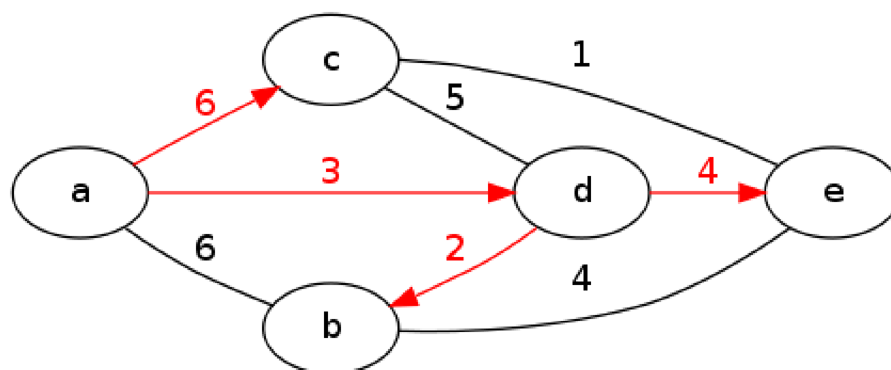
Chapitre 4

Algorithmes sur les graphes



4.1 Représentation d'un graphe

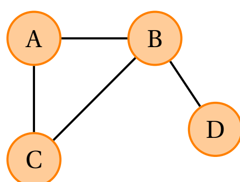
Découvrons tout d'abord comment représenter un graphe. Nous ferons la distinction entre graphe pondéré et non pondéré, mais tout ce qui suit peut s'appliquer que le graphe soit orienté ou non.

4.1.1 Graphe non pondéré

Sous forme de listes d'adjacence

Une manière assez simple pour représenter un graphe non pondéré est d'utiliser des listes d'adjacence, c'est à dire d'énumérer la liste des voisins ou des successeurs de chaque sommet. En Python, on utilisera un dictionnaire, parfaitement adapté.

Exemple. Voici un graphe non orienté ainsi que les listes de voisins associées :



Sommet	Voisins
A	B,C
B	A,C,D
C	A,B
D	B

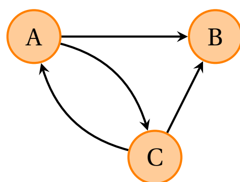
On peut alors représenter le graphe avec le dictionnaire suivant (représenté sur plusieurs lignes pour plus de clarté) :

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

Chaque clé du dictionnaire correspond à un sommet du graphe, et la valeur correspondante est la liste des voisins.

Dans le cas d'un graphe orienté, on code alors la liste des successeurs à la place de la liste des voisins.

Exemple.



Sommet	Successeurs
A	B,C
B	∅
C	A,B

Dans ce cas, la représentation du graphe est la suivante :

```
G = {
    'A' : ['B', 'C'],
    'B' : [],
    'C' : ['A', 'B']
}
```

Sous forme d'objet

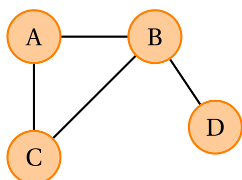
On peut également représenter le graphe sous forme d'objet en implémentant une classe `Graphe`. Le principe est le même que précédemment : on stocke les informations sur les sommets et les relations dans un dictionnaire, et on implémente des méthodes qui permettent de modifier l'objet, comme `ajouterSommet`, `ajouterArete` ...

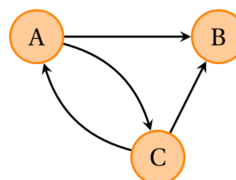
```python
class Graphe:
    """ Représentation d'un graphe sous forme d'objet """

    def __init__(self, oriente = False):
        self.oriente = oriente
        self.dictionnaire = {}

    def __repr__(self):
        """ Affichage du graphe """
        return str(self.dictionnaire)

    def ajouterSommet(self, S):
        """ Ajout du sommet S au graphe """
        if S not in self.dictionnaire: # Si S n'est pas déjà dans les clés du dictionnaire
            self.dictionnaire[S] = [] # On crée la clé avec une valeur initiale vide

    def ajouterArete(self, S1, S2):
        """ Ajout d'une arête (ou d'un arc si le graphe est orienté) """
        if S1 in self.dictionnaire and S2 in self.dictionnaire:
            if S2 not in self.dictionnaire[S1]: # Ajout de la relation S1 -> S2
                self.dictionnaire[S1].append(S2)
            if not self.oriente and S1 not in self.dictionnaire[S2]: # Ajout de la relation S2 -> S1
                self.dictionnaire[S2].append(S1)
```

Question 01 Expliquer chacune des conditions présentes dans la méthode `ajouterArete`.

