♦ TP9 – Compléments sur les graphes

Les graphes sont des objets théoriques étudiés à la fois en mathématiques et en informatique qui peuvent modéliser de nombreuses situations de la vie courante (plan du métro, carte routière, organigramme d'une entreprise, liens entre les différentes pages d'un site internet, liens entre les profils sur un réseau social, etc...)

Le principe général d'un graphe est de considérer des objets (stations de métro, villes, postes dans une entreprise, pages d'un site internet, profils sur un réseau social) modélisés par des points appelés les sommets du graphe et de représenter les liens entre ces différents objets par des segments ou des arcs entre les sommets appelés les arêtes du graphe.

I. — Rappels théoriques sur les graphes

1) Notion de graphe

a) Graphe non orienté

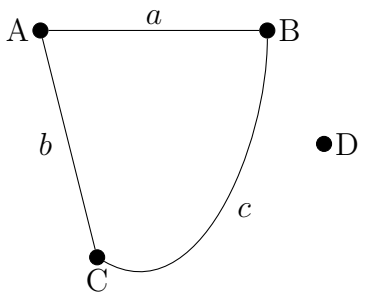
Définition 1

Un graphe (simple) non orienté $\mathscr{G}$ est la donnée de deux ensembles :

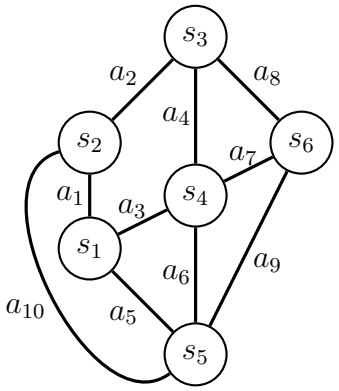
1. un ensemble fini et non vide $\mathcal{S}$ dont les éléments sont appelés les **sommets** du graphe ;
2. un ensemble fini (éventuellement vide) $\mathcal{A}$ formé de paires d'éléments distincts de $\mathcal{S}$. Les éléments de $\mathcal{A}$ sont appelés les **arêtes** du graphe.
3. Si $a = \{A, B\}$ est une arête du graphe, on dit que les sommets A et B sont les **extrémités** de a.

Remarque 2. La meilleure façon de présenter un graphe est d'en donner une représentation graphique. Un sommet est représenté par un point (ou un cercle) et une arête par un arc (ou un segment) reliant deux sommets.

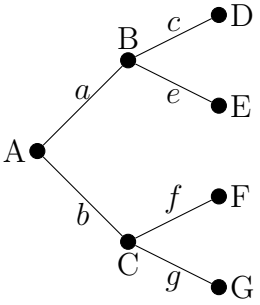
Exemple 3.



Graphe $\mathscr{G}_1$



Graphe $\mathscr{G}_2$



Graphe $\mathscr{G}_3$

Question 1. Déterminer le nombre de sommets et le nombre d'arêtes des graphes $\mathscr{G}_1$, $\mathscr{G}_2$ et $\mathscr{G}_3$.

Définition 4

Le nombre de sommets d'un graphe $\mathscr{G}$ est appelé l'**ordre** de $\mathscr{G}$.

b) Graphe orienté

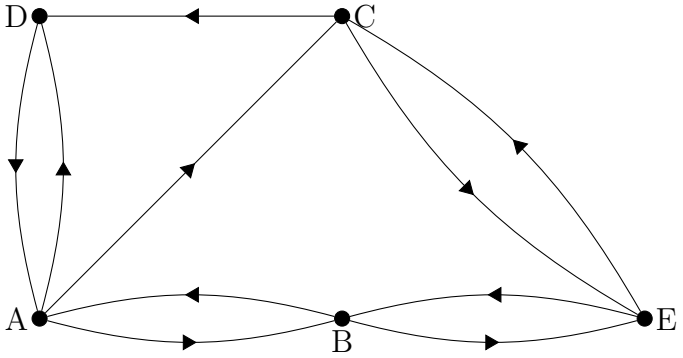
Il est parfois nécessaire dans un graphe d'orienter les arêtes c'est-à-dire d'indiquer un sens de parcours entre deux sommets. C'est le cas, par exemple, si on veut représenter le plan d'une ville avec des sens uniques.

Définition 5

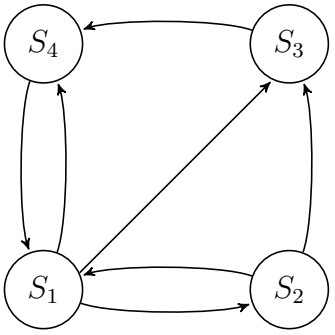
Un **graphe orienté** est un graphe dans lequel les arêtes ne sont plus des paires de sommets mais des couples de sommets. On parle alors d'arc orienté plutôt que d'arête. De plus, si $c = (\mathrm{A}, \mathrm{B})$ est un arc orienté formé par deux sommets A et B, on dit que A est l'**origine** (extrémité initiale) de c et que B est l'**extrémité terminale** (ou finale) de c. On dit également que c va de A vers B.

Remarque 6. Graphiquement, on traduit l'orientation du graphe par des flèches sur les arcs.

Exemple 7. Les graphes suivants sont orientés.



Graphe $\mathscr{G}_4$



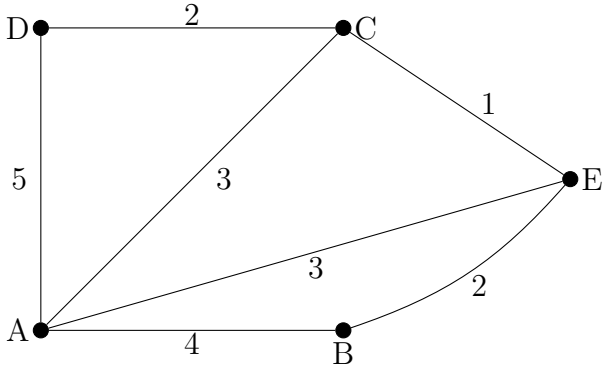
Graphe $\mathscr{G}_5$

c) Graphe pondéré

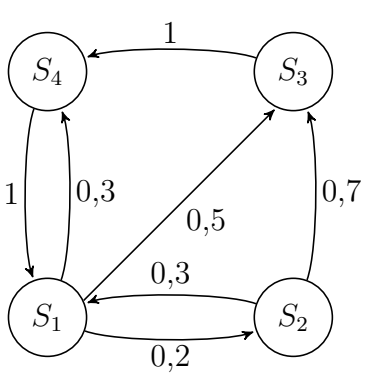
Définition 8

Un graphe (simple) **pondéré** est un graphe (orienté ou non) dans lequel un nombre strictement positif est attribué à chaque arête. Ce nombre est appelé le **poids** de l'arête.

Exemple 9. Les graphes suivants sont des graphes pondérés.



Graphe $\mathscr{G}_6$



Graphe $\mathscr{G}_7$

2) Liste et matrice d'adjacences

Définition 10

Soit $\mathscr{G}$ un graphe, A et B deux sommets de $\mathscr{G}$. On dit que B est **adjacent** à A s'il existe :

- une arête ayant pour extrémités A et B dans le cas d'un graphe non orienté ;
- une arête de A vers B dans le cas d'un graphe orienté.

Exemple 11. Dans le graphe $\mathscr{G}_1$ de l'exemple 3, C est adjacent à A et B mais pas à D. Dans le graphe $\mathscr{G}_5$, S_3 est adjacent à S_1 mais S_1 n'est pas adjacent à S_3.

Définition 12

Étant donné un graphe $\mathscr{G}$ et un sommet A de $\mathscr{G}$, on appelle **liste d'adjacence** de A comme la liste des sommets adjacents à A.

Exemple 13. On peut représenter un graphe non pondéré par la liste de listes d'adjacence de ses sommets. Par exemple, en ordonnant les sommets du graphes $\mathscr{G}_1$ par ordre alphabétique, la liste des listes d'adjacence de ce graphe est [[B,C], [A,C], [A,B], [ ]] car B et C sont adjacents à A, A et C sont adjacents à B, A et B sont adjacents à C et aucun sommet n'est adjacent à D.

Question 2. Déterminer les listes de listes d'adjacence des graphes $\mathscr{G}_2$, $\mathscr{G}_3$, $\mathscr{G}_4$ et $\mathscr{G}_5$ en ordonnant si besoin les sommets dans l'ordre alphabétique.

