Travaux pratiques 10

Graphe (1) – Parcours

Informatique tronc commun MPSI et PCSI

I Implémentation de base

I.1 Vocabulaire des graphes

Un graphe $G = (S, A)$ est défini par un ensemble de **sommets** et un ensemble d'arête (voir figures 1, 2 et 3). On rappelle que :

- le graphe est pondéré si les arêtes sont munies de poids.
- le graphe est non orienté si le chemin entre deux sommets voisins ne dépend pas du sens de parcours. Dans le cas contraire, les arêtes sont appelées arcs et indiqués par une flèche.
- le graphe est connexe s'il est d'un seul tenant. Pour un graphe non orienté et connexe, deux sommets sont toujours joignables par un chemin.

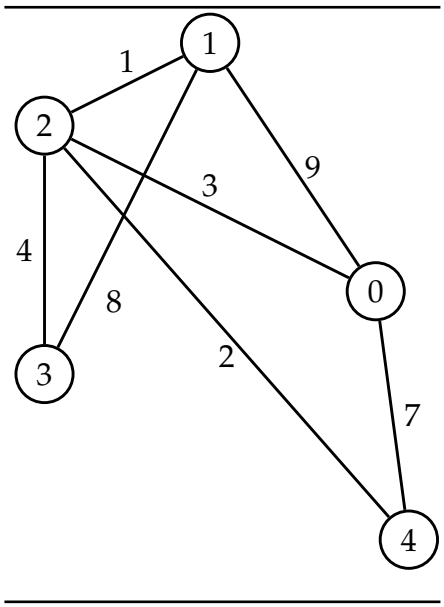


FIGURE 1 – Exemple de graphe pondéré, non orienté et connexe.

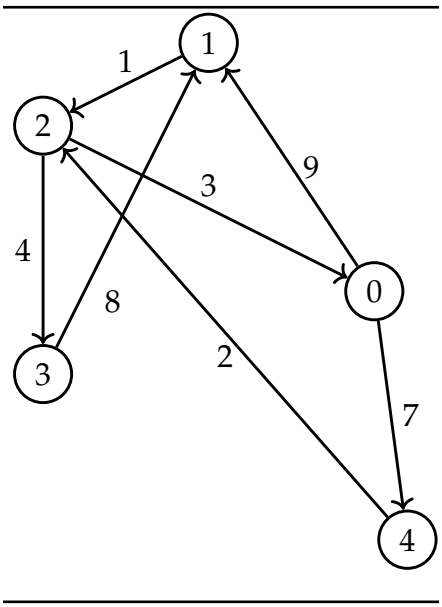


FIGURE 2 – Exemple de graphe pondéré, orienté et connexe.

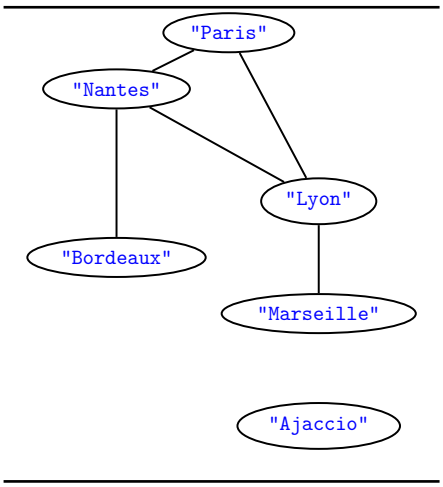


FIGURE 3 – Exemple de graphe non orienté, non pondéré et non connexe. De plus, les sommets sont ici représenté par des chaînes de caractère.

Remarque :

Dans la suite, et par souci de simplicité, nous considérerons des graphes non orientés, pondérés et dont les sommets sont représenté par des entiers.

I.2 Liste, dictionnaire et matrice d'adjacence

On considère l'exemple de la figure 1. On peut alors définir la liste d'adjacence par :

```
GL0 = [[[1, 9], [2, 3], [4, 7]],
       [[0, 9], [2, 1], [3, 8]],
       [[0, 3], [1, 1], [3, 4],
       [4, 2]], [[1, 8], [2, 4]],
       [[0, 7], [2, 2]]]
```

puis le dictionnaire d'adjacence par :

```
GD0 = {0: [[1, 9], [2, 3], [4, 7]],
      1: [[0, 9], [2, 1], [3, 8]],
      2: [[0, 3], [1, 1], [3, 4], [4, 2]],
      3: [[1, 8], [2, 4]],
      4: [[0, 7], [2, 2]]}
```