Question 02 L'instruction `G = Graphe()` crée-t-elle un graphe orienté ou non ?

Question 03 Donner la suite d'instructions permettant de créer chacun des graphes suivants :





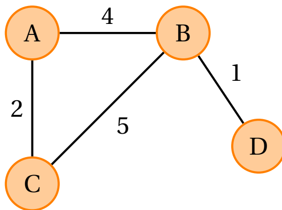
Question 04 Écrire une méthode `ordre(self)` qui retourne l'ordre du graphe.

Question 05 Écrire une méthode `nombreRelations(self)` qui retourne le nombre de relations du graphe, c'est à dire le nombre d'arêtes (si le graphe est non orienté) ou le nombre d'arcs (si le graphe est orienté).

4.1.2 Graphe pondéré

Représenter un graphe pondéré se fait de façon similaire : on utilise encore un dictionnaire, mais à chaque sommet n'est plus associé une liste mais un dictionnaire, car il faut enregistrer le poids de la relation.

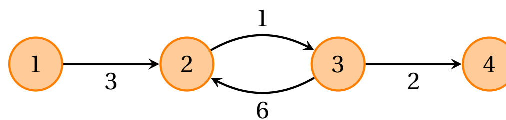
Exemple. Considérons le graphe pondéré suivant :



Les clés du dictionnaire `G` sont les sommets du graphe. Les valeurs associées sont des dictionnaires, dont les clés sont les sommets adjacents et les valeurs le poids de l'arête (ou de l'arc) associé.

```
G = {
    'A': {
        'B': 4,
        'C': 2
    },
    'B': {
        'A': 4,
        'C': 5,
        'D': 1
    },
    'C': {
        'A': 2,
        'B': 5
    },
    'D' : {
        'B': 1
    }
}
```

Question 06 Donner la représentation sous forme de dictionnaire du graphe orienté suivant :



Question 07 Implémenter une classe `GraphePondere` en adaptant la classe `Graphe` vue précédemment.

4.2 Parcours d'un graphe

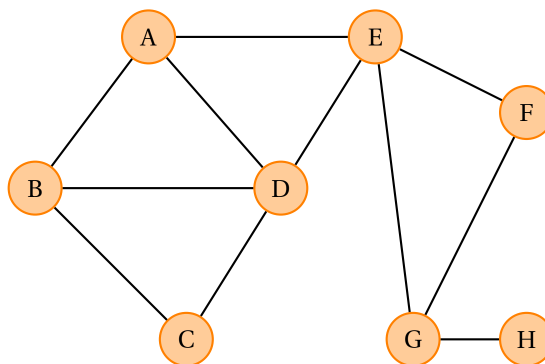
Parcourir un graphe non orienté consiste à lister tous les sommets qui le composent. Comme pour les arbres, il existe plusieurs types de parcours, les deux principaux étant le parcours en largeur et le parcours en profondeur.

4.2.1 Parcours en largeur

Dans un parcours en largeur, on se fixe un sommet initial (il n'y a pas de racine dans un graphe) et on parcourt les sommets par **distance croissante** du sommet de départ. L'idée est d'utiliser une file et une liste de la manière suivante :

```
F : File vide
L : Liste vide
S : Sommet

Enfiler le sommet de départ dans F
Ajouter le sommet de départ à L

Tant que F n'est pas vide:
    On défile F dans S
    Pour tout voisin V de S qui n'est pas dans L:
        On enfile V dans F
        On ajoute V à L

Retourner L
```

Question 08 Appliquer l'algorithme au graphe ci-contre, en partant du sommet A.



Question 09 À quoi sert la file `F` ? et la liste `L` ?

Question 10 Implémenter cet algorithme dans une fonction `parcours_largeur(G, SD)` où `G` est un graphe donné sous la forme d'un dictionnaire, et `SD` est le sommet de départ.

```
# G désigne le graphe ci-dessus

>>> parcours_largeur(G, 'A')
['A', 'B', 'D', 'E', 'C', 'F', 'G', 'H']
>>> parcours_largeur(G, 'E')
['E', 'A', 'D', 'F', 'G', 'B', 'C', 'H']
```

☞ *Dans l'exemple ci-dessus, les sommets sont parcourus dans l'ordre alphabétique, mais ce n'est pas une obligation*