Définition 14

On considère un graphe $\mathscr{G}$ ayant n sommets numérotés s_1, s_2, ..., s_n.

1. Si $\mathscr{G}$ n'est pas orienté, on appelle **matrice d'adjacence** de $\mathscr{G}$ la matrice $M = (m_{ij})$ où m_{ij} est le nombre d'arêtes ayant comme extrémités s_i et s_j. Autrement dit, $m_{ij} = 1$ si s_i et s_j sont adjacents et $m_{ij} = 0$ sinon.
2. Si $\mathscr{G}$ est orienté, on appelle **matrice d'adjacence** de $\mathscr{G}$ la matrice $M = (m_{ij})$ où m_{ij} est le nombre d'arêtes ayant comme origine s_i et comme extrémité terminale s_j. Autrement dit, $m_{ij} = 1$ s'il y a une arête de s_i vers s_j et $m_{ij} = 0$ sinon.
3. Si $\mathscr{G}$ est pondéré, on appelle **matrice d'adjacence** de $\mathscr{G}$ la matrice $M = (m_{ij})$ où m_{ij} est égale au poids de l'arête entre s_i et s_j si ces deux sommets sont adjacents et à 0 sinon (en tenant compte de l'orientation si le graphe est orienté).

Question 3. Déterminer la matrice d'adjacence des graphes des exemples 3, 7 et 9, en ordonnant si besoin les points selon l'ordre alphabétique.

Remarque 15.

1. Une matrice d'adjacence est toujours une matrice carrée dont l'ordre est l'ordre du graphe.
2. Un graphe est entièrement déterminé par sa matrice d'adjacence i.e. à chaque graphe correspond une unique matrice d'adjacence (à l'ordre des sommets près) et, inversement, à chaque matrice d'adjacance, correspond un unique graphe (aux noms des sommets près).

II. — Représentation d'un graphe en Python

1) Rappels sur les dictionnaires

Un dictionnaire est une structure ayant son propre type (`dict`) et qui représente un ensemble (non ordonné) de couples `clé:valeur`. Un dictionnaire est délimité par des accolades et ses éléments (qui sont des couples) sont séparés par des virgules.

Considérons la chaîne de caractères `ch="AGCTTACGATACTT"`. On peut créer un dictionnaire `nb_occ` pour répertorier le nombre d'occurences de chacune de lettres `A`, `C`, `G` et `T` dans la chaîne `ch`. Cela donne `nb_occ = { "A":4, "C":3, "G":2, "T":5 }`. Ce dictionnaire possède 4 clés `"A"`, `"C"`, `"G"` et `"T"` dont les valeurs associées sont respectivement `4`, `3`, `2`, `5`.

Le dictionnaire vide se définit par `{}`.

Un dictionnaire est un objet mutable (on peut modifier les valeurs ou ajouter des couples clé/valeur) mais, contrairement à une liste, il n'est pas ordonné et les couples n'ont pas d'indice. Pour accéder à une valeur, on utilise sa clé. Par exemple, en reprenant le dictionnaire `nb_occ`, si on veut connaître le nombre de `"A"` dans la chaîne `ch`, on utilisera la syntaxe `nb_occ["A"]` qui renvoie `4`. Si on modifie la chaîne `ch` par la syntaxe `ch += "G"` alors `ch` devient `ch="AGCTTACGATACTTG"` et, sans reconstruire complètement le dictionnaire `nb_occ`, on peut simplement modifier la valeur de la clé `"G"` grâce à l'instruction `nb_occ["G"] += 1`. Pour tester si une clé `c` est présente dans un dictionnaire `dic`, on peut utiliser la syntaxe `c in dic` qui renvoie un booléen. Ainsi, `"A" in nb_occ` renvoie `True` alors que `"B" in nb_occ` renvoie `False`.

Dans un dictionnaire, les clés sont des objets de types non mutables (des nombres, des chaînes de caractères, des booléens mais pas des listes) et les valeurs peuvent être de types quelconques. Dans un même dictionnaire, les différentes clés peuvent être de types différents (et de même pour les valeurs).

Enfin, un dictionnaire est itérable (c'est-à-dire, on peut le parcourir selon ses clés) et peut se définir en compréhension. Si on reprend l'exemple précédent, on peut obtenir la liste des clés de `nb_occ` par la syntaxe `[e for e in nb_occ]` et la liste des valeurs associées par la syntaxe `[nb_occ[e] for e in nb_occ]`. De même, si on a une liste de chaînes de caractères `L = ["CCT", "ACGTTA", "ATTCTGA", "AG"]`, on peut définir un dictionnaire qui associe à chaque chaîne sa longueur par la syntaxe suivante `{ ch:len(ch) for ch in L }`.

Question 4. La commande suivante permet de créer un dictionnaire donnant la masse moyenne en kg de certains animaux :

```
masse = {'girafe':1100, 'tigre':250, 'singe':70}.
```

1. Prédire puis vérifier à l'aide de l'ordinateur ce que renvoie les commandes suivantes.

   `type(masse)` `len(masse)` `masse[1]` `masse['tigre']` `masse['souris']`.

2. **a.** On exécute l'instruction `masse['souris'] = 0.02`. Que vaut à présent `len(masse)` ? Vérifier à l'aide de l'ordinateur.

   b. On exécute l'instruction `masse['souris'] = 0.03`. Que vaut à présent `len(masse)` ? Vérifier à l'aide de l'ordinateur.

3. Créer un dictionnaire `masse_gramme` qui donne la masse moyenne en gramme des animaux du dictionnaire `masse`.

Question 5. Écrire une fonction `chaines_2_lettres` qui prend en argument une chaîne de caractères `adn` composée avec les lettres `"A"`, `"C"`, `"G"` et `"T"` et qui renvoie un dictionnaire dont les clés sont les chaines de caractères de deux lettres consécutives apparaissant dans `adn` et les valeurs sont les nombres d'occurrences de ces chaînes. Par exemple, `chaine_2_lettres("ACCTAGCCCTA")` doit renvoyer le dictionnaire

`{"AC":1, "CC":3, "CT":2, "TA":2, "AG":1, "GC":1}`.

2) Représentation par listes d'adjacence

a) Cas des graphes non pondérés

Un graphe non pondéré peut se représenter par les listes d'adjacence de ses sommets. En Python, celles-ci peuvent être implémentées soit à l'aide d'une liste soit à l'aide d'un dictionnaire. L'avantage du dictionnaire réside dans le fait qu'il permet de donner le nom d'un sommet comme clé alors que, si on utilise une liste, il faut convenir d'un ordre des sommets.

Exemple 16. Reprenons le graphe $\mathscr{G}_1$.

- En convenant que les sommets sont ordonnés par ordre alphabétique, on peut représenter les listes d'adjacence par la liste de listes `[['B','C'], ['A','C'], ['A','B'], []]`.
- À l'aide d'un dictionnaire, on peut représenter les listes d'adjacence par

`{ 'A':['B','C'], 'B':['A','C'], 'C':['A','B'], 'D':[] }`.

Question 6. Implémenter sur l'ordinateur les listes d'adjacences des graphes $\mathscr{G}_2$, $\mathscr{G}_3$, $\mathscr{G}_4$ et $\mathscr{G}_5$ sous forme de listes de listes et sous forme de dictionnaires.

b) Cas des graphes pondérés

Pour un graphe pondéré, les listes d'adjacence de chaque sommet ne suffisent pas à décrire le graphe totalement puisqu'elles ne donnent pas le poids de chaque arête. Dans ce cas, on peut utiliser un dictionnaire dont les clés sont les sommets et les valeurs sont des dictionnaires dont les clés sont les sommets adjacents et les valeurs sont les poids des arêtes.

Exemple 17. On peut représenter le graphe $\mathscr{G}_6$ par le dictionnaire suivant :

```
{ 'A':{'B':4, 'E':3, 'C':3, 'D':5}, 'B':{'A':4, 'E':2}, 'E':{'B':2, 'A':3, 'C':1},
                              'C' :{'D' :2, 'A' :3, 'E' :1}, 'D' :{'A' :5, 'C' :2} }
```

On remarquera que, comme un dictionnaire n'est pas ordonnée, rien n'oblige à écrire les sommets dans l'ordre alphabétique.

Question 7. Implémenter dans Python sous forme de dictionnaire les listes d'adjacence du graphe $\mathscr{G}_7$.

3) Représentation par matrice d'adjacence

Une autre façon de représenter un graphe est d'utiliser une matrice d'adjacence qu'on peut implémenter en Python par une liste de listes.