puis la matrice d'adjacence par :

$$\text{GM0} = \begin{pmatrix} -1 & 9 & 3 & -1 & 7 \\ 9 & -1 & 1 & 8 & -1 \\ 3 & 1 & -1 & 4 & 2 \\ -1 & 8 & 4 & -1 & -1 \\ 7 & -1 & 2 & -1 & -1 \end{pmatrix}$$

```
GM0 = [[-1, 9, 3, -1, 7],
     [9, -1, 1, 8, -1],
     [3, 1, -1, 4, 2],
     [-1, 8, 4, -1, -1],
     [7, -1, 2, -1, -1]]
```

Remarque :

Le -1 de la matrice d'adjacence signifie que deux sommets ne sont pas reliés par une arête. Autrement dit, c'est comme s'ils étaient séparés d'une « distance infinie ». De ce fait, on trouve aussi des matrices d'adjacence écrites avec un infini (voir à droite). Le module `math` fournit le flottant `inf` qui permet d'utiliser cette convention pour la matrice.

$$\text{GM0} = \begin{pmatrix} \infty & 9 & 3 & \infty & 7 \\ 9 & \infty & 1 & 8 & \infty \\ 3 & 1 & \infty & 4 & 2 \\ \infty & 8 & 4 & \infty & \infty \\ 7 & \infty & 2 & \infty & \infty \end{pmatrix}$$

Remarque :

Dans la suite, nous utiliserons la convention -1 pour deux sommets non voisins.

I.3 Quelque exemples simples

Exercice 1 - Graphe 1

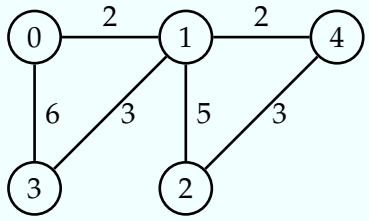


FIGURE 4 –

On considère le graphe de la figure 4.

1. Écrire en python la matrice d'adjacence `GM1` associé à ce graphe.
2. Écrire en python la liste d'adjacence `GL1` associé à ce graphe.
3. Écrire en python le dictionnaire d'adjacence `GD1` associé à ce graphe.

Exercice 2 - Graphe 2

On considère la matrice d'adjacence GM2 ci-contre.

1. Représenter le graphe associé.
2. Donner la liste d'adjacence `GL2`.
3. Donner le dictionnaire d'adjacence `GD2`.

$$\text{GM2} = \begin{pmatrix} -1 & -1 & 3 & 5 & -1 \\ -1 & -1 & -1 & 3 & 2 \\ 3 & -1 & -1 & 5 & -1 \\ 5 & 3 & 5 & -1 & 4 \\ -1 & 2 & -1 & 4 & -1 \end{pmatrix}$$

I.4 Calcul sur un graphe

Exercice 3 - Liste des voisins d'un sommet

Deux sommets sont voisins s'ils sont reliés par une arête. Écrire une fonction `Voisin(GM, sommet)` qui prend en argument une matrice d'adjacence et un entier `sommet` et qui renvoie la liste des voisins. Si un sommet n'a pas de voisin, la fonction renverra une liste vide.
Par exemple, `voisins(GM1, 1)` renverra `[0, 2, 3, 4]`.

Exercice 4 - Degré d'un sommet

Pour un graphe non orienté, le degré d'un sommet est le nombre de voisins, c'est-à-dire le nombre d'arêtes qui joignent ce sommet.
Écrire une fonction `degre(GM, sommet)` qui prend en argument une matrice d'adjacence et un entier `sommet` et qui renvoie le degré d'un sommet. Si un sommet n'a pas de voisin, la fonction renverra 0.
Par exemple, `degre(GM1, 1)` renverra `4`.

I.5 Conversion

On souhaite pouvoir passer d'un mode de représentation à un autre.

Exercice 5 - conversion 1

1. Écrire une fonction `mat_to_list` qui prend en argument une matrice d'adjacence et qui renvoie la liste d'adjacence associée.
   Par exemple, `mat_to_list(GM1)==GL1` correspondra à `True`.
2. Écrire une fonction `list_to_dict` qui prend en argument une liste d'adjacence et qui renvoie le dictionnaire d'adjacence associée.
   Par exemple, `list_to_dict(GL1)==GD1` correspondra à `True`.
3. Écrire une fonction `dict_to_mat` qui prend en argument un dictionnaire d'adjacence et qui renvoie la matrice d'adjacence associée.
   Par exemple, `list_to_dict(GL1)==GD1` correspondra à `True`.

