Chapitre I
Théorie des graphes

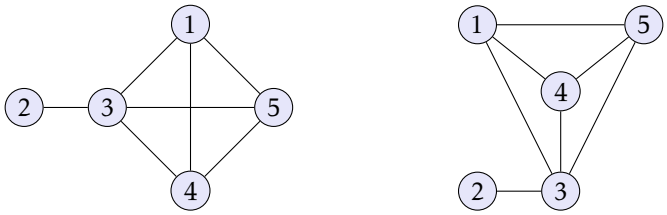
1. Vocabulaire et représentation

De manière générale, un graphe permet de représenter les connexions d'un ensemble en exprimant les relations entre ses éléments : réseau de communication, réseau routier, circuit électronique, ... , mais aussi relations sociales ou interactions entre espèces animales.
Le vocabulaire de la théorie des graphes est utilisé dans de nombreux domaines : chimie, biologie, sciences sociales, etc., mais c'est avant tout une branche à part entière et déjà ancienne des mathématiques (le fameux problème des ponts de Königsberg d'Euler date de 1736). Néanmoins, l'importance accrue que revêt l'aspect algorithmique dans ses applications pratiques en fait aussi un domaine incontournable de l'informatique. Pour schématiser, les mathématiciens s'intéressent avant tout aux propriétés globales des graphes (graphes eulériens, graphes hamiltoniens, dénombrement, ...) là où les informaticiens vont plutôt chercher à concevoir des algorithmes efficaces pour résoudre un problème faisant intervenir un graphe (recherche du plus court chemin, problème du voyageur de commerce, ...). Tout ceci forme un ensemble très vaste, dont nous n'aborderons que quelques aspects, essentiellement de nature algorithmique.

1.1 Graphes non orientés

Un graphe $G = (V, E)$ est défini par l'ensemble fini $V = \{v_1, v_2, \dots, v_n\}$ dont les éléments sont appelés *sommets* (*vertices* en anglais), et par l'ensemble fini $E = \{e_1, e_2, \dots, e_m\}$ dont les éléments sont appelés *arêtes* (*edges* en anglais).
Une arête e de l'ensemble E est définie par une paire non ordonnée de sommets, appelés les *extrémités* de e. Si l'arête e relie les sommets a et b, on dira que ces sommets sont *adjacents*.
Enfin, on appelle *ordre* d'un graphe le nombre de sommets n de ce graphe. Le *degré* $d(v)$ d'un sommet v est le nombre de sommets qui lui sont adjacents, et le *degré* d'un graphe le degré maximum de tous ses sommets.

Les graphes tirent leur nom du fait qu'on peut les représenter graphiquement : à chaque sommet de G on fait correspondre un point du plan et on relie les points correspondant aux extrémités de chaque arête. Il existe donc une infinité de représentations possibles. Par exemple, les deux dessins qui suivent représentent le même graphe G, avec $V = \{1, 2, 3, 4, 5\}$ et $E = \{(1,3), (1,4), (1,5), (2,3), (3,4), (3,5), (4,5)\}$:



G est un graphe d'ordre 5 ; il possède un sommet de degré 1 (le sommet 2), trois sommets de degré 3 (les sommets 1, 4 et 5) et un sommet de degré 4 (le sommet 3).

Remarque. On peut observer que ce graphe a la particularité de posséder une représentation (celle de droite) pour laquelle les arêtes ne se coupent pas. Un tel graphe est dit *planaire*. Cette notion ne sera pas abordée dans la suite de ce cours.

Un graphe est dit *simple* lorsqu'aucun sommet n'est adjacent à lui-même. Par la suite, nous nous restreindrons à l'étude des graphes simples.