Ainsi, en ordonnant les sommets par ordre alphabétique, la matrice d'adjacence du graphe $\mathscr{G}_1$ peut s'implémenter dans Python par le liste de listes

`[[0,1,1,0],[1,0,1,0],[1,1,0,0],[0,0,0,0]]`.

Question 8. Implémenter dans Python les matrices d'adjacence des graphes $\mathscr{G}_2$, $\mathscr{G}_3$, $\mathscr{G}_4$, $\mathscr{G}_5$, $\mathscr{G}_6$ et $\mathscr{G}_7$.

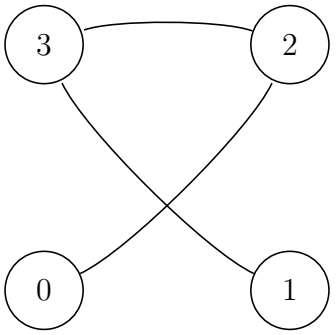
4) Exercices

Question 9. On considère le graphe $\mathscr{G}$ représenté en Python par la liste de listes d'adjacence suivante :

`L = [[1], [0, 3, 4], [1,3], [1], [0,3]]`.

1. Ce graphe est-il orienté ? Est-il pondéré ?
2. Représenter sur votre feuille le graphe $\mathscr{G}$.
3. Écrire en Python la matrice `Mat` d'adjacence de $\mathscr{G}$ sous forme de liste de listes.
4. Quelle commande Python permet d'obtenir à partir de `L` l'ordre du graphe `G` ?
5. Quelle commande Python permet d'obtenir à partir de `L` le nombre de sommets adjacents au sommet 2 ?

Question 10. Écrire dans la console Python la liste de listes représentant le matrice d'adjacence du graphe suivant :



Question 11. On considère un graphe dont les sommets sont A, B, C, D et E et la matrice d'adjacence est

$$M = \begin{pmatrix} 0 & 3 & 5 & 0 & 0 \\ 1 & 0 & 0 & 3 & 2 \\ 0 & 5 & 0 & 2 & 6 \\ 0 & 0 & 0 & 0 & 4 \\ 0 & 0 & 3 & 1 & 0 \end{pmatrix}$$

1. Ce graphe est-il orienté ? Est-il pondéré ?
2. Représenter ce graphe sur votre feuille.
3. Écrire dans la console Python un dictionnaire représentant ce graphe.
4. Écrire dans la console Python une liste de listes représentant ce graphe.

Question 12. Dans cet exercice, on suppose que les sommets d'un graphe d'ordre n sont numérotés 0, 1, ..., $n-1$ et que, dans les listes d'adjacence, les sommets sont représentés par ces numéros.

1. Écrire une fonction `liste_vers_matrice` qui prend en argument une liste de listes représentant les listes d'adjacences d'un graphe (non pondéré) et qui renvoie la liste de listes représentant la matrice d'adjacence de ce graphe.
2. Écrire une fonction `matrice_vers_liste` qui prend en argument une liste de listes représentant la matrice d'adjacence d'un graphe (non pondéré) et qui renvoie la liste de listes représentant les listes d'adjacences de ce graphe.
3. En reprenant la liste `L` et la matrice `Mat` de la **Question 9**, vérifier que `liste_vers_matrice(L)` renvoie `Mat` et que `matrice_vers_liste(Mat)` renvoie `L`.

III. — Parcours d'un graphe

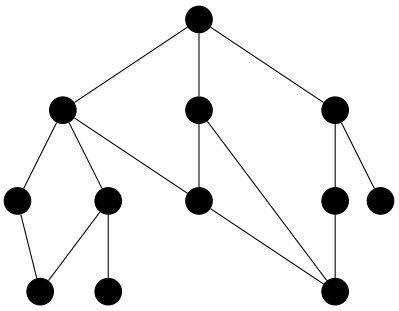
Comme nous l'avons dit en introduction, un graphe peut modéliser un site internet : les sommets représentent les pages du site et les arêtes représentent les liens hypertextes qui permettent de naviguer d'une page à l'autre. Imaginons qu'on recherche une information sur ce site à l'aide d'un mot-clé. Il va falloir parcourir toutes les pages du site et chercher sur chacune d'elles si le mot-clé apparaît. Pour être sûr de ne pas rater une information, il faut être certain d'avoir parcouru toutes les pages du site. Dans cette situation, il est important d'avoir un algorithme qui nous assure donc de parcourir toutes les pages i.e., du point du vue du graphe, tous les sommets.

Il existe plusieurs façons de parcourir un graphe. Nous allons en voir deux : le parcours en largeur et le parcours en profondeur.

1) Parcours en largeur

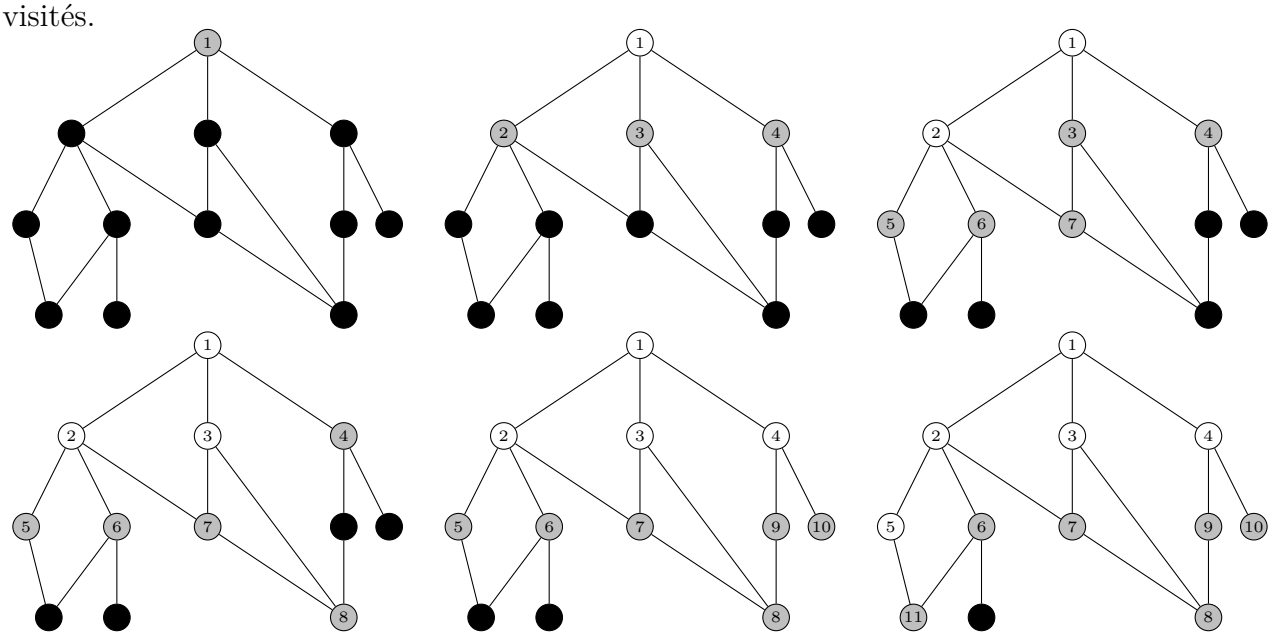
Le principe général du parcours en largeur est le suivant : on part d'un sommet du graphe, on visite tous les sommets qui lui sont adjacents puis on visite tous les sommets adjacents à ces sommets et qui n'ont pas encore été visités et ainsi de suite.