Exercice 6 - conversion 2

En réutilisant les fonctions précédentes, écrire une fonction :

1. `list_to_mat` qui prend en argument une liste d'adjacence et qui renvoie la matrice d'adjacence associée.
2. `dict_to_list` qui prend en argument une dictionnaire d'adjacence et qui renvoie le liste d'adjacence associée.
3. `mat_to_dict` qui prend en argument un matrice d'adjacence et qui renvoie la dictionnaire d'adjacence associée.

II Parcours sur un graphe

II.1 Parcours en profondeur

On souhaite parcourir un graphe depuis un sommet appelé `depart` jusqu'à un autre sommet appelé `arrivee`. On note n le nombre total de sommet du graphe. On cherche l'existence d'un chemin entre `depart` et `arrivee`. On renverra

`True` s'il existe un chemin entre ces deux sommets et `False` dans le cas contraire. Au fur et à mesure du parcours, on utilise deux listes :

- `accessible` qui est une liste de taille n initialement composée du booléen `True`.
- `a_visiter` dans laquelle on stockera les indices des sommets à visiter. Au début, elle ne contient que `depart`.

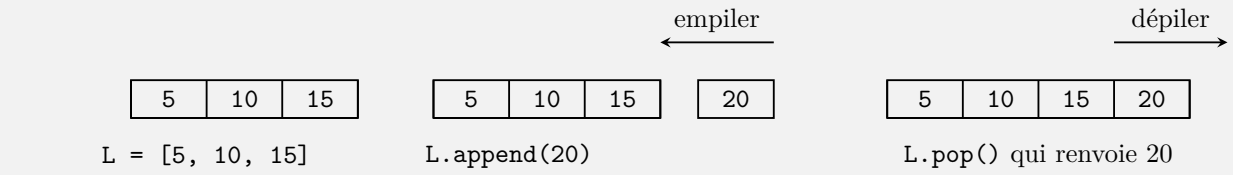
Parcours en profondeur :

Tant que la liste `a_visiter` n'est pas vide :

- on en sort l'élément le plus à droite de `a_visiter` :
  - si c'est l'arrivée, la fonction renvoie `True`. On a trouvé un chemin joignant `depart` et `arrivee`.
  - sinon, si ce sommet n'a pas encore été exploré, on le marque `False` dans la liste `accessible` et on l'ajoute à la droite de la liste `a_visiter` tous les voisins de ce sommet.
- si la liste `a_visiter` est vide, on renvoie `False` : il n'y a pas de chemin reliant `depart` et `arrivee`.

Remarque :

La liste `a_visiter` modélise ici une `pile`. On rappelle qu'une pile est une structure de donnée basée sur le principe : « dernier arrivé, premier sorti » [a].



a. LIFO : last in, first out.

Exercice 7 - Traverser la France en profondeur

On se place dans le cas de l'exemple de la figure 3 et on débute le parcours depuis Paris.

1. Que renvoie l'algorithme de parcours en profondeur si on souhaite rejoindre Ajaccio ?
2. Quelles sont les villes successivement testées si on souhaite rejoindre Bordeaux ?
3. Justifier alors l'appellation « parcours en profondeur ».

Exercice 8 - algorithme de parcours en profondeur

1. Écrire une fonction `test_chemin_profondeur(GL, depart, arrivee)` qui prend en argument une liste d'adjacence et deux entiers `depart` et `arrivee` et qui renvoie un booléen suivant qu'il existe ou non un chemin joignant ces deux sommets.
2. Tester cette fonction.
3. Que peut-on dire de la complexité de cet algorithme ?