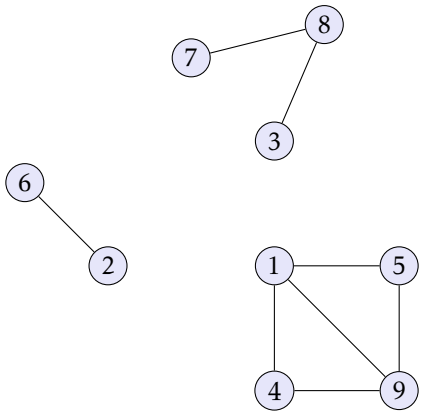
Théorème 1.1 — *Si G est un graphe non orienté simple, La somme des degrés de ses sommets est égal à deux fois le nombre de ses arêtes :*

$$\sum_{v \in V} \deg(v) = 2|E|.$$

■ Chemins

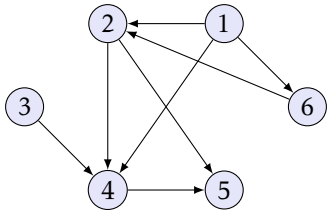
Un *chemin* de longueur k reliant les sommets a et b est une suite finie $x_0 = a, x_1, \dots, x_k = b$ de sommets tel que pour tout $i \in [\![0, k-1]\!]$, la paire (x_i, x_{i+1}) soit une arête de G. Ce chemin est dit *cyclique* lorsque $a = b$.

La *distance* entre deux sommets a et b est la plus petite des longueurs des chemins reliant a et b, si tant est qu'il en existe. Lorsque tous les sommets sont à distance finie les uns des autres, on dit que le graphe est *connexe*. Un graphe non connexe peut être décomposé en plusieurs *composantes connexes*, qui sont des sous-graphes connexes maximaux.
À titre d'exemple, le graphe suivant possède trois composantes connexes :



1.2 Graphes orientés

On obtient un graphe *orienté* en distinguant la paire de sommets (a, b) de la paire (b, a). Dans ce cas, on définit pour chaque sommet v son degré *sortant* $d_+(v)$, égal au nombre d'arcs (dans le cas d'un graphe orienté, on parle souvent d'arc plutôt que d'arêtes) dont il est la première composante, et son degré *entrant* $d_-(v)$, égal au nombre d'arcs dont il est la seconde composante. Par exemple, le graphe qui suit est défini par $V = \{1, 2, 3, 4, 5, 6\}$ et $E = \{(1,2), (1,4), (1,6), (2,4), (2,5), (3,4), (4,5), (6,2)\}$:



Le sommet 1 a un degré sortant égal à 3 et un degré entrant égal à 0, tandis que le sommet 2 a un degré entrant et un degré sortant tous deux égaux à 2.

Les notions de chemin et de distance s'étendent sans difficulté aucune au cas des graphes orientés.

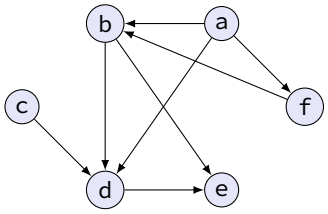
1.3 Mise en œuvre pratique

Deux méthodes principales s'offrent à nous pour représenter un graphe en machine :

- à l'aide de *listes d'adjacence* (à chaque sommet on associe la liste de ses voisins) ;
- à l'aide de *matrice d'adjacence* (le coefficient d'indice (i, j) traduit l'existence ou non d'une liaison entre deux sommets).

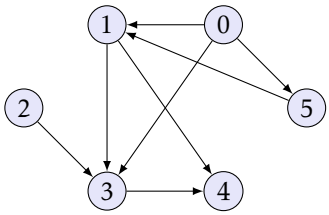
■ Listes d'adjacence

Cette représentation est économique en mémoire, en particulier pour des graphes ayant peu d'arcs ou arêtes. Elle consiste à définir un dictionnaire dont les clefs sont les sommets et les valeurs les listes des successeurs de ces sommets.




```
G = {'a': ['b', 'd', 'f'],
     'b': ['d', 'e'],
     'c': ['d'],
     'd': ['e'],
     'e': [],
     'f': ['b']}
```

Si on numérote les sommets de 0 à $n-1$ on peut même se contenter d'une liste de listes.




```
G = [[1, 3, 5], [3, 4], [3], [4], [], [1]]
```

Désorientation d'un graphe

Un inconvénient potentiel de la représentation par listes d'adjacence est qu'il est difficile de déterminer rapidement si un graphe est non orienté : il faut pour chacune des arêtes vérifier que l'arête réciproque est aussi présente.

