Cours & TD 4 : Piles, files et graphes

Objectifs :

- Présenter les algorithmes de manipulation des piles ;
- Comprendre le fonctionnement d'un algorithme récursif et l'utilisation de la mémoire lors de son exécution ;
- Présenter les éléments de base de l'algorithmique des graphes pour la recherche opérationnelle et les réseaux ;
- Représenter les graphes pondérés par des matrices d'adjacence ;
- Programmer l'algorithme de *Dijkstra* de recherche du plus court chemin dans un graphe pondéré à poids positifs.

Première partie

Piles et files

Calcul d'une somme L'objectif ici est de calculer $S = \sum_{i=1}^{n} i$.

Voici un premier algorithme.

```
def somme1(n):
    '''renvoie la somme des n premiers termes de manière récursive,
pour n entier non nul.'''
    if n == 1:  # cas de base
        return(1)
    else :
        return(n + somme1(n - 1))
    # le variant est n et l'invariant est
    # "n + (n-1) + ... + (n-k) + somme1(n-k-1) = sum(i=1 à n)i"
```

Pour `somme1(10)`, en déduire la taille minimale de la pile pour gérer ce calcul.

Edsger Dijkstra met en œuvre la notion de pile et la notion de récursivité quand il développe le langage de programmation *ALGOL60*.

Voici un nouvel algorithme.

```
def somme2(n, acc = 0):
    '''renvoie la somme des n premiers termes de manière récursive,
pour n entier non nul.'''
    if n == 0:  # cas de base
        return(acc)
    else :
        return(somme2(n - 1, acc + n))
    # le variant est n et l'invariant est
    # "acc + somme2(n - k, acc + (n - k + 1)) = sum(i=1 à n)i"
```

Pour `somme2(10)`, en déduire la taille minimale de la pile pour gérer ce calcul.

Deuxième partie

Bases de l'algorithme des graphes

Le premier traitement de la théorie des graphes date du XVIIIème siècle lorsque *Leonhard Euler* a résolu le problème des 7 ponts de Königsberg.

La théorie des graphes a toujours été un domaine essentiel de l'informatique théorique et elle a pris une importance considérable depuis la multiplication des réseaux de toutes sortes. Outre les réseaux numériques, les graphes servent aussi à modéliser les réseaux de transport (problèmes de plus court chemin, du voyageur de commerce ou du routage de données sur un réseau informatique), l'ordonnancement des tâches (programmation, productique, logistique, gestion de chantiers, recette de cuisine...).

1 Présentation succincte des graphes

Cette section prend appui sur le cours sur les graphes de *Jean-Michel Rey*.

1.1 Premières définitions

Un *graphe* est un couple (S, A) où S est un ensemble fini non vide de *sommets* (ou *nœuds*) et A une partie de $S \times S$ dont les éléments sont appelés *arêtes* (ou *arcs*). Si $A = S \times S$, le graphe est *complet*.

Un *arc* est plus précisément une paire ordonnée de sommets, i.e. une arête orientée : il n'y a qu'un sens de parcours pour un arc alors que pour une arête on peut aller indifféremment d'un sommet à l'autre. Dans le premier cas on parle de *graphe orienté*, et dans le second cas de *graphe non orienté*.

Dans les graphes que nous considérerons, on les supposera sans *boucles*, c'est à dire si $(i, j) \in A$ alors $i \neq j$ et on supposera que deux sommets sont reliés par au plus une arête (dans le cas contraire, on parle de *multigraphe*).

Un graphe est *dynamique* si on peut lui ajouter, supprimer ou modifier des sommets, des arêtes. Dans le cas contraire il est dit *statique*.

