque([])
>>> est_file_vide(f)
True
>>> enfiler(f, 5)
>>> f
deque([5])
>>> est_file_vide(f)
False
>>> enfiler(f, 9)
>>> f
deque([5, 9])
```

```
>>> enfiler(f, 9)
>>> enfiler(f, 3)
>>> enfiler(f, 7)
>>> f
deque([5, 9, 9, 3, 7])
>>> defiler(f)
5
>>> f
deque([9, 9, 3, 7])
>>> defiler(f)
9
>>> f
deque([9, 3, 7])
```

Remarque 16 : On considère que les quatre opérations élémentaires sur les piles ont une complexité constante.

Partie IV Parcours d'un graphe

IV.A - Généralités

Parcourir un graphe, c'est énumérer l'ensemble des sommets accessibles par une chaîne (ou un chemin) à partir d'un sommet de départ, dans l'objectif de leur faire subir un certain traitement. En général, on utilise :

- une liste ou un dictionnaire `deja_traites` contenant des booléens pour marquer les sommets déjà traités ;
- une structure séquentielle `a_traiter` dont les éléments sont les sommets en cours de traitements.

L'algorithme de base pour parcourir un graphe est le suivant.

Algorithme de base pour parcourir un graphe

```
Fonction parcours(G, s_d)
    Entrée : Un graphe G, un sommet s_d de G
    Sortie : À adapter en fonction du problème étudié
    Initialiser un objet deja_traites avec False pour chaque sommet de G
    Initialiser une structure linéaire vide a_traiter
    Ajouter s_d à a_traiter
    tant que a_traiter n'est pas vide faire
        Extraire un élément s de a_traiter
        si deja_traites[s] est False alors
            deja_traites[s] ← True
            Instructions sur s dépendant du problème étudié
            pour chaque sommet v adjacent (ou successeur) de s faire
                si deja_traites[v] est False alors
                    Ajouter v à a_traiter
                Instructions sur v dépendant du problème étudié
    retourner À adapter en fonction du problème étudié
```

Nous allons voir que les différents algorithmes de parcours d'un graphe que nous allons étudier dans la suite reprennent la structure globale du pseudo-code ci-dessus.

Remarques 17 :

a) Une fois le parcours terminé, les sommets traités sont exactement ceux qui sont atteignables avec une chaîne (ou un chemin) depuis le sommet de départ.

b) L'ordre de traitement des différents sommets du graphe dépend du choix de l'élément extrait à chaque tour de la boucle `while` qui dépend lui-même de la structure retenue pour l'objet `a_traiter`.

c) Pour appliquer l'algorithme ci-dessus, il n'est pas nécessaire de connaître le graphe entièrement : il suffit de savoir comment trouver les sommets adjacents (ou successeurs) d'un sommet donné. En particulier, cela permet d'utiliser les algorithmes de parcours sur des graphes gigantesques qui ne sont pas possibles à générer intégralement en un temps raisonnable.

IV.B - Parcours en profondeur

L'algorithme de parcours en profondeur (ou DFS pour *Depth First Search* en anglais) consiste à explorer une chaîne (ou un chemin) jusqu'à rencontrer une impasse ou un sommet déjà traité, auquel cas il revient sur le dernier sommet où on pouvait suivre une autre chaîne (ou chemin) qu'il explore alors.

Exemples 20 :

a) En appliquant l'algorithme de parcours en profondeur au graphe non orienté de l'exemple 1 en partant du sommet 1, un ordre d'exploration possible des sommets est $1-2-3-5-4-6$.

b) En effectuant d'autres choix pour les sommets adjacents à explorer en premier, un autre ordre d'exploration possible pour l'exemple précédent est $1-3-2-5-6-4$.

c) En appliquant l'algorithme de parcours en profondeur au graphe orienté de l'exemple 8 en partant du sommet 3, un ordre d'exploration possible des sommets est $3-5-4-6$.

Remarque 18 : Lors d'un parcours en profondeur, l'ordre d'exploration des sommets n'est pas unique en général : il dépend de l'ordre choisi pour explorer les sommets adjacents (ou successeurs) à chaque étape.

Pour mettre en pratique ce principe, il suffit de reprendre l'algorithme de base pour parcourir un graphe rédigé dans la sous-partie précédente en employant une pile pour gérer les sommets en cours de traitement.