Exercice 1

Rédiger une fonction `estNonOriente(G)` qui prend pour argument un graphe représenté par listes d'adjacence et renvoie `True` s'il s'agit d'un graphe non orienté, et `False` sinon.

Désorienter un graphe consiste à calculer le plus petit graphe non orienté G' contenant G. Pour toute arête reliant les sommets a et b il faut donc ajouter l'arête reliant b à a, si celle-ci ne figure pas déjà dans le graphe.

Exercice 2

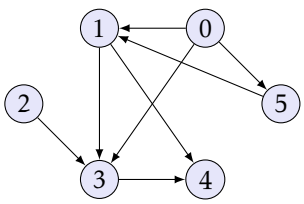
Écrire une fonction `desoriente(G)` qui prend pour argument un graphe orienté G représenté par liste d'adjacence et qui renvoie en nouveau graphe représentant la désorientation de G.

■ Matrice d'adjacence

Lorsque le nombre d'arcs ou d'arêtes est important, on ordonne les sommets $V = \{v_1, v_2, \ldots, v_n\}$ afin de représenterle graphe G par une matrice $M \in \mathcal{M}_n(\{0,1\})$:

$$m_{ij} = \begin{cases} 1 & \text{si } (v_i, v_j) \in E \\ 0 & \text{sinon} \end{cases}$$

autrement dit, d'un point de vue informatique, par un tableau bi-dimensionnel. En outre, compte tenu de l'indexation des tableaux en Python il peut être judicieux de numéroter les sommets de 0 à $n-1$:




```
G = [[0, 1, 0, 1, 0, 1],
     [0, 0, 0, 1, 1, 0],
     [0, 0, 0, 1, 0, 0],
     [0, 0, 0, 0, 1, 0],
     [0, 0, 0, 0, 0, 0],
     [0, 1, 0, 0, 0, 0]]
```

Déterminer si un graphe est orienté est chose aisée (il suffit de vérifier que la matrice est symétrique), de même de la désorientation d'un graphe, mais elle présente l'inconvénient d'occuper beaucoup plus d'espace mémoire, en particulier lorsque le nombre d'arêtes est réduit vis-à-vis du nombre de sommets : si $n = |V|$ et $p = |E|$, le coût spatial de la représentation par matrice d'adjacence est un $O(n^2)$, contre un $O(n+p)$ pour une liste d'adjacence.

Exercice 3

On considère un graphe non orienté dont les sommets sont notés $0, 1, \ldots, n-1$.

a. Rédiger une fonction `mat2lst(G)` qui prend en argument un graphe représenté par matrice d'adjacence et renvoie sa représentation par listes d'adjacence.

b. Rédiger une fonction `lst2mat(G)` qui prend en argument un graphe représenté par listes d'adjacence et renvoie sa représentation par matrice d'adjacence.

2. Parcours d'un graphe

Parcourir un graphe, c'est énumérer l'ensemble des sommets accessibles par un chemin à partir d'un sommet donné, dans l'objectif de leur faire subir un certain traitement. Différentes solutions sont possibles mais en règle générale celles-ci tiennent à jour deux listes : la liste des sommets rencontrés (« déjàVu ») et la liste des sommets en cours de traitement (« àTraiter ») et ces méthodes vont différer par la façon dont sont insérés puis retirés les sommets dans cette structure de données.

Le parcours générique à partir d'un sommet source s_0 se déroule de la façon suivante :

```
procedure PARCOURS(sommet : s_0)
    àTraiter ← s_0
    déjàVu ← s_0
    while àTraiter ≠ ∅ do
        àTraiter → s
        for t ∈ voisins(s) do
            if t ∉ déjàVu then
                àTraiter ← t
                déjàVu ← t
```

S'il y a lieu, le traitement associé à un sommet s peut être placé en différents endroits, en général au moment où le sommet s entre dans « àTraiter » ou au moment où il en sort.