1.2 Autres définitions

Voisins	les voisins ou successeurs d'un sommet u sont les sommets v tels que l'arête (l'arc) $(u, v) \in A$. On dit aussi que v est *adjacent* à u.
Degré	le degré sortant d'un sommet u est le nombre de sommets v qui lui sont adjacents (i.e. $(u, v) \in A$), c'est à dire le nombre de ses successeurs. Le degré entrant de u est le nombre de sommets v auxquels u est adjacent (i.e.$(v, u) \in A$). On remarque que ces notions sont équivalentes dans le cas d'un graphe non orienté.
Chemin, distance	un chemin de u à v est une suite $x_0 = u, x_1, \ldots, x_N = v$ telle que pour tout $i \in [\![0; N-1]\!]$, $(x_i, x_{i+1}) \in A$. N est alors appelée la *longueur du chemin*. Dans ce cas, la *distance* entre u et v est la plus petite longueur d'un chemin de u à v.
Connexité	le graphe est fortement connexe si pour tout couple $(u, v) \in S^2$, $u \neq v$, il existe un chemin de u à v et un chemin de v à u. On parle de graphe connexe pour un graphe non orienté. Une composante fortement connexe est un sous-graphe (S', A') fortement connexe, maximal.
Pondéré	un graphe est pondéré s'il est muni d'une application $f : A \to \mathbb{R}$.

2 Implantation d'un graphe

Il existe deux représentations classiques d'un graphe $G = (S, A)$:

1. par liste d'adjacence ;
2. par matrice d'adjacence.

Dans les deux cas, les éléments de S sont désignés par des entiers.

2.1 liste d'adjacence

On associe à chaque sommet la liste de ses successeurs.

L'intérêt de cette implantation est que la place mémoire requise est linéaire en $Card(A) = |A|$ (nombre d'arêtes), mais le principal défaut est que l'on ne sait pas en temps constant s'il existe une arête entre deux sommets i et j, car il faut parcourir les listes.

Cette implantation n'est pas celle retenue dans le programme d'informatique pour tous.

2.2 matrice d'adjacence

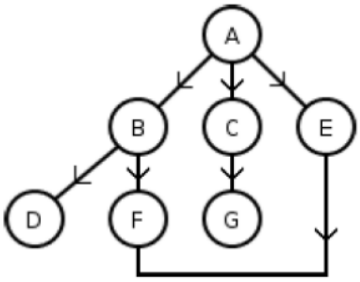
On définit la matrice d'adjacence $M = (a_{i,j})$ où $a_{i,j} = \begin{cases} 1 & \text{si } (i,j) \in A \\ 0 & \text{sinon} \end{cases}$.

Lorsque le graphe est non orienté, la matrice est *symétrique*. La place utilisée est de l'ordre de n^2.

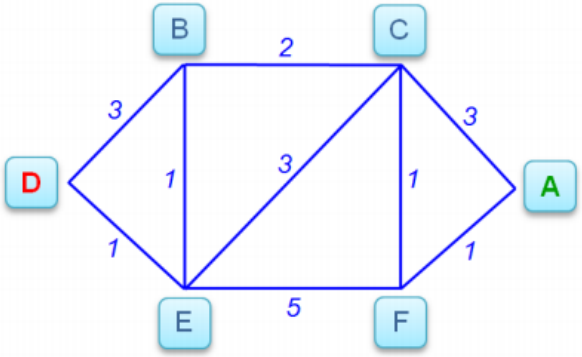
L'intérêt principal de cette implantation est que l'on sait en temps constant s'il existe une arête entre les sommets i et j. Le défaut majeur est que dans le cas d'un graphe peu dense ($|A|$ très inférieur à $|S|^2$), la place mémoire utilisée est toujours n^2.

Exercice 1 :
Pour chaque exemple ci-dessous, nommer le type de graphe qu'il décrit et donner sa matrice d'adjacence, que l'on notera MA_i ou `MA_i` dans les programmes.