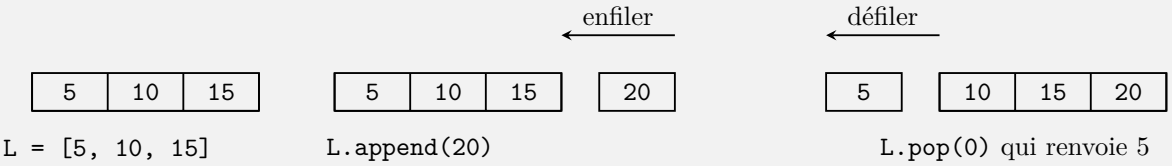
Exercice 9 - Chemin en profondeur

Écrire une fonction `chemin_profondeur(GL, depart, arrivee)` qui est une variante de la fonction précédente et qui renvoie le parcours entre `depart` et `arrivee`. S'il n'y a aucun chemin, la fonction renverra la liste vide. Dans le cas d'un chemin, la fonction renverra la liste des sommets entre le départ et l'arrivée.
Dans l'exemple de la figure 2 et pour aller de Paris à Bordeaux, on explore successivement Paris, Lyon, Nantes puis Bordeaux.

II.2 Parcours en largeur

Parcours en largeur :

On modifie l'algorithme précédent pour faire un parcours en largeur. Pour cela, la liste `a_visiter` doit se comporter comme une **file** : on ajoute les éléments par la droite mais on en sort d'abord les éléments par la gauche.
On rappelle qu'une file est une structure de donnée basée sur le principe : « premier arrivé, premier sorti » [a].



a. *FIFO : first in, first out.*

Remarque :

Bien que l'on puisse « simuler » la file par une liste, le programme officiel indique : « *On introduit à cette occasion les piles et les files ; on souligne les problèmes d'efficacité posés par l'implémentation des files par les listes de Python et l'avantage d'utiliser un module dédié tel que* `collections.deque` . »
Le *deque* [a] est une généralisation des files et des piles.

```
from collections import deque

D = deque([])          # création d'un deque vide
D.append(10)           # ajouter 10 par la droite D = [10]
D.append(15)           # ajouter 15 par la droite D = [10, 15]
D.appendleft(5)        # ajouter 5 par la gauche D = [5, 10, 15]
D.pop()                # dépiler par la droite D = [5, 10]
D.popleft()            # défiler par la gauche D = [10]
```

a. pour *double-ended queue*

Exercice 10 - Traverser la France en largeur
On se place dans le cas de l'exemple de la figure 3 et on débute le parcours depuis Paris.

1. Quelles sont les villes successivement testées si on souhaite rejoindre Bordeaux ?
2. Justifier alors l'appellation « parcours en profondeur ».

Exercice 11 - Chemin en largeur

1. Écrire une fonction `test_chemin_largeur(GL, depart, arrivee)` qui prend en argument une liste d'adjacence et deux entiers `depart` et `arrivee` et qui renvoie un booléen suivant qu'il existe ou non un chemin joignant ces deux sommets.
2. Écrire une fonction `chemin_largeur(GL, depart, arrivee)` qui est une variante de la fonction précédente et qui renvoie le parcours entre `depart` et `arrivee`. S'il n'y a aucun chemin, la fonction renverra la liste vide. Dans le cas d'un chemin, la fonction renverra la liste des sommets entre le départ et l'arrivée.

III Génération aléatoire de graphes

Dans cette section, on utilise les deux méthodes `random` et `randint` du module `random`.

Exercice 12 - Matrice d'adjacence aléatoire

1. Écrire une fonction `mat_adj_alea(n, A, p)` qui prend en argument deux entiers `n` et `A` et un flottant `p` et qui renvoie une matrice d'adjacence d'un graphe pseudo-aléatoire tel que chaque sommet ait une probabilité `p` d'avoir une arête avec chaque autre sommet et de poids tiré aléatoirement entre dans $[\![1, A]\!]$.
2. Tester la fonction. On vérifiera notamment que la matrice est symétrique et qu'elle n'est constituée que de -1 pour $p = 0$.

Exercice 13 - Liste d'adjacence aléatoire

1. Écrire une fonction `list_adj_alea(n, A, p)` qui prend en argument deux entiers `n` et `A` et un flottant `p` et qui renvoie une liste d'adjacence d'un graphe pseudo-aléatoire tel que chaque sommet ait une probabilité `p` d'avoir une arête avec chaque autre sommet et de poids tiré aléatoirement entre dans $[\![1, A]\!]$.
2. Tester la fonction. On vérifiera notamment que la matrice est symétrique et qu'elle n'est constituée que de -1 pour $p = 0$.

Exercice 14 - Test et statistiques

On prend dans la suite $n = 20$ sommets et $p = 0.5$. Proposer une simulation qui permette d'évaluer la probabilité $\mathbb{P}$ pour que deux sommets pris au hasard dans le graphe soient joignables par un chemin.