Coût du parcours

Dans ce qui suit, nous considérerons un graphe défini par listes d'adjacence, avec $V = [\![0, n-1]\!]$.

Lors d'un parcours, chaque sommet entre au plus une fois dans la liste des sommets à traiter, et n'en sort donc aussi qu'au plus une fois. Si ces opérations d'entrée et de sortie dans la liste sont de coût constant, le coût total des manipulations de la liste « àTraiter » est un $O(n)$.
Chaque liste d'adjacence est parcourue au plus une fois donc le temps total consacré à scruter les listes de voisinage est un $O(p)$ où $p = |E|$ est le nombre d'arêtes/arcs, *à condition de déterminer si un sommet a déjà été vu en coût constant*. Dans ce cas, le coût total d'un parcours est un $O(n+p)$. Pour réaliser cette condition, la solution que nous adopterons consistera à numéroter les sommets de 0 à $n-1$ et utiliser un tableau booléen pour représenter « déjàVu », destiné à marquer chaque sommet au moment où il entre dans la liste « àTraiter ».

2.1 Parcours en largeur et en profondeur

Nous allons maintenant nous intéresser à deux types de parcours, qui diffèrent par la façon d'extraire les sommets de la liste « àTraiter » : les parcours en largeur et en profondeur.
Comme on peut le constater figure 1, seule la ligne 7 diffère : pour sortir de la liste « àTraiter », le parcours en largeur suit le principe d'une *file* : « premier entré, premier sorti » et le parcours en profondeur le principe d'une *pile* : « premier entré, dernier sorti ». L'ordre du traitement des sommets ne va pas être le même, et chaque parcours aura donc un usage propre.

■ Parcours en largeur

Il consiste à traiter tous les sommets à une distance égale à 1 du sommet initial, puis à une distance égale à 2, à 3, etc. Ce type de parcours est donc idéal pour trouver la plus courte distance entre deux sommets du graphe : il suffit d'ajouter au parcours en largeur un tableau `dist` destiné à enregistrer la distance à la source s_0. Lorsqu'on découvre un nouveau sommet t en examinant les voisins du sommet s, `dist[t]` prend la valeur `dist[s] + 1`.

```
1  def parcoursLargeur(G, s0):
2      n = len(G)
3      dejaVu = [False for _ in range(n)]
4      aTraiter = [s0]
5      dejaVu[s0] = True
6      while len(aTraiter) > 0:
7          s = aTraiter.pop(0)
8          for t in G[s]:
9              if not dejaVu[t]:
10                 aTraiter.append(t)
11                 dejaVu[t] = True
```

```
1  def pseudoParcoursProfondeur(G, s0):
2      n = len(G)
3      dejaVu = [False for _ in range(n)]
4      aTraiter = [s0]
5      dejaVu[s0] = True
6      while len(aTraiter) > 0:
7          s = aTraiter.pop()
8          for t in G[s]:
9              if not dejaVu[t]:
10                 aTraiter.append(t)
11                 dejaVu[t] = True
```

Figure 1 – Parcours en largeur et en pseudo-parcours en profondeur.

```
def distance(G, s0):
    n = len(G)
    dejaVu = [False for _ in range(n)]
    dist = [None for _ in range(n)]
    aTraiter = [s0]
    dejaVu[s0] = True
    dist[s0] = 0
    while len(aTraiter) > 0:
        s = aTraiter.pop(0)
        for t in G[s]:
            if not dejaVu[t]:
                dist[t] = dist[s] + 1
                aTraiter.append(t)
                dejaVu[t] = True
    return dist
```

```
from collections import deque

def distance(G, s0):
    n = len(G)
    dejaVu = [False for _ in range(n)]
    dist = [None for _ in range(n)]
    aTraiter = deque([s0])
    dejaVu[s0] = True
    dist[s0] = 0
    while len(aTraiter) > 0:
        s = aTraiter.popleft()
        for t in G[s]:
            if not dejaVu[t]:
                dist[t] = dist[s] + 1
                aTraiter.append(t)
                dejaVu[t] = True
    return dist
```

Figure 2 – Calcul des distances à un sommet à l'aide d'un parcours en largeur.