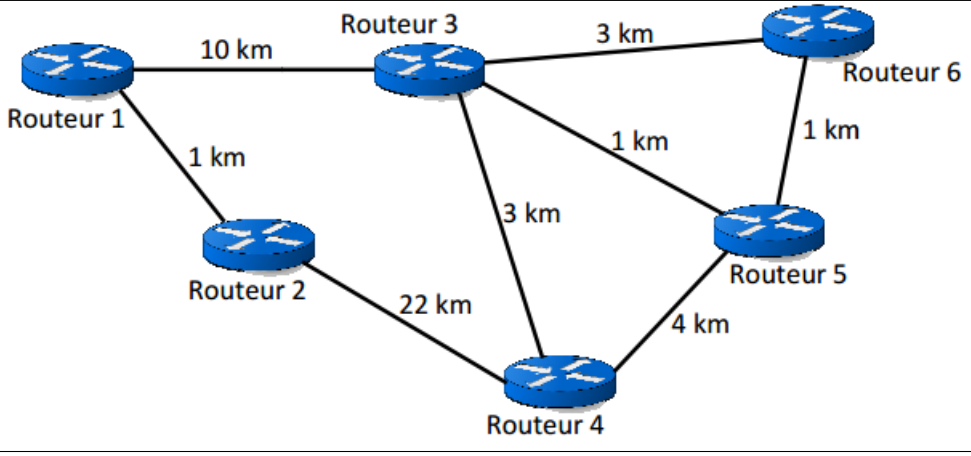
Exemple 1



Exemple 2

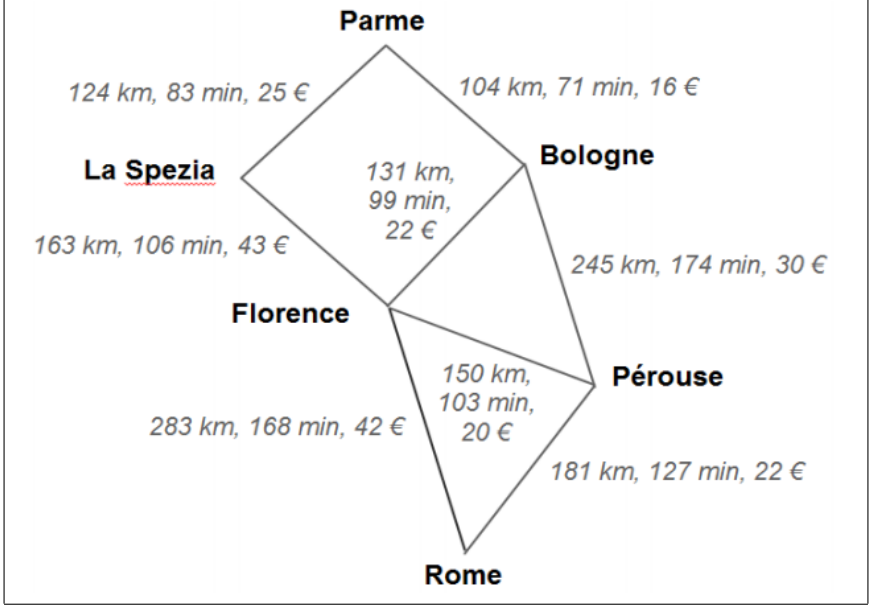


Exemple 3



Pour ce dernier exemple, qui est un multigraphe, on notera $MA_{4-distance}$, $MA_{4-duree}$ et MA_{4-cout} les matrices d'adjacence correspondantes aux différentes valeurs de pondération à entrer.

Exemple 4



Source (distances, durées et coûts) : *ViaMichelin.*

Programmation 1. *Écrire une fonction* `creation_graphe()` *qui permette de créer un graphe à partir de données entrées par l'opérateur. Elle doit renvoyer une liste contenant la liste des noms des sommets et la matrice d'adjacence de type* `array` *du module* `numpy`.

Utiliser votre fonction pour créer les graphes des exemples ci-dessous.