On peut améliorer l'algorithme précédent en utilisant un dictionnaire contenant l'état « visité » ou « non visité » de chaque sommet. Cela permet d'éviter un trop grand nombre de recherches dans la liste `L` précédente.

```
F : File vide
L : Liste vide
D : Dictionnaire vide
S : Sommet

Enfiler le sommet de départ dans F
Ajouter le sommet de départ à L

Pour chaque sommet S du graphe:
    D[S] = Faux

Tant que F n'est pas vide:
    On défile F dans S
    Pour tout voisin V de S tel que D[S] est Faux:
        On enfile V dans F
        On ajoute V à L
        D[S] = Vrai

Retourner L
```

À titre complémentaire, on pourra traiter l'exercice suivant, présentant une certaine difficulté :

Exercice 01 Écrire une fonction `distances(G,S)` qui retourne la distance du sommet `S` à tous les sommets du graphe `G`.

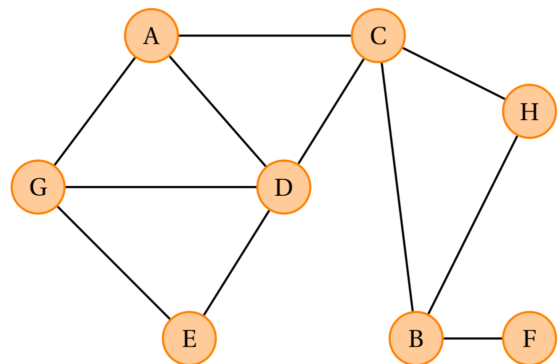
4.2.2 Parcours en profondeur

Comme pour le parcours en largeur, on commence par choisir un sommet de départ. Le parcours en profondeur ne liste pas les sommets par distance croissante du sommet de départ, mais cherche à aller « le plus loin possible » d'un sommet donné. On utilise cette fois-ci une pile et une liste :

```
P : Pile vide
L : Liste vide
S : Sommet

Empiler le sommet de départ dans P
Ajouter le sommet de départ à L

Tant que P n'est pas vide:
    On met le sommet de P dans S
    Si S a un voisin V qui n'est ni dans P, ni dans L:
        On empile V dans P
        On ajoute V à L
    Sinon:
        On dépile P

Retourner L
```

L'algorithme est similaire au parcours en largeur, mais l'utilisation d'une pile change le sens de parcours du graphe.

Question 11 Donner le parcours en profondeur du graphe ci-contre, en partant du sommet A puis du sommet E, en choisissant les voisins dans l'ordre alphabétique.



Question 12 Implémenter cet algorithme dans une fonction `parcours_profondeur(G, SD)` où `G` est un graphe donné sous la forme d'un dictionnaire, et `SD` est le sommet de départ.

```
# G désigne le graphe ci-dessus

>>> parcours_profondeur(G, 'A')
['A', 'C', 'B', 'F', 'H', 'D', 'E', 'G']
>>> parcours_profondeur(G, 'E')
['E', 'D', 'A', 'C', 'B', 'F', 'H', 'G']
```

4.3 Recherche de chemins

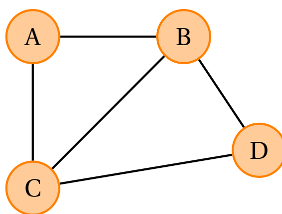
4.3.1 Recherche d'un chemin

Pour chercher un chemin reliant deux sommets d'un graphe, on dispose de l'algorithme suivant, qui fait encore intervenir une pile. La particularité de cet algorithme est que l'on empile non pas des sommets mais des couples (*sommet*, *chemin*).

```
P : Pile
S : sommet
Sd : sommet de départ
Sa : sommet d'arrivée
chemin : liste de sommets

Empiler dans P le couple (Sd, [Sd])
Tant que P n'est pas vide:
    S, chemin = tête de P
    liste_noeuds = liste des sommets adjacents à S qui ne sont pas dans chemin

    Pour N dans liste_noeuds:
        Si N = Sd:
            Afficher chemin + [N]
        Sinon:
            Empiler (N, chemin + [N])
```

Cet algorithme ne retourne pas un chemin, mais affiche tous les chemins reliant les sommets de départ et d'arrivée.

Question 13 Appliquer cet algorithme au graphe suivant afin de déterminer les chemins reliant A à D :



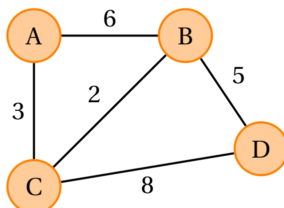
Question 14 Implémenter cet algorithme dans une fonction (ou une méthode) `rechercheChemin`.