IV Solutions

Solution de l'exercice 1 - Graphe 1

```
GL1 =  [[[1,2], [3,6]],
        [[0,2], [2,5], [3,3], [4,2]],
        [[1,5], [4,3]],
        [[0,6], [1,3]],
        [[1,2], [2,3]]]
GM1 =  [[-1, 2, -1, 6, -1],
        [2, -1, 5, 3, 2],
        [-1, 5, -1, -1, 3],
        [6, 3, -1, -1, -1],
        [-1, 2, 3, -1, -1]]
GD1 = {0: [[1, 2], [3, 6]],
       1: [[0, 2], [2, 5], [3, 3], [4, 2]],
       2: [[1, 5], [4, 3]],
       3: [[0, 6], [1, 3]],
       4: [[1, 2], [2, 3]]}
```

Solution de l'exercice 2 - Graphe 2

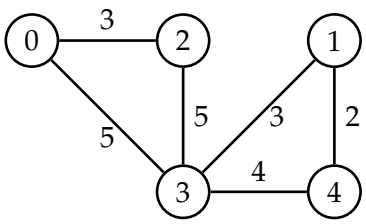


FIGURE 6 –

On donne en figure 6 le graphe associé à cette matrice d'adjacence.

```
GM2 = [[-1,-1,3, 5,-1],
       [-1,-1,-1,3,2],
       [3,-1,-1,5,-1],
       [5,3,5,-1,4],
       [-1,2, -1,4,-1]]
GL2 =  [[[2, 3], [3, 5]],
        [[3, 3], [4, 2]],
        [[0, 3], [3, 5]],
        [[0, 5], [1, 3], [2, 5], [4,
           4]],
        [[1, 2], [3, 4]]]

GD2 = {0: [[2, 3], [3, 5]],
       1: [[3, 3], [4, 2]],
       2: [[0, 3], [3, 5]],
       3: [[0, 5], [1, 3], [2, 5], [4,
           4]],
       4: [[1, 2], [3, 4]]}
```

Solution de l'exercice 3 - Liste des voisins d'un sommet

```
def voisins(M, sommet):
    n = len(M)
    L = [] # liste des voisins
    for i in range(n):
        if M[i][sommet] != -1:
            L.append(i)
    return L
```

Solution de l'exercice 4 - Degré d'un sommet

```
def degre(M, sommet):
    n = len(M)
    r = 0
    for i in range(n):
        if M[i][sommet]!=-1:
            r += 1
```

```
    return r
```

Solution de l'exercice 5 - conversion 1

```
def mat_to_list(mat):
    L = []
    n = len(mat)
    for i in range(n):
        L2 = []
        for j in range(n):
            if mat[i][j] != -1:
                L2.append([j, mat[i][j]])
        L.append(L2)
    return L
def list_to_dict(GL):
    n = len(GL)
    D = {}
    for i in range(n):
        D[i] = GL[i]
    return D
def dict_to_mat(dico):
    n = len(dico)
    m = [[-1 for j in range(n)] for j in range(n)]
    for i in range(n):
        for j in range(len(dico[i])):
            m[i][dico[i][j][0]] = dico[i][j][1]
    return m
```

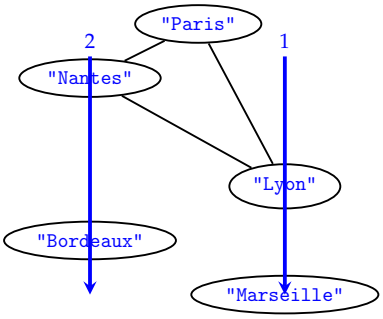
Solution de l'exercice 6 - conversion 2

```
def list_to_mat(GL):
    return dict_to_mat(list_to_dict(GL))

def dict_to_list(dico):
    return mat_to_list(dict_to_mat(dico))

def mat_to_dict(mat):
    return list_to_dict(mat_to_list(mat))
```