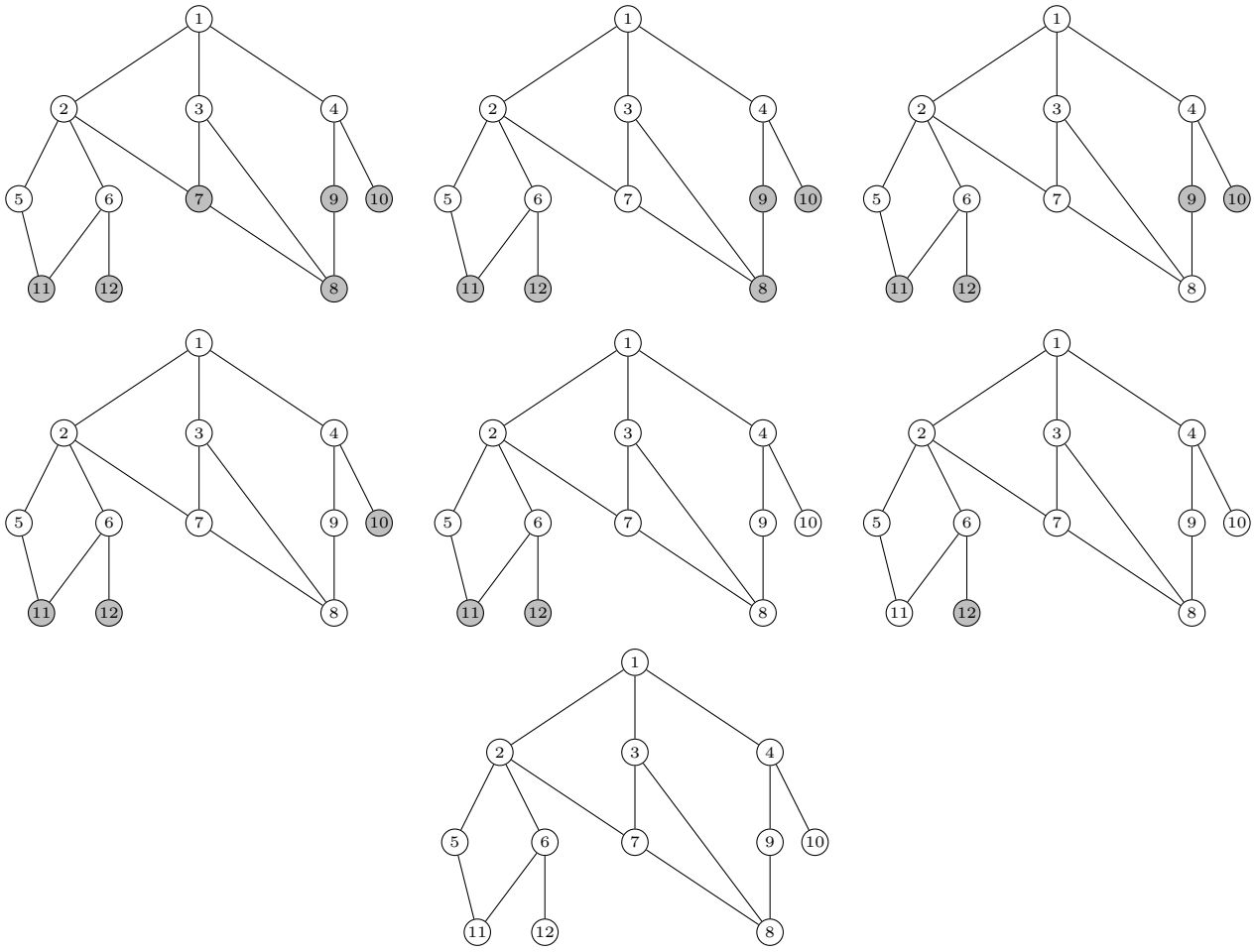
Mettons-le en œuvre sur l'exemple suivant :



Le premier sommet est purement arbitraire. Dans l'exemple de l'introduction, il peut s'agir de la page d'accueil du site internet. Supposons qu'ici ce soit le sommet le plus haut dans le graphe.

Dans ce qui suit, les sommets noirs sont les sommets qui ne sont pas encore visités, les sommets gris sont les prochains sommets à visiter et les sommets blancs sont les sommets déjà visités.





Pour programmer le parcours en largeur, on va utiliser le principe informatique de la file. Il s'agit du même principe que dans une file d'attente à un guichet : les gens arrivent les uns derrière les autres, le premier arrivé est le premier servi ; une fois qu'il est servi, il se retire de la file, puis le second arrivé est servi ; et ainsi de suite.

Le principe de programmation du parcours en largeur va être le suivant :

1. on enfile le premier sommet (i.e. on le met au début de la file) ;

2. on enfile à la suite tous les sommets adjacents au premier sommet qui ne sont pas déjà présents dans la file ;

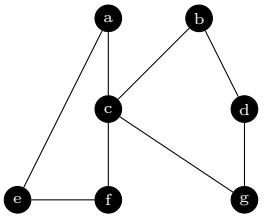
3. on défile le premier sommet (i.e. on le retire de la file) ;

4. tant que la file n'est pas vide, on réitère les points **2.** et **3.**.

Plus concrètement, on va partir d'un graphe représenté par un dictionnaire et on va utiliser trois variables :

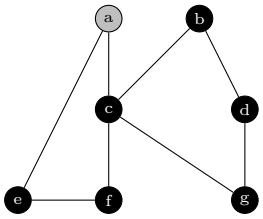
- une liste `File` représentant la file telle que `File[0]` est le dernier sommet visité et les éléments suivants de `File` sont les sommets adjacents à `File[0]` ;
- un dictionnaire `Predecesseur` tel que, à chaque étape, les clés de `Predecesseur` sont les sommets `S` visités avec, comme valeur associée à `S`, le sommet à partir duquel on est arrivé à `S` i.e. le sommet qui était en tête de file lorsqu'on a ajouté `S` à la liste `File` ;
- un dictionnaire `Couleur` dont les clés sont les sommets du graphe et les valeurs associées sont des couleurs : `'noir'` si le sommet n'est pas encore passé dans la file (il ne fait pas partie des sommets déjà visités ou des prochains sommets à visiter), `'gris'` si le sommet est dans la file (il est en cours de visite ou fait partie des prochains sommets à visiter) et `'blanc'` si le sommet est sortie de la file (il a été visité).

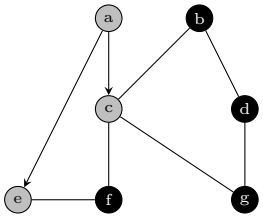
On considère le graphe suivant.

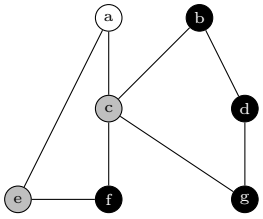


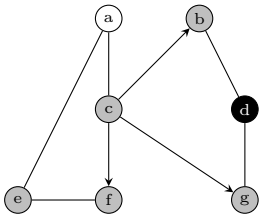
Question 13. Représenter ce graphe à l'aide d'un dictionnaire `G`.

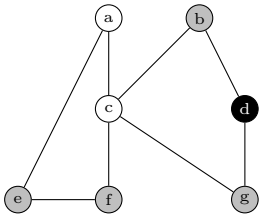
Question 14. Compléter le tableau en donnant les valeurs des variables `File`, `Predecesseur` et `Couleur` à chaque étape du parcours en largeur du graphe commençant par le sommet a.

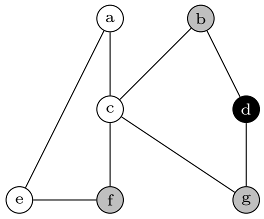
	`File = ['a']`
	
	`Predecesseur = { 'a':None }`
	
	`Couleur = { 'a':'gris', 'b':'noir', 'c':'noir', 'd':'noir', 'e':'noir', 'f':'noir', 'g':'noir' }`

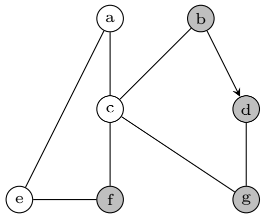
	`File = ['a', 'c', 'e']`
	
	`Predecesseur = { 'a':None, 'c':'a', 'e':'a' }`
	
	`Couleur = { 'a':'gris', 'b':'noir', 'c':'gris', 'd':'noir', 'e':'gris', 'f':'noir', 'g':'noir' }`

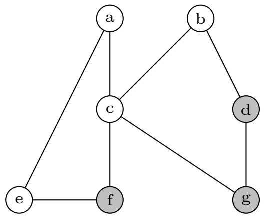
	`File = ['c', 'e']`
	
	`Predecesseur = { 'a':None, 'c':'a', 'e':'a' }`
	
	`Couleur = { 'a':'blanc', 'b':'noir', 'c':'gris', 'd':'noir', 'e':'gris', 'f':'noir', 'g':'noir' }`

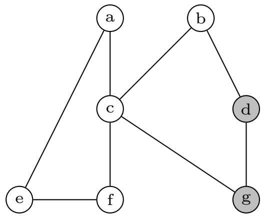
	`File =`
	
	`Predecesseur =`
	
	`Couleur =`

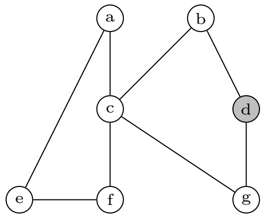
	`File =`
	
	`Predecesseur =`
	
	`Couleur =`

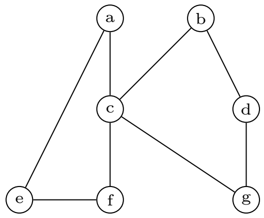
	`File =`
	
	`Predecesseur =`
	
	`Couleur =`

	`File =`
	
	`Predecesseur =`
	
	`Couleur =`

	`File =`
	
	`Predecesseur =`
	
	`Couleur =`

	`File =`
	
	`Predecesseur =`
	
	`Couleur =`

	`File =`
	
	`Predecesseur =`
	
	`Couleur =`

	`File =`
	
	`Predecesseur =`
	
	`Couleur =`

Question 15. Compléter le script de la fonction `parcours_largeur` suivante qui prend en argument un graphe représenté par un dictionnaire `G` et un sommet `S` et qui détermine un parcours en largeur de `G` en partant du sommet `S`.