Remarque. Nous l'avons dit plus haut, pour réaliser une complexité en $O(n+p)$ il est nécessaire que la méthode permettant de sortir un élément de la liste `aTraiter` soit de complexité constante, ce qui n'est pas le cas de la méthode `.pop(0)`. Pour pallier à ce problème, on peut utiliser la structure `deque` (*double ended queue*) du module `collections`. Cette structure fonctionne peu ou prou comme une liste, si ce n'est qu'elle permet en outre l'insertion et la suppression en temps constant à gauche à l'aide des méthodes `.appendleft()` et `.popleft()`. Comme on peut le constater figure 2, les modifications sont minimes.

■ Parcours en profondeur

Au contraire d'un parcours en largeur, le parcours en profondeur consiste à s'enfoncer le plus possible dans un graphe avant de remontrer vers les sommets déjà rencontrés. La version présentée figure 1 n'est pas un vrai parcours en profondeur car tous les voisins d'un même sommet entrent dans `aTraiter` au même moment. Cette version ne sera donc utilisée que lorsque l'ordre de traitement des sommets n'a pas d'importance, par exemple pour détecter la connexité d'un graphe ou calculer ses composantes connexes (voir figure 3).

En revanche, si un traitement nécessite que l'ordre du parcours en profondeur soit impérativement suivi, la version présentée figure 1 n'est pas suffisante. Dans ce cas de figure, le plus simple est d'adopter une version récursive du parcours en profondeur, sur le modèle présenté figure 4 : la fonction récursive est définie lignes 5-9 et appelée sur le sommet s_0 ligne 11.

```python
def estConnexe(G):
    n = len(G)
    dejaVu = [False for _ in range(n)]
    aTraiter = [0]
    dejaVu[0] = True
    vus = 1
    while len(aTraiter) > 0:
        s = aTraiter.pop()
        for t in G[s]:
            if not dejaVu[t]:
                aTraiter.append(t)
                dejaVu[t] = True
                vus += 1
    return vus == n
```

```python
def composantesConnexes(G):
    n = len(G)
    dejaVu = [False for _ in range(n)]
    composantes = []
    for i in range(n):
        if not dejaVu[i]:
            aTraiter = [i]
            dejaVu[i] = True
            comp = [i]
            while len(aTraiter) > 0:
                s = aTraiter.pop()
                for t in G[s]:
                    if not dejaVu[t]:
                        aTraiter.append(t)
                        dejaVu[t] = True
                        comp.append(t)
            composantes.append(comp)
    return composantes
```

Figure 3 – Test de non connexité d'un graphe et calcul des composantes connexes.

```python
def parcoursProfondeur(G, s0):
    n = len(G)
    dejaVu = [False for _ in range(n)]

    def parcours(s):
        if not dejaVu[s]:
            for t in G[s]:
                dejaVu[t] = True
                parcours(t)

    parcours(s0)
```

Figure 4 – Un parcours en profondeur récursif.

Détection de cycle dans un graphe orienté

L'exemple typique pour lequel l'ordre dans lequel les sommets sont découverts est important est la détection d'un cycle dans un graphe. En effet, dans un « vrai » parcours en profondeur, si un sommet tout juste découvert a pour voisin un sommet en cours de traitement, c'est qu'il y a un cycle. Dans la version récursive, les sommets en cours de traitement sont situés dans la pile de récursivité, qui n'est pas accessible. Aussi on remplace la liste booléenne `dejaVu` par une liste pouvant prendre trois valeurs : `'blanc'` pour les sommets non encore découverts, `'gris'` pour les sommets en cours de traitement, et `'noir'` pour les sommets dont le traitement est fini. On obtient la version présentée figure 5 : la fonction récursive définie lignes 5-14 renvoie `True` en présence d'un cycle et `False` dans le cas contraire. Les lignes 16 à 20 se contentent de tester la présence d'un cycle partant de chaque sommet non encore rencontré du graphe.