```
1  def parcours_profondeur(dico_adjacence, s_d):
2
3      # On initialise un dictionnaire pour marquer les sommets déjà traités
4      deja_traites = {s: False for s in dico_adjacence}
5
6      # On initialise une pile pour les sommets en cours de traitement
7      pile_traitement = nouvelle_pile()
8
9      # On initialise une liste pour sauvegarder l'ordre de traitement des sommets
10     ordre = []
11
12     empiler(pile_traitement, s_d)
13
14     while not est_pile_vide(pile_traitement):
15         s = depiler(pile_traitement)
16         if not deja_traites[s]:
17             deja_traites[s] = True
18             ordre.append(s)
19             for v in dico_adjacence[s]:
20                 if not deja_traites[v]:
21                     empiler(pile_traitement, v)
22
23     return(ordre)
```

Pour observer l'évolution de la pile, ajoutons l'instruction `print(pile_traitement)` entre la ligne 14 et la ligne 15. Notons `d1` et `d2` les dictionnaires des listes d'adjacence respectifs des graphes de l'exemple 1 et de l'exemple 8.

```
>>> r = parcours_profondeur(d1, 3)
[3]
[1, 2, 5]
[1, 2, 2, 4, 6]
[1, 2, 2, 4]
[1, 2, 2]
[1, 2, 1]
[1, 2]
[1]
>>> r
[3, 5, 6, 4, 2, 1]
```

```
>>> r = parcours_profondeur(d2, 2)
[2]
[1, 3, 4, 5]
[1, 3, 4, 4, 6]
[1, 3, 4, 4]
[1, 3, 4]
[1, 3]
[1]
>>> r
[2, 5, 6, 4, 3, 1]
```

Remarque 19 : En notant $G = (S, A)$ le graphe passé en entrée, la complexité de la fonction `parcours_profondeur` est en $O(n+p)$ où $n = \text{Card}(S)$ est l'ordre de G et $p = \text{Card}(A)$ est la taille de G.

Exercice 8 : Écrire une fonction `presence_cycle(dico_adjacence)` prenant en entrée le dictionnaire d'adjacence `dico_adjacence` d'un graphe non orienté G et renvoyant un booléen indiquant si le graphe G contient un cycle ou non.

Exercice 9 : Écrire une fonction `existe_chemin(dico_adjacence, s_d, s_a)` prenant en entrée le dictionnaire d'adjacence `dico_adjacence` d'un graphe orienté G et deux sommets `s_d` et `s_a` et renvoyant un booléen indiquant s'il existe un chemin du sommet `s_d` au sommet `s_a` ou non.

IV.C - Parcours en largeur

L'algorithme de parcours en profondeur (ou BFS pour *Breadth First Search* en anglais) consiste à explorer le sommet de départ, puis ses successeurs non traités, puis les sommets successeurs non traités des successeur, etc.

Plus formellement, si s et s' sont deux sommets d'un graphe G, on définit la distance de s à s' comme la plus petite des longueurs des chaînes (ou chemins) allant de s à s', i.e.

$$d(s,s') = \inf\{n \in \mathbb{N} \mid \text{il existe une chaine (ou un chemin) de longueur } n \text{ allant de } s \text{ à } s'\}.$$

Remarque 20 : On a que $d(s,s') = +\infty$ si et seulement si il n'existe aucune chaîne (ou chemin) allant de s à s'.

Avec cette nouvelle notion, l'algorithme de parcours en largeur consiste à explorer le sommet de départ, puis les sommets à distance 1 du sommet de départ, puis les sommets à distance 2 du sommet de départ, etc.

Exemples 21 :

a) En appliquant l'algorithme de parcours en largeur au graphe non orienté de l'exemple 1 en partant du sommet 4, un ordre d'exploration possible des sommets est $4-5-2-3-6-1$.

b) En effectuant d'autres choix pour les sommets adjacents à explorer en premier, un autre ordre d'exploration possible pour l'exemple précédent est $4-5-6-2-3-1$.

c) En appliquant l'algorithme de parcours en largeur au graphe orienté de l'exemple 8 en partant du sommet 2, un ordre d'exploration possible des sommets est $2-1-3-4-5-6$.

Remarque 21 : Lors d'un parcours en largeur, l'ordre d'exploration des sommets n'est pas unique en général : il dépend de l'ordre choisi pour explorer les sommets adjacents (ou successeurs) à chaque étape.

Pour mettre en pratique ce principe, il suffit de reprendre l'algorithme de base pour parcourir un graphe rédigé dans la sous-partie précédente en employant une file pour gérer les sommets en cours de traitement.