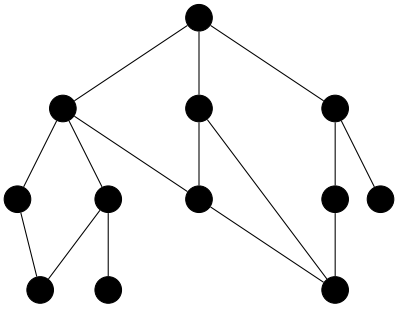
```
def parcours_largeur(G,S):
  File = ...
  Couleur = { ... for x in G }
  Predecesseur, Couleur[S] = {S:...}, ...
  while ...:
    u = File[0]
    for v in ...:
      if Couleur[v] == ...:
        Predecesseur[v], Couleur[v] = ..., ...
        File ...
    File.pop(0)
    Couleur[u] = ...
  return Predecesseur
```

Question 16. Tester la fonction `parcours_largeur` sur le graphe de la question **13** et vérifier qu'on trouve le même parcours (à l'ordre près) que dans la question **14**.

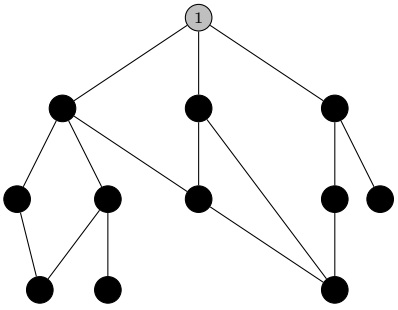
2) Parcours en profondeur

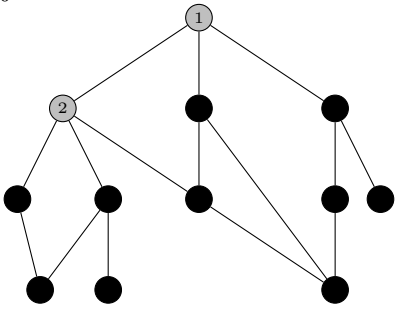
Le principe général du parcours en profondeur est le suivant : on part d'un sommet S_1 du graphe, on visite un sommet S_2 adjacent à S_1 puis un sommet S_3 (non encore visité) adjacent à S_2 et on continue ainsi tant que c'est possible. Lorsqu'on arrive à un sommet S_k qui n'a pas de sommets adjacents non visités, on remonte la liste S_k, S_{k-1}, ..., jusqu'à un sommet qui possède un sommet adjacent non visité et on réitère le processus tant que c'est possible.

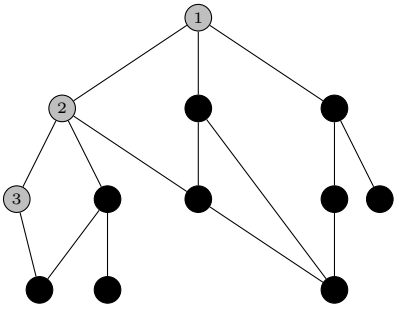
Mettons-le en œuvre sur l'exemple suivant :