Solution de l'exercice 7 - Traverser la France en profondeur

1. Il n'y a pas de chemin possible entre Paris et Ajaccio donc l'algorithme doit renvoyer `False`. Cette réponse n'est possible que dans le cas d'algorithme non connexe.
2. On part de Paris. La pile `a_visiter` est alors `["Paris"]`.
   - On enlève Paris puis on ajoute Nantes et Lyon. La pile `a_visiter` est alors `["Nantes", "Lyon"]`.
   - On enlève Lyon puis on ajoute Nantes[1] et Marseille. La pile `a_visiter` est alors `["Nantes", "Nantes", "Marseille"]`.
   - On enlève Marseille. La pile `a_visiter` est alors `["Nantes", "Nantes"]`.
   - On enlève Nantes et on ajoute Bordeaux. La pile `a_visiter` est alors `["Nantes", "Bordeaux"]`.
   - On enlève Bordeaux et on s'arrête car c'est la ville d'arrivée.
3. En observant la figure ci-dessous, on comprend la notion de parcours en profondeur. Si on fait le parallèle avec un arbre, et partant de la racine (Paris), Lyon et Nantes sont deux branches et Marseille et Bordeaux sont des feuilles, l'algorithme va parcourir la feuille d'une branche avant de parcourir la branche parallèle.

1. qui n'a pas encore été visité !



Solution de l'exercice 8 - algorithme de parcours en profondeur

1.
```
def test_chemin_profondeur(GL, depart, arrivee):
    n = len(GL)
    accessible = [True for _ in range(n)]
    a_visiter = [depart]
    while len(a_visiter)!=0:
        x = a_visiter.pop()
        if x==arrivee:
            return True
        accessible[x] = False
        for j in range(len(GL[x])): # on ajoute les voisins
            if accessible[GL[x][j][0]]==True:
                a_visiter.append(GL[x][j][0])
    return False

GM3 = [[-1,1,-1], [1,-1,-1], [-1,-1,-1]]
GL3 = [[[1, 1]], [[0, 1]], []]
print(test_chemin_profondeur(GL3, 0, 2)) # renvoie False car non connexe
```

3. La complexité dépend fortement du type de graphe. Cependant, on peut dire que chaque sommet n'est visité qu'une fois et on a n sommets. Pour chaque sommet visité, la boucle `for` ajoute les voisins qui augmentent avec le nombre d'arêtes a. Une arête n'est prise en compte qu'une seule fois. Ainsi, la liste `a_visiter` aura, dans le pire des cas, une taille majorée par le nombre de sommets et le nombre d'arête.

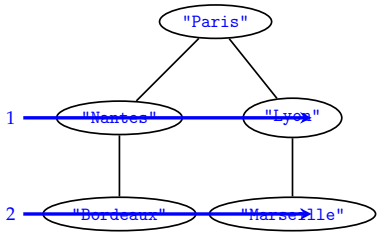
Solution de l'exercice 9 - Chemin en profondeur

```
def peres(GL, depart, arrivee):
    n = len(GL)
    accessible = [True for _ in range(n)]
    a_visiter = [depart]
    Lperes = [-1 for _ in range(n)]
    while len(a_visiter)!=0:
        x = a_visiter.pop()
        if x==arrivee:
            return Lperes
        accessible[x] = False
        for j in range(len(GL[x])):
            if accessible[GL[x][j][0]]==True:
                a_visiter.append(GL[x][j][0])
                print(GL[x][j][0])
                Lperes[GL[x][j][0]] = x
    return Lperes

def chemin_profondeur(GL, depart, arrivee):
    Lperes = peres(GL, depart, arrivee)
    if len(Lperes)==0:
        return []
    else:
        res = [arrivee]
```

```
        x = arrivee
        while x!=depart:
            y = Lperes[x]
            res.append(y)
            x = y
    return res[::-1]
L = ["Paris", "Nantes", "Lyon", "Bordeaux", "Marseille", "Ajaccio"]
GM4 = [[-1,1,1,-1, -1,-1], [1,-1,1,1,-1,-1],[1,1,-1,-1,1,-1],[-1,-1,1,-1,
   -1,-1], [-1,-1,-1,-1,-1,-1]]
GL4 = [[[1, 1], [2, 1]], [[0, 1], [2, 1], [3, 1]], [[0, 1], [1, 1], [4, 1]],
   [[2, 1]], []]
print(chemin_profondeur(GL4, 0, 3))
```

Solution de l'exercice 10 - Traverser la France en largeur

1. On part de Paris. La pile `a_visiter` est alors `["Paris"]`.
   - On enlève Paris et ajoute Nantes et Lyon. La file `a_visiter` est alors `["Nantes", "Lyon"]`.
   - On enlève Nantes et on ajoute Bordeaux. La file `a_visiter` est alors `["Lyon", "Bordeaux"]`.
   - On enlève Lyon et on ajoute Marseille. La file `a_visiter` est alors `["Bordeaux", "Marseille"]`.
   - On enlève Bordeaux et on s'arrête car c'est la ville d'arrivée.
2. En observant la figure ci-dessous, on comprend la notion de parcours en largeur. Si on fait le parallèle avec un arbre, et partant de la racine (Paris), Lyon et Nantes sont deux branches et Marseille et Bordeaux sont des feuilles, l'algorithme va parcourir toutes les branches d'un niveau avant de passer aux feuilles du niveau suivant.