```python
def parcours_largeur(dico_adjacence, s_d):

    # On initialise un dictionnaire pour marquer les sommets déjà traités
    deja_traites = {s: False for s in dico_adjacence}

    # On initialise une file pour les sommets en cours de traitement
    file_traitement = nouvelle_file()

    # On initialise une liste pour sauvegarder l'ordre de traitement des sommets
    ordre = []

    enfiler(file_traitement, s_d)

    while not est_file_vide(file_traitement):
        s = defiler(file_traitement)
        if not deja_traites[s]:
            deja_traites[s] = True
            ordre.append(s)
            for v in dico_adjacence[s]:
                if not deja_traites[v]:
                    enfiler(file_traitement, v)

    return(ordre)
```

Pour observer l'évolution de la file, ajoutons l'instruction `print(file_traitement)` entre la ligne 14 et la ligne 15. Notons `d1` et `d2` les dictionnaires des listes d'adjacence respectifs des graphes de l'exemple 1 et de l'exemple 8.

```
>>> r = parcours_largeur(d1, 1)
deque([1])
deque([2, 3])
deque([3, 3, 5])
deque([3, 5, 5])
deque([5, 5])
deque([5, 4, 6])
deque([4, 6])
deque([6])
>>> r
[1, 2, 3, 5, 4, 6]
```

```
>>> r = parcours_largeur(d1, 1)
deque([1])
deque([2, 3])
deque([3, 3, 4, 5])
deque([3, 4, 5, 5])
deque([4, 5, 5])
deque([5, 5])
deque([5, 6])
deque([6])
>>> r
[1, 2, 3, 4, 5, 6]
```

Remarque 22 : En notant $G = (S, A)$ le graphe passé en entrée, la complexité de la fonction `parcours_largeur` est en $O(n + p)$ où $n = \text{Card}(S)$ est l'ordre de G et $p = \text{Card}(A)$ est la taille de G.

Exercice 10 : Écrire une fonction `est_connexe(dico_adjacence)` prenant en entrée le dictionnaire d'adjacence `dico_adjacence` d'un graphe non orienté G et renvoyant un booléen indiquant si le graphe G est connexe ou non.

Exercice 11 : Écrire une fonction `distances(dico_adjacence, s_d)` prenant en entrée le dictionnaire d'adjacence `dico_adjacence` d'un graphe G et un sommet `s_d` de G et qui renvoie un dictionnaire dont les clés sont les sommets `s` de G avec pour valeur la distance du sommet `s_d` au sommet `s`.

Partie V Problème de plus court chemin

Pour alléger la rédaction, on utilisera dans cette partie le vocabulaire des graphes orientés, mais l'ensemble des notations et des algorithmes restent valables dans un graphe non orienté.

V.A - Généralités

Nous venons de voir qu'en utilisant un parcours en largeur, nous pouvons déterminer le nombre minimal d'arcs à parcourir entre deux sommets.

Cette solution n'est pas satisfaisante dans certains problèmes représentés par un graphe pondéré $G = (S, A, \omega)$. Dans ce cadre, on définit le poids d'un chemin c, noté $\omega(c)$, comme la somme des poids des arcs qui le composent. Pour tous sommets s et s' du graphe G, on définit la distance (pondérée) de s à s' comme le poids minimal des chemins allant de s à s', i.e.

$$\delta(s, s') = \inf\{\omega(c) \in \mathbb{R} \mid c \text{ est un chemin allant de } s \text{ à } s'\}.$$

Remarques 23 :

a) On a $\delta(s, s') = +\infty$ si et seulement si il n'existe aucun chemin allant de s à s'.