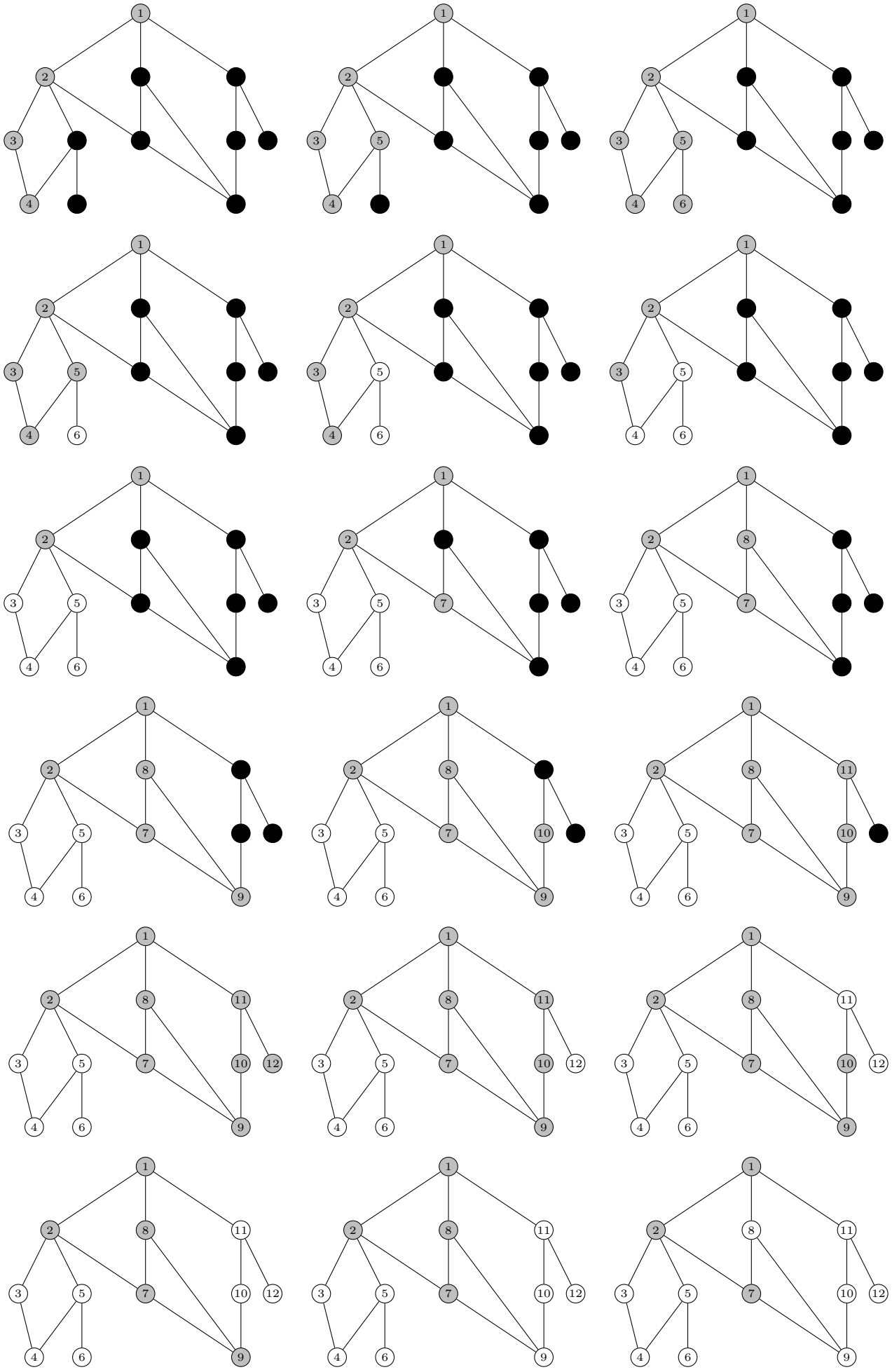


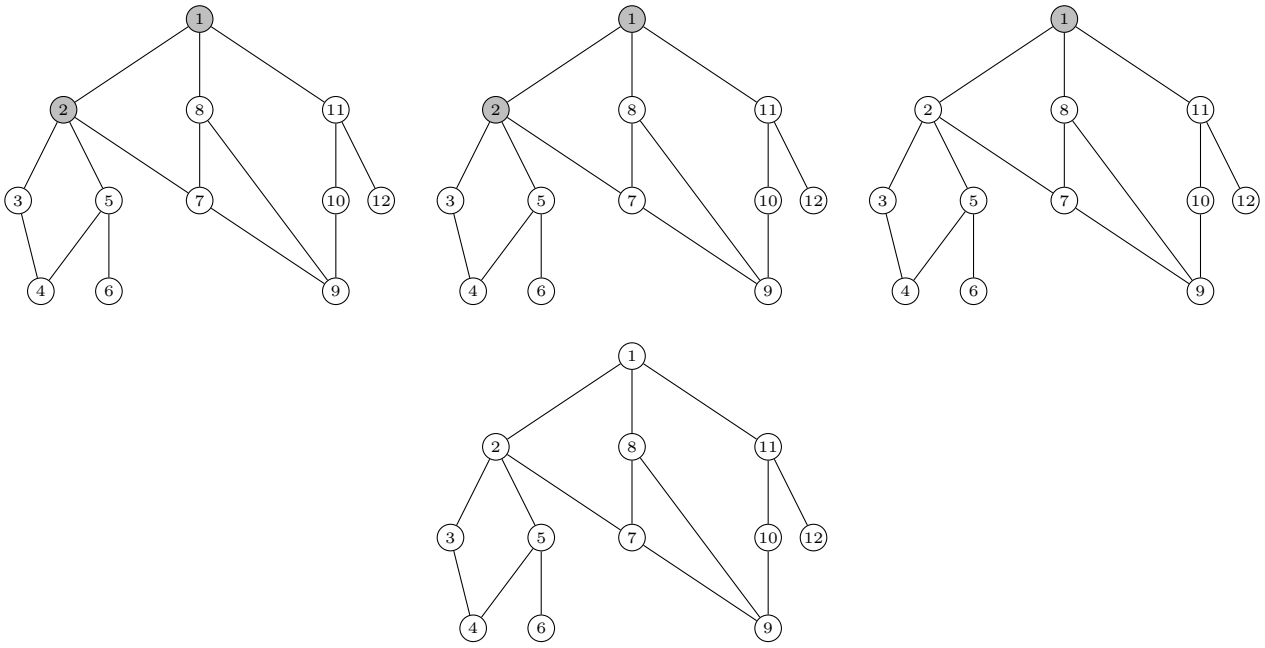
Comme pour le parcours en largeur, le premier sommet est purement arbitraire et nous prendrons dans le sommet le plus haut du graghe. Dans ce qui suit, les sommets noirs sont les sommets qui ne sont pas encore visités, les sommets gris sont les sommets visités qui possèdent peut-être encore des sommets adjacents non visités et les sommets blancs sont les sommets visités dont tous les sommets adjacents ont été visités.











Pour programmer le parcours en profondeur, on va utiliser le principe informatique de la pile. Le principe est le même que dans une pile d'assiette : lorsqu'on a lavé les assiettes, on les empile les unes au-dessus des autres, la dernière lavée étant placée tout en haut de la pile ; ensuite, lorsqu'on a va réutiliser les assiettes, on va les prendre de haut en bas, de sorte que la dernière assiette qui a été mise sur la pile et la première a être utilisée.

Le principe de programmation du parcours en profondeur va être le suivant :

1. on enpile le premier sommet (i.e. on le met en haut de la pile) ;

2. si le sommet `S` qui se trouve en haut de la pile possède un sommet adjacent `S'` qui n'a pas encore été visité, on empile `S'` par dessus `S` et, sinon, on dépile `S` (i.e. on le retire de la pile) ;

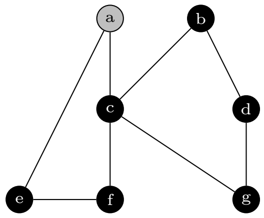
3. tant que la file n'est pas vide, on réitère le point **2.**.

Plus concrètement, on va partir d'un graphe représenté par un dictionnaire et on va utiliser trois variables similaires au parcours en largeur :

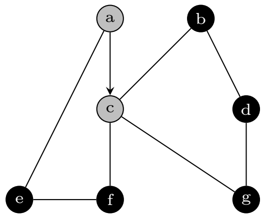
- une liste `Pile` représentant la pile telle que `Pile[-1]` (i.e. le dernier élément de la liste) est le dernier sommet visité et les éléments précédents de `Pile` sont les sommets qui ont mené à `Pile[-1]` ;
- un dictionnaire `Predecesseur` tel que, à chaque étape, les clés de `Predecesseur` sont les sommets `S` visités avec, comme valeur associée à `S`, le sommet à partir duquel on est arrivé à `S` i.e. le sommet qui était en haut de la pile lorsqu'on a ajouté `S` à la pile `Pile` ;
- un dictionnaire `Couleur` dont les clés sont les sommets du graphe et les valeurs associées sont des couleurs : `'noir'` si le sommet n'est pas encore passé dans la pile (il n'a pas encore été visité), `'gris'` si le sommet est dans la pile (il a été visité mais possède peut-être encore des sommets adjacents non visités) et `'blanc'` si le sommet est sortie de la pile (il a été visité et ne possède plus de sommets adjacents non visités).

On considère le même graphe que pour le parcours en largeur.

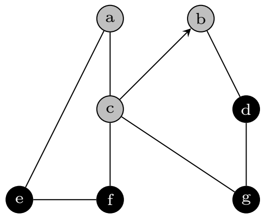
Question 17. Compléter le tableau en donnant les valeurs des variables `File`, `Predecesseur` et `Couleur` à chaque étape du parcours en profondeur de `G` commençant par le sommet a.

<table>
<tr><td> 
</td><td>Pile = ['a']

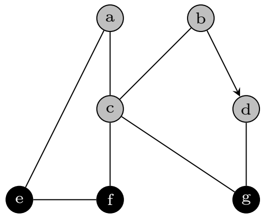
Predecesseur = { 'a':None }

Couleur = { 'a':'gris', 'b':'noir', 'c':'noir', 'd':'noir', 'e':'noir', 'f':'noir', 'g':'noir' }</td></tr>
<tr><td> 
</td><td>Pile = ['a', 'c']

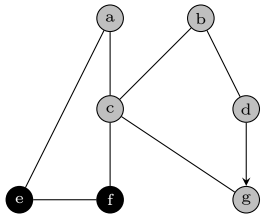
Predecesseur = { 'a':None, 'c':'a' }

Couleur = { 'a':'gris', 'b':'noir', 'c':'gris', 'd':'noir', 'e':'noir', 'f':'noir', 'g':'noir' }</td></tr>
<tr><td> 
</td><td>Pile = ['a', 'c', 'b']

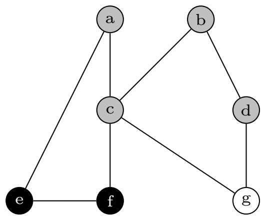
Predecesseur = { 'a':None, 'c':'a', 'e':'a', 'b':'c' }

Couleur = { 'a':'gris', 'b':'gris', 'c':'gris', 'd':'noir', 'e':'gris', 'f':'noir', 'g':'noir' }</td></tr>
<tr><td> 
</td><td>Pile =

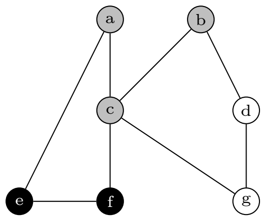
Predecesseur =

Couleur =</td></tr>
<tr><td> 
</td><td>Pile =

Predecesseur =

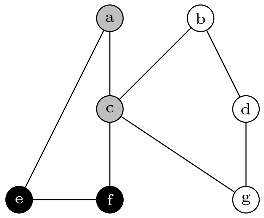
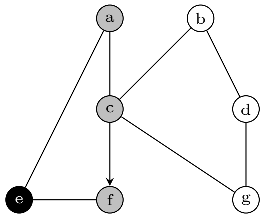
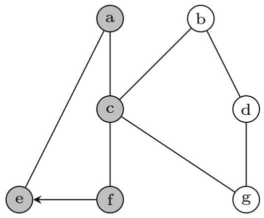
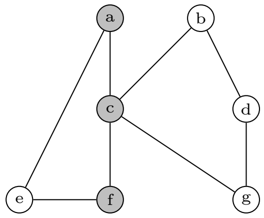
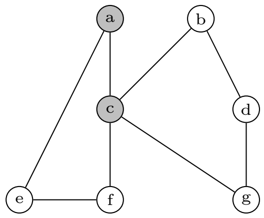
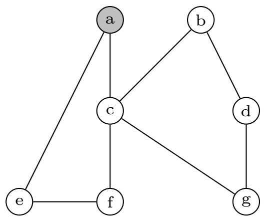
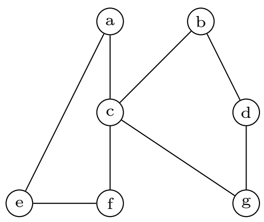
Couleur =</td></tr>
<tr><td> 
</td><td>Pile =

Predecesseur =

Couleur =</td></tr>
<tr><td> 
</td><td>Pile =

Predecesseur =

Couleur =</td></tr>
</table>

	Pile = Predecesseur = Couleur =
	Pile = Predecesseur = Couleur =
	Pile = Predecesseur = Couleur =
	Pile = Predecesseur = Couleur =
	Pile = Predecesseur = Couleur =
	Pile = Predecesseur = Couleur =
	Pile = Predecesseur = Couleur =

Question 18. Compléter le script de la fonction `parcours_profondeur` suivante qui prend en argument un graphe représenté par un dictionnaire `G` et un sommet `S` et qui détermine un parcours en profondeur du graphe en partant du sommet `S`.