Question 15 Modifier l'algorithme précédent afin qu'il retourne un chemin de longueur minimale (contenant le moins d'arêtes possible) entre le sommet de départ et le sommet d'arrivée.

4.3.2 Recherche du plus court chemin : algorithme de Dijkstra

Dans l'algorithme précédent, on définit le plus court chemin entre deux sommets comme un chemin possédant le moins d'arêtes possible. Dans le cas d'un graphe pondéré, la détermination du chemin le plus court est légèrement différente : on ne cherche pas un chemin possédant un nombre minimal d'arêtes, mais un chemin dont **la somme des poids est minimale**.

Question 16 Déterminer le plus court chemin entre les sommets A et D dans le graphe pondéré suivant :



Dans un cas simple comme ci-dessus, on peut lister tous les chemins menant de A à D et évaluer leur poids, ce qui prend quelques secondes à un humain et une fraction de seconde à une machine.

Mais de façon générale, ce problème est plus complexe qu'il n'y paraît. Il est impensable de lister tous les chemins lorsque le graphe possède un nombre important de relations. On peut s'imaginer une carte routière joignant des centaines de villes et villages par des milliers de routes. Le nombre de chemins entre 2 villes peut être très important, et la simple détermination de tous ces chemins peut demander un temps non négligeable, même à une puissante machine.

Pour répondre à ce genre de problématique, inutile de lister tous les chemins. On dispose d'un algorithme efficace, l'**algorithme de Dijkstra**, du nom de son découvreur *Edsger Dijkstra*, un mathématicien et informaticien néerlandais, en 1959.

Un exemple local

Un touriste débarque par bateau à Calvi et souhaite admirer les falaises de Bonifacio, mais il hésite sur le parcours à emprunter. Pour faire son choix, il dispose d'une carte de la Corse, ainsi que des informations sur la distance entre plusieurs grandes villes :

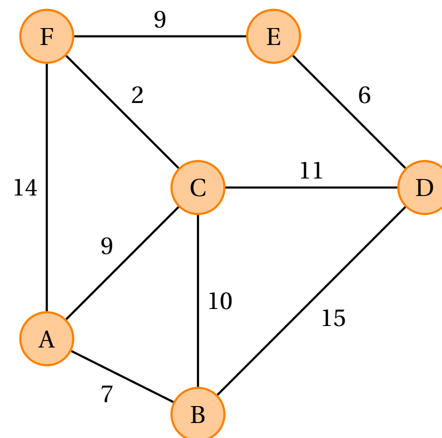
- Calvi - Ajaccio : 154km
- Calvi - Bastia : 92km
- Calvi - Corte : 86km
- Bastia - Corte : 67km
- Bastia - Porto-Vecchio : 143km
- Corte - Ajaccio : 80km
- Corte - Porto-Vecchio : 118km
- Ajaccio - Bonifacio : 131km
- P.-Vecchio - Bonifacio : 32km

Question 17 Représenter la situation par un graphe, en notant sur chaque arête la distance entre les deux villes correspondantes.

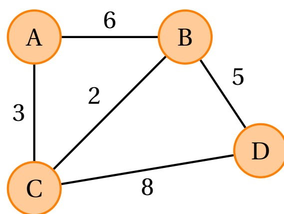
Question 18 Déterminer le chemin le plus court reliant Calvi à Bonifacio en utilisant l'algorithme de Dijkstra.

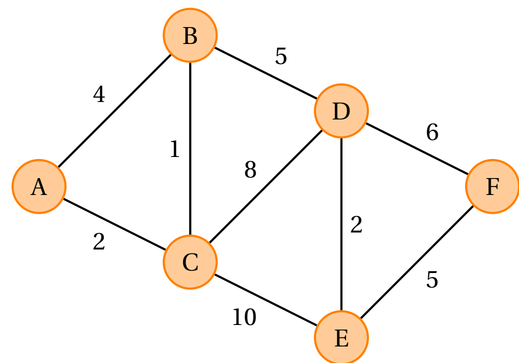
1. à l'aide d'un arbre
2. à l'aide d'un tableau

Question 19 Appliquer l'algorithme de Dijkstra afin de déterminer le plus court chemin entre les sommets A et E dans le graphe suivant :



Question 20 Appliquer l'algorithme de Dijkstra afin de déterminer le plus court chemin entre deux sommets quelconques des graphes suivants :





Question 21 ★ Programmer l'algorithme de Dijkstra !

Exercice 02 ★ Un exemple de problème faisant intervenir l'algorithme de Dijkstra :

`https://projecteuler.net/problem=83`