b) S'il existe une chaîne (ou un chemin) allant de s à s' contenant un cycle (ou un circuit) de poids strictement négatif, alors on a $\delta(s, s') = -\infty$. En effet, dans ce cas, on peut générer des chaînes (ou des chemins) de poids arbitrairement petit en multipliant les passages par ce cycle (ou ce circuit).

Dans la suite, on s'intéresse au problème de plus courts chemins dans un graphe pondéré $G = (S, A, \omega)$, que l'on peut en fait distinguer en trois problèmes :

(i) calculer le chemin de poids minimal entre un sommet de départ `s_d` et un sommet d'arrivée `s_a` ;

(ii) calculer les chemins de poids minimal entre un sommet de départ `s_d` et tout autre sommet du graphe ;

(iii) calculer tous les chemins de poids minimal entre deux sommets quelconques du graphe.

Exemples 22 :

a) Dans le graphe pondéré $G = (S, A, \omega)$ de l'exemple 14, la chaîne de poids minimal allant du sommet 1 au sommet 6 est $c = (1, 2, 3, 5, 6)$ et on a $\delta(1, 6) = \omega(c) = 2 + (-1) + 2 + (-2) = 1$.

b) Considérons le graphe $G = (S, A)$ associé au réseau ferroviaire français.

- On peut choisir de pondérer le graphe G en ajoutant à chacune de ses arêtes la distance de la voie ferrée qu'elle représente. Dans ce cas, le problème du plus court chemin revient à chercher le trajet en train le plus court entre la gare de départ et la gare d'arrivée.
- On peut choisir de pondérer le graphe G en ajoutant à chacune de ses arêtes le temps nécessaire pour parcourir la voie ferrée qu'elle représente. Dans ce cas, le problème du plus court chemin revient à chercher le trajet en train le plus rapide entre la gare de départ et la gare d'arrivée.

Remarques 24 :

a) Pour résoudre ces problèmes, une première approche serait d'énumérer l'ensemble des chemins possibles pour trouver celui de poids minimal. En général, cette approche ne peut aboutir pour des raisons de complexité dès que l'ordre du graphe est grand.

b) Il n'existe pas actuellement un algorithme permettant de fournir une solution du premier problème sans résoudre le second. Dans la partie suivante, nous allons résoudre le second problème pour un graphe pondéré positivement avec l'algorithme de Dijkstra.

c) Le troisième problème peut se résoudre avec l'algorithme de Floyd-Warshall que vous étudierez peut-être en seconde année dans le cadre du chapitre sur la programmation dynamique.

d) Il existe de nombreux algorithmes pour résoudre un problème du plus court chemin dans un graphe pondéré adaptés à la nature des valeurs et des contraintes supplémentaires qui peuvent être imposées.

V.B - Algorithme de Dijkstra

Dans cette sous-partie, nous allons étudier une approche plus efficace : l'algorithme de Dijkstra, publié en 1959 par l'informaticien néerlandais Edsger Dijkstra. Dans le cas d'un graphe pondéré avec des poids positifs, ce dernier permet de calculer tous les chemins de poids minimal entre un sommet `s_d` et les autres sommets du graphe.

Son principe reprend celui du parcours en largeur en partant d'un sommet de départ `s_d` et en effectuant pour chaque nouveau sommet traité `s` une mise à jour des distances des voisins `v` de `s` au sommet de départ `s_d`. Pour s'assurer du bon fonctionnement de l'algorithme, le point clé est de traiter le sommet en attente dont la distance à `s_d` est minimale.

Algorithme de Dijkstra