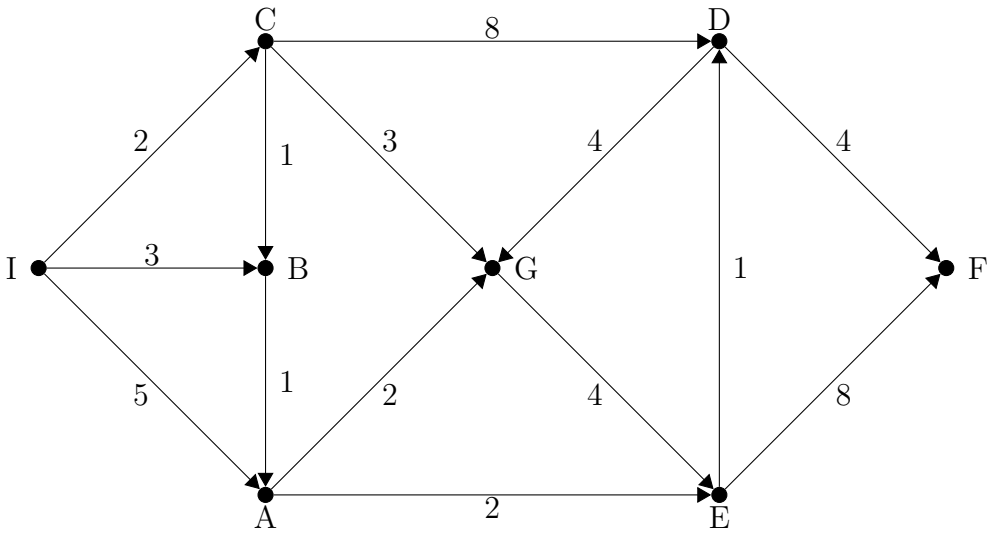
```
def parcours_profondeur(G,S):
    Pile = ...
    Couleur = { ... for x in G }
    Predecesseur, Couleur[S] = {S:...}, ...
    while ...:
        u = Pile[-1]
        R = [y for y in ... if Couleur[y] == ...]
        if ...:
            v = ...
            Predecesseur[v], Couleur[v] = ..., ...
            Pile ...
        else:
            Pile.pop()
            Couleur[u] = ...
    return Predecesseur
```

Question 19. Tester la fonction `parcours_largeur` sur le graphe de la question **13** et vérifier qu'on trouve le même parcours (à l'ordre près) que dans la question **14**.

IV. — Recherche d'un plus court chemin

Dans ce paragraphe, on va considérer des graphes pondérés. Dans un tel graphe, le poids d'un chemin est la somme des poids des arêtes qui constituent ce chemin. Étant donné un sommet initial I et un sommet final F, on va chercher, parmi tous les chemins dans le graphe qui conduisent de I à F, un chemin de poids minimum. Un tel chemin est appelé un **plus court chemin** de I à F. Ainsi, ce qu'on cherche à minimiser, ce n'est pas le nombre d'arêtes du chemin mais le poids total du chemin. La terminologie « plus court chemin » peut sembler un peu étrange mais si on pense que les poids sur les arêtes représentent des durées de trajet ou des nombres de kilomètre alors un plus court chemin représente bien un chemin le plus court (en temps ou en distance).

Considérons l'exemple suivant :



Ici, il y a différents chemins de I à F : par exemple, I – A – E – F dont le poids $5+2+8=15$ ou I – C – D – F dont le poids est $2+8+4=14$. Ainsi, le second chemin a un poids inférieur au premier. Il y a cependant un chemin de poids encore inférieur, il s'agit de I – B – A – E – D – F dont le poids est $3+1+2+1+4=11$. On va montrer qu'il s'agit du plus court chemin entre I et F.

Pour cela, on va utiliser l'algorithme de Dijkstra dont le principe est le suivant : si un plus court chemin $\mathcal{C}$ entre I et F passe par un sommet S alors la portion de $\mathcal{C}$ qui mène de I à S est un plus court chemin entre I et S.

Par exemple, pour le graphe ci-dessus, si un plus court chemin entre I et F passe par A alors il commence par I – B – A plutôt que par I – A car le poids du premier $(3+1=4)$ est inférieur au poids du second (5).

L'idée est donc de partir de I et de parcourir le graphe en choisissant à chaque étape un sommet S parmi ceux-qui n'ont pas encore été visité de telle façon que le chemin entre I et S ait le plus petit poids possible.

Pour programmer l'algorithme de Dijkstra, on va utiliser la matrice d'adjacence du graphe.

Question 20. Déterminer la matrice d'adjacence du graphe ci-dessus puis la représenter en Python par dictionnaire `Mat_adj`.

Ensuite, on va utiliser trois variables :

- une liste `Non_parcourus` représentant les sommets non encore visités ;
- un dictionnaire `Predecesseur` tel que, à chaque étape, les clés de `Predecesseur` sont les sommets du graphe et, pour chaque sommet `S`, la valeur de `S` est le prédécesseur de `S` dans un plus court chemin entre `I` et `S` ;
- un dictionnaire `Poids` tel que, à chaque étape, les clés de `Poids` sont les sommets du graphe et, pour chaque sommet `S`, la valeur de `S` est le poids d'un plus court chemin de `I` à `S`.

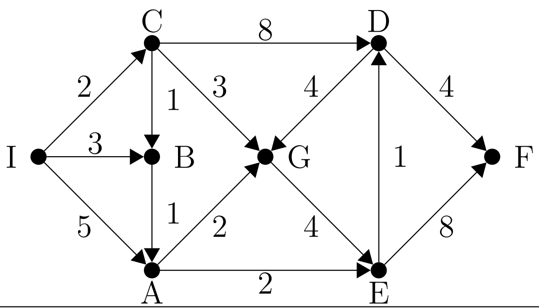
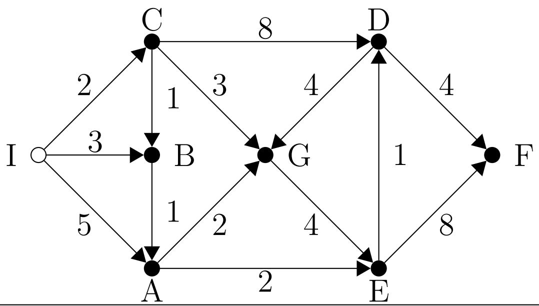
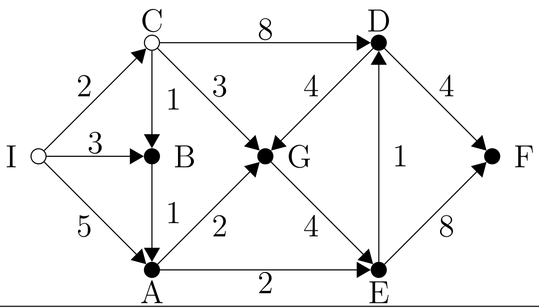
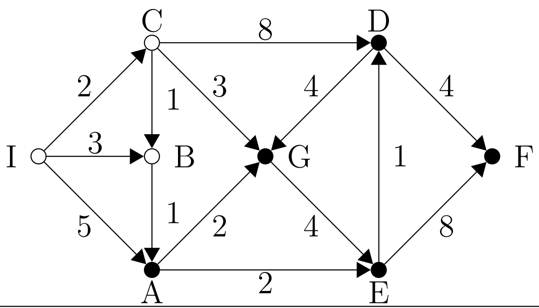
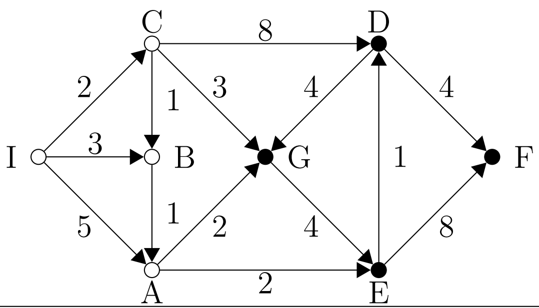
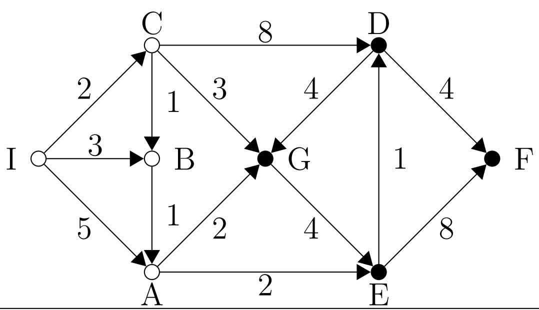
Initialement, la variable `Non_parcourus` contiendra tous les sommets du graphe, les valeurs de `Predecesseur` seront `'NC'` (pour Non Connu) sauf celle de `I` qui sera `None` et les valeurs de `Poids` seront `-1` sauf celle de `I` qui sera `0`.