Solution de l'exercice 11 - Chemin en largeur

1.
```
def test_chemin_largeur(GL, depart, arrivee):
    n = len(GL)
    accessible = [True for _ in range(n)]
    a_visiter = [depart]
    while len(a_visiter)!=0:
        x = a_visiter.pop(0)
        if x==arrivee:
            return True
        accessible[x] = False
        for j in range(len(GL[x])): # on ajoute les voisins
            if accessible[GL[x][j][0]]==True:
                a_visiter.append(GL[x][j][0])
    return False
```

2.
```
def peres(GL, depart, arrivee):
    n = len(GL)
    accessible = [True for _ in range(n)]
    a_visiter = [depart]
    Lperes = [-1 for _ in range(n)]
    while len(a_visiter)!=0:
        x = a_visiter.pop(0)
        if x==arrivee:
            return Lperes
        accessible[x] = False
        for j in range(len(GL[x])):
```

```
            if accessible[GL[x][j][0]]==True:
                a_visiter.append(GL[x][j][0])
                print(GL[x][j][0])
                Lperes[GL[x][j][0]] = x
    return Lperes

def chemin_largeur(GL, depart, arrivee):
    Lperes = peres(GL, depart, arrivee)
    if len(Lperes)==0:
        return []
    else:
        res = [arrivee]
        x = arrivee
        while x!=depart:
            y = Lperes[x]
            res.append(y)
            x = y
    return res[::-1]
L = ["Paris", "Nantes", "Lyon", "Bordeaux", "Marseille", "Ajaccio"]
GM4 = [[-1,1,1,-1, -1,-1], [1,-1,1,1,-1,-1],[1,1,-1,-1,1,-1],[-1,-1,1,-1,
   -1,-1], [-1,-1,-1,-1,-1,-1]]
GL4 = [[[1, 1], [2, 1]], [[0, 1], [2, 1], [3, 1]], [[0, 1], [1, 1], [4,
   1]], [[2, 1]], []]
print(chemin_largeur(GL4, 0, 3))
```

Dans l'exemple de la figure 2 et pour aller de Paris à Bordeaux, on explore successivement Paris, Nantes puis Bordeaux.

Solution de l'exercice 12 - Matrice d'adjacence aléatoire

```
from random import randint, random
def mat_adj_alea(n, A, p):
    res = [[-1 for j in range(n)] for i in range(n)]
    for i in range(n):
        for j in range(i+1, n):
            if random()< p:
                x = randint(1, A)   # poids aléatoire
                res[i][j] = x
                res[j][i] = x
    return res

import numpy as np
print(np.array(mat_adj_alea(20, 5, 0.5)))
```

Solution de l'exercice 13 - Liste d'adjacence aléatoire

```
from random import randint, random
# solution de facilité :
def list_adj_alea(n, A, p):
    return mat_to_list(mat_adj_alea(n, A, p))
# autre solution
def list_adj_alea(n, A, p):
    L = [[] for i in range(n)]
    for i in range(n):
        for j in range(i+1, n):
            if random()< p:
                x = randint(1, A)   # poids aléatoire
                L[i].append([j, x])
                L[j].append([i, x])
    return L
print(len(list_adj_alea(20, 2, 0)))
```

Solution de l'exercice 14 - Test et statistiques
La probabilité augmente rapidement avec p. Si $p = 0$, cette probabilité vaut 0 car le nombre d'arêtes est nul (graphe non connexe).

```
def test(N):
    r = 0
    n = 20
    n2 = n*(n-1)/2 # nombre maximal d'arête
    A = 5  # la valeur de A importe peu
    p = 0.5
    for _ in range(N):
        # on tire une liste d'adjacence au hasard
        GL = list_adj_alea(n, A, p)

        # print(GL)
        for i in range(n):
            for j in range(i+1, n):
                if test_chemin_largeur(GL, i, j)==True:
                    r+=1
    return r/N/n2
print(test(100))
```