```
Fonction dijkstra(G, s_d)
    Entrée : Un graphe pondéré G avec des poids positifs, un sommet s_d de G
    Sortie : Les poids minimaux de s_d aux autres sommets de G

    Initialiser un objet deja_traites avec False pour chaque sommet de G
    Initialiser une structure linéaire vide a_traiter
    Initialiser un objet distances avec +∞ pour chaque sommet de G
    Initialiser un objet predecesseurs avec None pour chaque sommet de G

    Ajouter s_d à a_traiter
    distances[s_d] ← 0

    tant que a_traiter n'est pas vide faire
        Extraire un élément s de a_traiter tel que distances[s] est minimal

        si deja_traites[s] est False alors
            pour chaque sommet v successeur de s faire
                si deja_traites[v] est False alors
                    Ajouter v à a_traiter

                si distances[s] + poids[s,v] < distances[v] alors
                    distances[v] ← distances[s] + poids[s,v]
                    predecesseurs[v] ← s

    retourner (distances, predecesseurs)
```

Remarques 25 :

a) L'objet `predecesseurs` permet de mémoriser pour chaque sommet `s` le prédécesseur de `s` par lequel est passé un chemin de poids minimal de `s_d` à `s`. En utilisant les informations dans `predecesseurs`, on peut ainsi reconstruire un chemin de poids minimal du sommet de départ `s_d` à tout autre sommet `s_a` du graphe.

b) L'algorithme se termine car le nombre de sommets non traités est un variant de boucle.

c) En notant T l'ensemble des sommets traités, on peut démontrer que la propriété

$$\forall s \in S, \quad \texttt{distances[s]} = \begin{cases} \delta(\texttt{s\_d},\texttt{s}) & \text{si} \quad \texttt{s} \in T \\ \inf\{\texttt{distances[t]} + \texttt{poids[t,s]} \mid \texttt{t} \in T \text{ est un prédécesseur de s}\} & \text{si} \quad \texttt{s} \notin T \end{cases}$$

est un invariant de la boucle `while`, ce qui permet de prouver la correction totale de l'algorithme. Le point clé est le principe de sous-optimalité suivant : si c est un plus court chemin de `s_d` à `s_a` passant par `s`, alors les chemins obtenus en scindant c en `s` sont des plus courts chemins de `s_d` à `s` et de `s` à `s_a`.

d) Si on ne souhaite que récupérer le plus court chemin du sommet `s_d` à un sommet `s_a` passé en entrée, alors on peut ajouter arrêter la boucle « tant que » dès que le sommet `s_a` a été traité.

Voici une implémentation possible de l'algorithme de Dijkstra en Python.

```python
def dijkstra(dico_adjacence, s_d):

    # On initialise un dictionnaire pour marquer les sommets déjà traités
    deja_traites = {s: False for s in dico_adjacence}

    # On initialise une liste pour les sommets à traiter
    liste_traitement = []

    # On initialise deux dictionnaires pour les distances à s_d et les prédécesseurs
    distances = {s: float('inf') for s in dico_adjacence}
    predecesseurs = {s: None for s in dico_adjacence}

    liste_traitement.append(s_d)
    distances[s_d] = 0

    while liste_traitement != []:
        # Recherche du sommet à traiter avec une distance minimale à s_d
        s = liste_traitement[0]
        for t in liste_traitement:
            if distances[t] < distances[s]:
                s = t
        liste_traitement = [t for t in liste_traitement if t != s]

        if not deja_traites[s]:
            deja_traites[s] = True

            for (v, d) in dico_adjacence[s]:
                if not deja_traites[v]:
                    liste_traitement.append(v)

                if distances[s] + d < distances[v]:
                    distances[v] = distances[s] + d
                    predecesseurs[v] = s