Programmation 2. *Écrire les fonctions suivantes :*

1. `voisins(Graphe, Sommet)`*, qui renvoie la liste des sommets voisins du sommet entré en argument pour un graphe.*
2. `adjacents(Graphe, Sommet)`*, qui renvoie la liste des sommets adjacents au sommet entré en argument pour un graphe.*
3. `degre_sortant(Graphe, Sommet)`*, qui renvoie le degré sortant du sommet entré en argument.*
4. `degre_entrant(Graphe, Sommet)`*, qui renvoie le degré entrant du sommet entré en argument.*
5. *On peut compléter cette liste par des fonctions permettant de modifier un arc ou un arête, d'ajouter ou de supprimer un sommet dans un graphe.*

Le graphe passé en argument est une liste comportant la liste des noms des sommets et la matrice d'adjacence de type `array` *du module* `numpy`.

3 Parcours d'un graphe

On suppose que le graphe est *connexe*, c'est à dire qu'à partir du sommet choisi pour le départ, tous les sommets du graphe sont accessibles : si le sommet de départ est u, pour tout sommet v il existe un chemin (orienté pour un graphe orienté) de u vers v.

3.1 Parcours en profondeur

Á partir du sommet de départ :

1. on suit le plus longtemps possible un chemin en visitant, à partir du sommet où l'on se trouve, le premier voisin **non encore exploré** ;
2. lorsqu'on ne peut plus, on revient en arrière explorer les possibilités laissées de côté, comme on le ferait dans un labyrinthe.

Exercice 2 :
Déterminer le parcours en profondeur de l'exemple 1 à partir du sommet A.

Pour mettre ce parcours en évidence, il faut gérer la liste des sommets visités, et la liste des prochains sommets à visiter. Cette seconde liste a une structure de *pile*.

Programmation 3. *Écrire la fonction* `parc_prof(Mat_adj, Depart)`. *Cette fonction renvoie le parcours en profondeur du graphe représenté par la matrice d'adjacence passée en argument à partir du sommet Depart.*

Tester son programme avec les 4 exemples précédents en utilisant les fonctions `Etiq` *ou* `Index` *du fichier* `graphes` *fourni. Ces fonctions permettent de convertir les noms des sommets en index et réciproquement, ce qui facilite la lecture des résultats.*

3.2 Parcours en largeur

Á partir d'un sommet d'origine *s*, on visite tous les voisins (successeurs) de *s* avant de visiter de la même manière les autres descendants.

Exercice 3 :
Déterminer le parcours en largeur de l'exemple 1 toujours à partir du sommet *A*.

Pour mettre ce parcours en évidence, il faut gérer la liste des sommets visités, et la liste des prochains sommets à visiter. Cette seconde liste a une structure de *file*.

Programmation 4. *Écrire la fonction* `parc_larg(Graphe, Depart)` *qui renvoie le parcours en largeur du graphe à partir du sommet Départ. Tester son programme avec les 4 exemples précédents.*

4 Algorithme de plus courts chemins

4.1 Présentation

Les problèmes de plus courts chemins que l'on peut étudier sont :

1. les plus courts chemins d'un sommet i à tous les autres ;
2. les plus courts chemins entre tous les sommets et un sommet j ;
3. les plus courts chemins entre tous les couples de sommets.

4.2 Algorithme de Dijkstra

> We use the fact that, if R is a node on the minimal path from P to Q, knowledge of the latter implies the knowledge of the minimal path from P to R. In the solution presented, the minimal paths from P to the other nodes are constructed in order of increasing length until Q is reached.

Edsger Wybe Dijkstra, « A note on two problems in connexion with graphs », 1959.

4.2.1 Principe de l'algorithme

Il s'agit ici de trouver tous les plus courts chemins d'origine fixe. Étant donné un sommet $origine$, trouver des plus courts chemins entre $origine$ et tous les autres sommets du graphe.

Exercice 4 :
Déterminer le plus court chemin de l'exemple 2 du sommet *D* au sommet *A*.

4.2.2 Implantation de l'algorithme

Programmation 5. *Écrire la fonction* `Dijkstra(Graphe, Depart, Arrivee)` *qui renvoie le parcours de plus court chemin du graphe du sommet Départ au sommet Arrivée. Tester son programme avec les 3 derniers exemples précédents.*