Remarque. Cette version de l'algorithme de détection d'un cycle ne s'applique pas au cas d'un graphe non orienté car dès lors que deux sommets u et v sont voisins, le chemin $u \to v \to u$ est détecté comme un cycle par cette version.

3. Graphe pondéré et plus court chemin

Déterminer le plus court chemin entre deux sommets d'un graphe non pondéré n'est pas difficile : il suffit d'effectuer un parcours en largeur à partir d'un des deux sommets jusqu'à trouver l'autre.

Cependant, connaître le nombre minimal d'arêtes à parcourir entre deux sommets n'est pas toujours suffisant :

```python
def cycle(G):
    n = len(G)
    dejaVu = ['blanc' for _ in range(n)]

    def parcours(s):
        dejaVu[s] = 'gris'
        for t in G[s]:
            if dejaVu[t] == 'blanc':
                if parcours(t):
                    return True
            elif dejaVu[t] == 'gris':
                    return True
        dejaVu[s] = 'noir'
        return False

    for i in range(n):
        if dejaVu[i] == 'blanc':
            if parcours(i):
                return True
    return False
```

FIGURE 5 – Détection de cycle à l'aide d'un parcours en profondeur.

de nombreux problèmes ajoutent une *pondération* à chaque arête et définissent le *poids* d'un chemin comme la somme des poids des arêtes qui le composent. Commençons par donner quelques définitions.

3.1 Graphe pondéré

Étant donné un graphe (orienté ou non) $G = (V, E)$, on appelle *pondération* une application $w : E \to \mathbb{R}$, et pour tout $(a, b) \in E$ on dit que $w(a, b)$ est le *poids* de l'arête (a, b).
Il sera par la suite commode de prolonger la définition de w sur $V \times V$ en posant :

$$\forall (a,b) \in (V \times V) \setminus E, \quad w(a,b) = \begin{cases} 0 & \text{si } a = b \\ +\infty & \text{sinon} \end{cases}$$

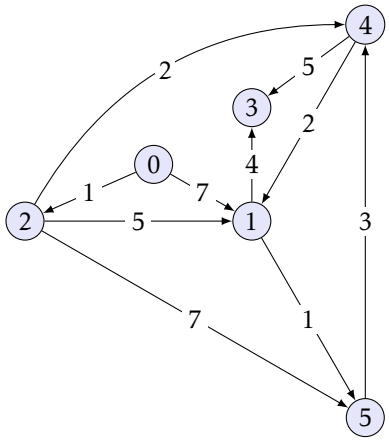
Le *poids* d'un chemin est la somme des poids des arêtes qui le composent. On notera $\delta(a, b)$ le poids du plus court chemin allant de a à b, s'il en existe. Dans le cas contraire, on posera $\delta(a, b) = +\infty$.

Remarque. En présence de poids négatifs il convient de prendre quelques précautions pour assurer l'existence de ce minimum. En particulier, s'il existe un chemin menant de a et b et comprenant un circuit fermé de poids strictement négatif, il convient de poser $\delta(a, b) = -\infty$ car dans ce cas on peut faire indéfiniment décroitre le poids de ce chemin en multipliant les passages par cette boucle. Par la suite, nous éviterons cette situation en supposant *que tous les poids sont positifs ou nuls*.

■ Représentation en mémoire

Dans cette section on représentera un graphe pondéré en adaptant la représentation par matrice d'adjacence : celle-ci sera maintenant une matrice à valeurs dans $\mathbb{R}_+ \cup \{+\infty\}$ contenant les poids des différentes arêtes (illustration figure 6).

Remarque. Rappelons que Python offre la possibilité de définir un flottant représentant $+\infty$ en écrivant `float('inf')`.




```
inf = float('inf')

G = [[0, 7, 1, inf, inf, inf],
     [inf, 0, inf, 4, inf, 1],
     [inf, 5, 0, inf, 2, 7],
     [inf, inf, inf, 0, inf, inf],
     [inf, 2, inf, 5, 0, inf],
     [inf, inf, inf, inf, 3, 0]]
```

FIGURE 6 – Un exemple de représentation d'un graphe pondéré.