    return(distances, predecesseurs)
```

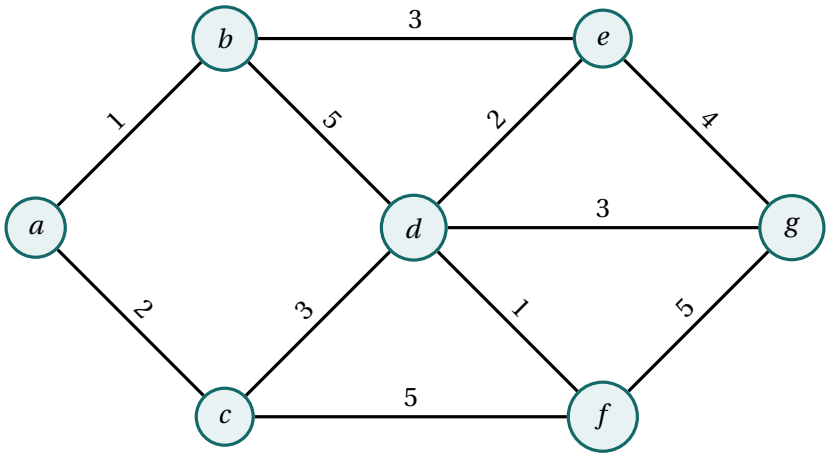
Remarques 26 : Notons $n = \text{Card}(S)$ et $p = \text{Card}(A)$.

a) La version ci-dessus de l'algorithme utilise une structure de liste pour mémoriser les sommets à traiter. La recherche du sommet à traiter se trouvant à une distance minimale du sommet de départ et sa suppression de la liste se fait avec une complexité en $O(n)$, ce qui permet d'aboutir à une complexité en $O\left(n^2\right)$ pour l'algorithme de Dijkstra.

b) En utilisant une structure de file de priorité implémentée avec un tas de Fibonacci (hors-programme) pour mémoriser les sommets à traiter, on peut accéder au minimum et le supprimer de la file avec une complexité en $O(\log(n))$, ce qui permet d'améliorer la complexité de l'algorithme de Dijkstra en $O\left(p + n\log(n)\right)$.

Exercice 12 : Écrire une fonction `plus_court_chemin(dico_adjacence, s_d, s_a)` prenant en entrée le dictionnaire d'adjacence d'un graphe pondéré G qui renvoie un chemin de poids minimal allant du sommet `s_d` au sommet `s_a`.

Exemple 23 : Considérons le graphe non orienté pondéré ci-dessous.



On applique l'algorithme de Dijkstra en partant du sommet a. Pour cela, nous allons remplir un tableau à double entrée : la première ligne fournira la valeur du sommet traité `s` et celles de `distances` avec celles de `predecesseurs` entre parenthèse avant d'entrer dans la boucle, puis chacune des lignes suivantes donnera les valeurs mises à jour de ces variables à chaque tour de la boucle. Une fois qu'on a traité un sommet, on arrête de remplir sa colonne, car les valeurs correspondantes ne peuvent plus changer.

		distances (predecesseurs)						
	s	a	b	c	d	e	f	g
Initialisation	×	0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
Tour 1	a	0	$1(a)$	$2(a)$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
Tour 2	b			$2(a)$	$6(b)$	$4(b)$	$+\infty$	$+\infty$
Tour 3	c				$5(c)$	$4(b)$	$7(c)$	$+\infty$
Tour 4	e				$5(c)$		$7(c)$	$8(e)$