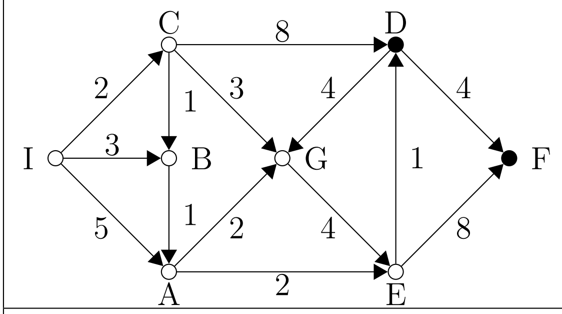
Ensuite, on fait évoluer les variables de la façon suivante : tant que la liste `Non_parcourus` n'est pas vide,

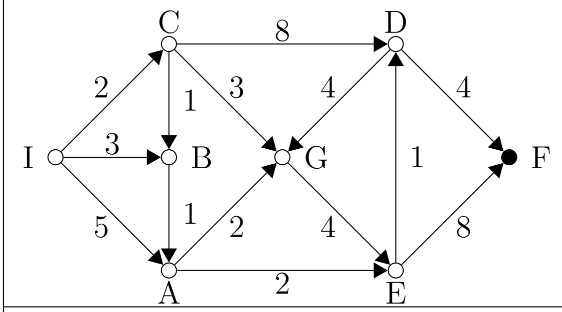
1. on choisit dans le dictionnaire `Poids` un sommet `S` ayant le poids $q \geqslant 0$ minimum parmi tous les sommets non parcourus ;
2. Pour tout sommet `T` adjacent à `S` et figurant dans la liste `Non_parcourus`, on note p le poids de l'arête entre `S` et `T` (i.e. `Mat_adj[S,T]`). Si $p+q$ est strictement inférieur au poids de `T`, on affecte la valeur $p+q$ à `Poids`[T] et on affecte la valeur `'S'` à `Predecesseur[T]`.
3. On supprime `S` de la liste des sommets non parcourus.

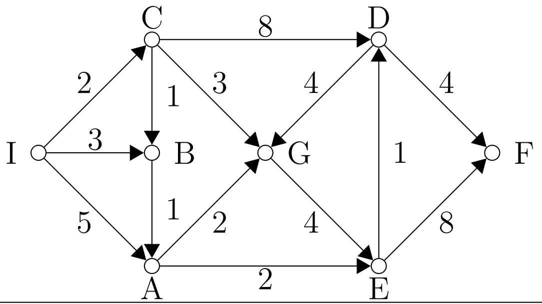
Question 21. Compléter le tableau en donnant les valeurs des variables `Non_parcourus`, `Poids` et `Predecesseur` à chaque étape de l'algorithme de Dijkstra.

Les sommets en blanc sont les sommets visités.

	Non_parcourus = ['I', 'A', 'B', 'C', 'D', 'E', 'G', 'F'] Predecesseur = { 'I':None, 'A':'NC', 'B':'NC', 'C':'NC', 'D':'NC', 'E':'NC', 'G':'NC', 'F':'NC' } Poids = { 'I':0, 'A':-1, 'B':-1, 'C':-1, 'D':-1, 'E':-1, 'G':-1, 'F':-1 }
	Non_parcourus = ['A', 'B', 'C', 'D', 'E', 'G', 'F'] Predecesseur = { 'I':None, 'A':'I', 'B':'I', 'C':'I', 'D':'NC', 'E':'NC', 'G':'NC', 'F':'NC' } Poids = { 'I':0, 'A':5, 'B':3, 'C':2, 'D':-1, 'E':-1, 'G':-1, 'F':-1 }
	Non_parcourus = ['A', 'B', 'D', 'E', 'G', 'F'] Predecesseur = { 'I':None, 'A':'I', 'B':'I', 'C':'I', 'D':'C', 'E':'NC', 'G':'C', 'F':'NC' } Poids = { 'I':0, 'A':5, 'B':3, 'C':2, 'D':10, 'E':-1, 'G':5, 'F':-1 }
	Non_parcourus = Predecesseur = Poids =
	Non_parcourus = Predecesseur = Poids =
	Non_parcourus = Predecesseur = Poids =

	`Non_parcourus =`
	
	`Predecesseur =`
	
	`Poids =`

	`Non_parcourus =`
	
	`Predecesseur =`
	
	`Poids =`

	`Non_parcourus =`
	
	`Predecesseur =`
	
	`Poids =`

Ainsi, le plus court chemin a bien un poids de 11 et pour retrouver ce chemin, on utilise le dictionnaire `Predecesseur` qui permet de « remonter » le chemin à partir de F :

$$\mathrm{F} \to \mathrm{D} \to \mathrm{E} \to \mathrm{A} \to \mathrm{B} \to \mathrm{I}$$

donc un plus court chemin est

$$\mathrm{I} - \mathrm{B} - \mathrm{A} - \mathrm{E} - \mathrm{D} - \mathrm{F}$$

Afin de programmer l'algorithme, nous allons avoir besoin de plusieurs fonctions intermédiaires.

Question 22. Écrire une fonction `Sommet_poids_min` qui prend en argument une liste de sommets `L` et un dictionnaire `P` dont les clés contiennent les sommets de `L` et les valeurs sont des nombres et qui renvoie un sommet de `L` ayant la valeur minimale parmi les valeurs positives ou nulles de `P`.

Question 23. Écrire une fonction `Chemin` qui prend en argument un dictionnaire `Predecesseur` et deux sommets `I` et F et qui renvoie, sous la forme d'une liste, les sommets du chemin entre `I` à `F`.

Question 24. Écrire une fonction `Suppr` qui prend en argument une liste `L` contenant des éléments distincts et un élément `e` de la liste `L` et qui supprime `e` dans `L`.

Indcation. On pourra commencer par chercher l'indice de `e` dans `L` puis utiliser la méthode de liste `.pop`.

Question 25. Compléter le script de la fonction `Plus_court_chemin` suivante qui prend en argument un graphe représenté par un dictionnaire `G` et deux sommets `Init` et `Final` et qui renvoie la liste des sommets d'un plus court chemin de `I` à `F` ainsi que le poids de ce chemin.

```
def Plus_court_chemin(G,Init,Final):
  Non_parcourus = [... for S in G]
  Predecesseur = { ...:'NC' for ...}
  Predecesseur[Init] = ...
  Poids = { ...:-1 for ...}
  Poids[Init] = ...
  while ...:
    S = Sommet_poids_min(..., ...)
    for T in Mat_adj[...]:
      if T in ...:
        q = ...
        p = ...
        if Poids[T] == ... or p + q < ...:
          Poids[T] = ...
          Predecesseur[T] = ...
    Suppr(..., ...)
  return Chemin(..., ..., ...), Poids[...]
```

Question 26. Tester la fonction `Plus_court_chemin` sur le graphe de la question **21** et vérifier qu'on trouve le même parcours que dans la question **22**.

Question 27. Dans le graphe suivant, déterminer le plus court chemin entre de A à N.