3.2 L'algorithme de DIJKSTRA

Cet algorithme résout le problème des plus courts chemins à partir d'une source $s_0 \in V$: il prend pour arguments un graphe pondéré G et un sommet s_0 et renvoie un tableau contenant l'ensemble des distances minimales reliant s_0 à un sommet quelconque de G. Cet algorithme nécessite de supposer que tous les poids sont positifs ou nuls.

■ Principe de l'algorithme

L'algorithme de Dijkstra remplit progressivement un tableau d de longueur n de sorte qu'à la fin de cet algorithme $d[v]$ soit égal à $\delta(s_0, v)$, le poids du plus court chemin entre s_0 et v. Pour ce faire, on fait évoluer une partition de $[\![1, n]\!]$: un sous-ensemble S (initialement vide) et son complémentaire $\overline{S}$.
L'ensemble S est destiné à représenter les sommets dont on a déterminé dans le tableau d le poids du chemin minimal à partir de s_0 (donc dont le traitement est terminé). Pour les éléments de $\overline{S}$, le tableau d contient le poids du plus court chemin *ne passant que par des sommets appartenant à* S. Ainsi, le tableau d est initialement défini par :

$$\forall v \in V, \quad d[v] = \begin{cases} 0 & \text{si } v = s_0 \\ +\infty & \text{sinon} \end{cases}$$

À chaque itération on choisit le sommet s de $\overline{S}$ dont la valeur associée dans le tableau d est minimale pour le transférer dans S, et on met à jour le tableau d en remplaçant $d[v]$ par $\min(d[v], d[s] + w(s, v))$. Ainsi, chaque élément de $\overline{S}$ va progressivement être transféré dans S.

On représente classiquement la partition $S \cup \overline{S}$ par un tableau de type `dejaVu`, de sorte que l'algorithme de Dijkstra s'écrit :

```
1  def dijkstra(G, s0):
2      n = len(G)
3      dejaVu = [False for _ in range(n)]
4      d = [float('inf') for _ in range(n)]
5      d[s0] = 0
6      for _ in range(n):
7          mini = float('inf')
8          for v in range(n):
9              if not dejaVu[v] and d[v] <= mini:
10                 s, mini = v, d[v]
11         dejaVu[s] = True
12         for v in range(n):
13             if not dejaVu[v]:
14                 d[v] = min(d[v], d[s] + G[s][v])
15     return d
```

Les lignes 7-10 ont pour objet de déterminer parmi les sommets non traités celui dont la valeur associée dans le tableau d est minimale. Les lignes 11-14 mettent à jour le tableau d en conséquence.
Cette démarche est répétée n fois de sorte que tous les sommets soient traités.

Observons comment se déroule cet algorithme sur le graphe représenté figure 6, à partir du sommet 0, en observant l'évolution du tableau d au cours de l'algorithme :

Initialisation

	0	1	2	3	4	5
déjàVu	·	·	·	·	·	·
d	0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$

étape 1

	0	1	2	3	4	5
déjàVu	×	·	·	·	·	·
d	0	7	1	$+\infty$	$+\infty$	$+\infty$

Le sommet 0 est traité, l'algorithme a découvert deux chemins vers les sommets 1 et 2.

étape 2

	0	1	2	3	4	5
déjàVu	×	·	×	·	·	·
d	0	6	1	$+\infty$	3	8

Le sommet 2 est traité, l'algorithme a découvert deux chemins vers les sommets 4 et 5 et un chemin plus court vers le sommet 1.

étape 3

	0	1	2	3	4	5
déjàVu	×	·	×	·	×	·
d	0	5	1	8	3	8

Le sommet 4 est traité, l'algorithme a découvert un chemin vers le sommet 3 et un chemin plus court vers le sommet 1.

étape 4

	0	1	2	3	4	5
déjàVu	×	×	×	·	×	·
d	0	5	1	8	3	6

Le sommet 1 est traité, l'algorithme a découvert un chemin plus court vers le sommet 5.