Tour 5	d						$6(d)$	$8(e)$
Tour 6	f							$8(e)$
Tour 7	g							

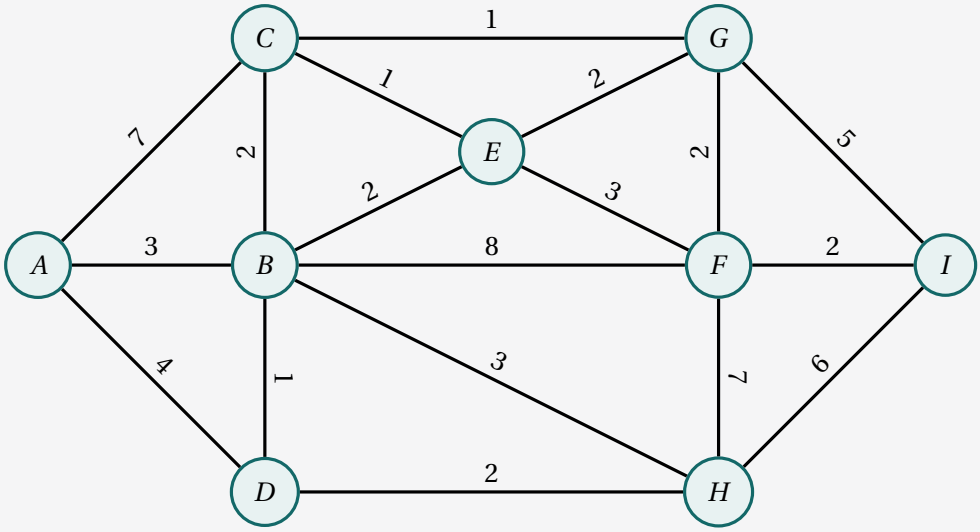
On en déduit toutes les distances pondérées entre a et les autres sommets du graphe.

$$\delta(a,a)=0,\quad \delta(a,b)=1,\quad \delta(a,c)=2,\quad \delta(a,d)=5,\quad \delta(a,e)=4\quad \delta(a,f)=6,\quad \delta(a,g)=8.$$

De plus, on peut reconstruire chaque chemin de poids minimal en remontant les prédécesseurs mémorisés à partir du sommet d'arrivée. Par exemple, le chemin de poids minimal allant du sommet a au sommet f est (a,c,d,f).

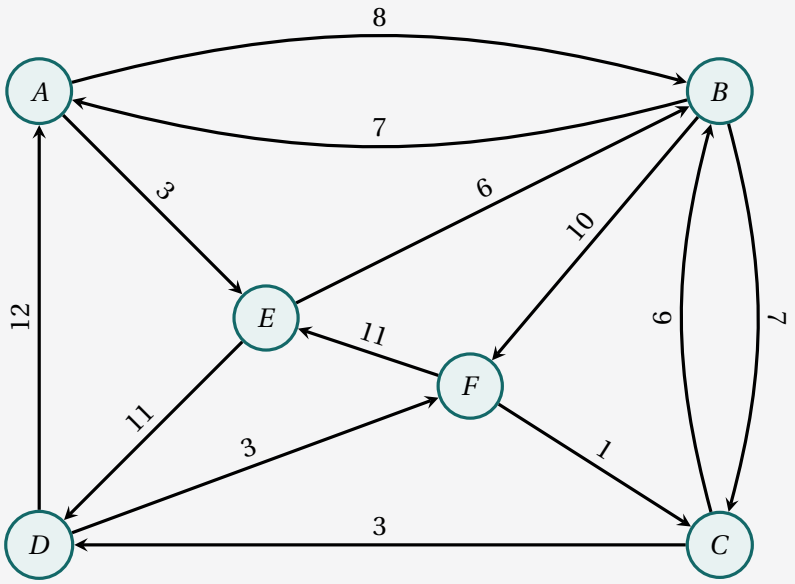
Exercice 13 : Appliquer l'algorithme de Dijkstra au graphe de l'exemple ci-dessus en partant du sommet f, puis donner le chemin de poids minimal allant du sommet f au sommet b?

Exercice 14 : On considère le graphe non orienté et pondéré ci-dessous.



1. Appliquer l'algorithme de Dijkstra au graphe ci-dessous en partant du sommet A.
2. En déduire les plus courts chemins partant du sommet A à chaque autre sommet du graphe.

Exercice 15 : On considère le graphe orienté et pondéré ci-dessous.



1. Appliquer l'algorithme de Dijkstra au graphe ci-dessous en partant du sommet A.
2. En déduire les plus courts chemins partant du sommet A à chaque autre sommet du graphe.

V.C - Algorithme A*

L'algorithme de A* est une variante de celui de Dijkstra. L'idée est modifier les sommets à explorer en priorité en ne comparant plus les distances d'un sommet `s` au sommet de départ `s_d`, mais une estimation du poids du chemin le plus court partant du sommet de départ `s_d`, passant par `s` et arrivant au sommet d'arrivée `s_a`.

Pour calculer cette estimation, il faut choisir une fonction $h : S \to \mathbb{R}_+$, appelée heuristique, telle que $h(s)$ soit une estimation de la distance du sommet `s` au sommet d'arrivée `s_a`. Le choix d'une fonction h pertinente dépend du problème étudié.

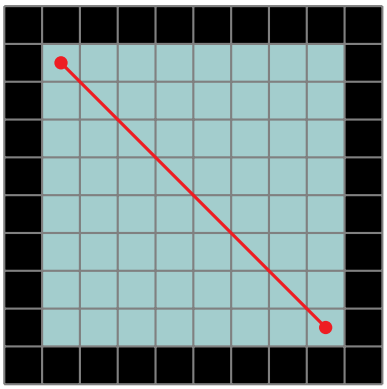
Voici une implémentation possible de l'algorithme A* en Python.

```python
def A_etoile(dico_adjacence, s_d, s_a, h):

    # On initialise un dictionnaire pour marquer les sommets déjà traités
    deja_traites = {s: False for s in dico_adjacence}

    # On initialise une liste pour les sommets à traiter
    liste_traitement = []

    # On initialise trois dictionnaires pour les différentes valeurs à mémoriser
    distances = {s: float('inf') for s in dico_adjacence}
    predecesseurs = {s: None for s in dico_adjacence}
    estimations = {s: float('inf') for s in dico_adjacence}

    liste_traitement.append(s_d)
    distances[s_d] = 0
    estimations[s_d] = h(s_d, s_a)

    while liste_traitement != []:
        # Recherche du sommet à traiter avec une distance minimale à s_d
        s = liste_traitement[0]
        for t in liste_traitement:
            if distances[t] < distances[s]:
                s = t
        liste_traitement = [t for t in liste_traitement if t != s]

        if s == s_a:
            return(distances, predecesseurs)

        if not deja_traites[s]:
            deja_traites[s] = True

            for (v, d) in dico_adjacence[s]:
                if not deja_traites[v]:
                    liste_traitement.append(v)

                if distances[s] + d < distances[v]:
                    distances[v] = distances[s] + d
                    predecesseurs[v] = s
                    estimations[v] = distances[v] + h(v)
```

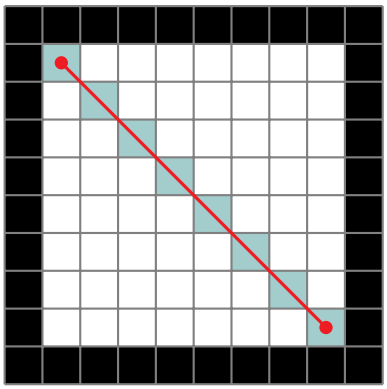
Remarque 27 : En prenant $h = 0$ dans l'algorithme A*, on retrouve l'algorithme de Dijkstra.

Exemple 24 : On considère ci-dessous des labyrinthes de taille 10×10 dans lequel un visiteur part de la case blanche au nord-ouest est souhaite se rendre à la case blanche au sud-est en se déplaçant horizontalement, verticalement ou en diagonale. Ces derniers peuvent se représenter comme un graphe non orienté pondéré dont les sommets sont les cases blanches et dont les arêtes sont les déplacements possibles pour le visiteur entre deux cases blanches voisines (avec pour poids la distance de déplacement).

Pour appliquer l'algorithme A*, on considère la fonction $h_2 : S \to \mathbb{R}_+$ où $h_2(\mathtt{s})$ est la distance à vol d'oiseau du sommet $\mathtt{s}$ au sommet $\mathtt{s\_a}$. On représente ci-dessous en bleu les sommets traités par chacun des algorithmes.

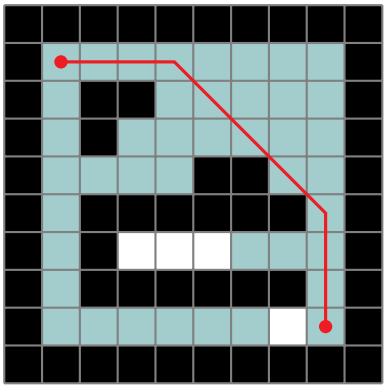


Algorithme de Dijkstra
64 sommets traités
chemin de poids $7\sqrt{2}$

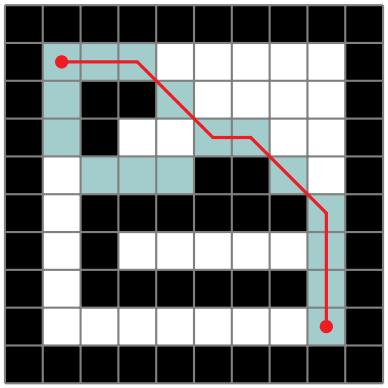


Algorithme A* avec $h = h_2$
8 sommets traités
chemin de poids $7\sqrt{2}$

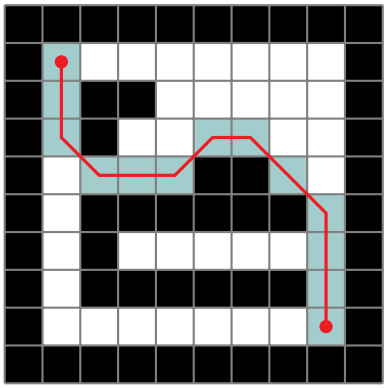
Sur l'exemple simple ci-dessous, on constate que l'algorithme A* est plus rapide et qu'il a trouvé le même chemin que celui de Dijkstra. Voici un autre exemple de labyrinthe.



Algorithme de Dijkstra
42 sommets traités
chemin de poids $6+4\sqrt{2}$



Algorithme A* avec $h = h_2$
16 sommets traités
chemin de poids $6+4\sqrt{2}$



Algorithme A* avec $h = 2h_2$
13 sommets traités
chemin de poids $8+4\sqrt{2}$

L'application de l'algorithme A* avec $h = h_2$ fournit un chemin différent de celui renvoyé par celui de Dijkstra, mais les deux sont des plus courts chemins de $\mathtt{s\_d}$ à $\mathtt{s\_a}$. Dans le dernier exemple, on a augmenté l'estimation en considérant l'heuristique $h = 2h_2$: la convergence est plus rapide, mais on constate que le chemin retourné par l'algorithme A* n'est plus de poids minimal.

Remarques 28 :

a) Dans l'exemple ci-dessus, comme l'heuristique choisie vérifie l'inégalité $h_2(\mathtt{s}) \leqslant \delta(\mathtt{s}, \mathtt{s\_a})$ pour tout $\mathtt{s} \in S$, on peut démontrer que le chemin construit par l'algorithme A* avec l'heuristique h_2 est de poids minimal.

b) En général, l'algorithme A* ne fournit pas un chemin de poids minimal, mais plus l'heuristique utilisée sera une bonne estimation de la distance au sommet d'arrivée, plus le chemin trouvé sera qualitatif.