étape 5

	0	1	2	3	4	5
déjàVu	×	×	×	·	×	×
d	0	5	1	8	3	6

Le sommet 5 est traité, aucun nouveau chemin n'est découvert.

étape 6

	0	1	2	3	4	5
déjàVu	×	×	×	×	×	×
d	0	5	1	8	3	6

Le sommet 3 est traité, l'algorithme est terminé.

■ Complexité de l'algorithme de Dijkstra

Tel que nous l'avons écrit, l'algorithme de Dijkstra a une complexité quadratique en $O(n^2)$. Il est possible de mieux faire en utilisant des structures de données mieux adaptées pour représenter l'ensemble S, mais ces considérations sont hors-programme.

■ Détermination des chemins de poids minimaux

Pour garder trace du chemin parcouru, il suffit d'utiliser un tableau c que l'on modifie en même temps que d : lorsque $d[v]$ est remplacé par $d[v]+w(s,v)$, on mémorise s dans $c[v]$. Ainsi, à la fin de l'algorithme $c[v]$ contient le sommet précédant v dans un chemin minimal allant de s_0 à v, ce qui permet de reconstituer le chemin optimal une fois l'algorithme terminé.

Pour obtenir le chemin minimal on modifie donc l'algorithme de Dijkstra pour qu'il crée puis renvoie le tableau c, et on utilise cette version modifiée pour obtenir le plus court chemin entre deux sommets a et b :

```
def predecesseurs(G, s0):
    n = len(G)
    dejaVu = [False for _ in range(n)]
    d = [float('inf') for _ in range(n)]
    c = [None for _ in range(n)]
    d[s0] = 0
    for _ in range(n):
        mini = float('inf')
        for v in range(n):
            if not dejaVu[v] and d[v] <= mini:
                s, mini = v, d[v]
        dejaVu[s] = True
        for v in range(n):
            if not dejaVu[v] and d[s] + G[s][v] < d[v]:
                d[v] = d[s] + G[s][v]
                c[v] = s
    return c
```

```
def plusCourtChemin(G, a, b):
    pred = predecesseurs(G, a)
    if pred[b] is not None:
        chemin = [b]
        while True:
            chemin.append(pred[chemin[-1]])
            if chemin[-1] == a:
                return list(reversed(chemin))
```

3.3 L'algorithme A*

Dans la pratique, il est rare que tous les chemins de poids minimaux issus de s_0 nous intéressent ; en général seuls la distance et le chemin minimal entre une source a et un but b nous intéressent. On peut bien sûr utiliser l'algorithme de Dijkstra à partir de a et l'arrêter dès que le chemin minimal reliant a et b est trouvé, mais cette optimisation ne permet guère d'accélérer le processus lorsque l'objectif b fait partie des derniers points traités.

De manière absolue, on ne peut guère faire mieux que l'algorithme de Dijkstra, mais dans certains cas il est possible d'éliminer bon nombre de calculs inutiles en utilisant une *heuristique* ; c'est ce que fait l'algorithme A* (prononcer « A star »).

Cet algorithme suppose connue une fonction h (l'heuristique) capable d'évaluer grossièrement le poids du plus court chemin entre tout sommet s et le but b. L'algorithme fonctionne alors comme celui de Dijkstra, si ce n'est qu'au lieu de sélectionner à chaque étape le sommet s minimisant la quantité $d[s]$, on sélectionne celui minimisant la quantité $h(s)+d[s]$.

Nous admettrons que si la fonction h vérifie : $\forall s \in V,\ 0 \leqslant h(s) \leqslant \delta(s,b)$ alors l'algorithme A renvoie la distance minimale entre a et b.*

Remarque. La difficulté consiste à déterminer une fonction h vérifiant cette condition. Dans le cas d'un graphe dont les poids sont générés aléatoirement, cette condition parait bien difficile à satisfaire. En revanche, *sur un graphe constitué de points du plan et dont les arêtes sont pondérées par leurs longueurs respectives*, la distance euclidienne constitue une heuristique admissible, qui plus est facile à calculer. C'est exactement la situation d'un GPS d'une voiture vous menant d'un point a à un point b.