

Parcours

September 6, 2022

1 Principe

Le but de ce TP est de pratiquer les parcours vus en classe. Pour cela, on va suivre assez fidèlement le poly du cours http://mpsi.classeprepa.net/poly/chap_structuresdonnees_graphes.html#sec:parcours (codes d'accès utilisateur *mpsi* et mot de passe *2137*).

L'objectif n'est pas de réécrire un parcours *ex nihilo* mais de savoir modifier les parcours vus en classe pour répondre à des questions.

On va donc faire beaucoup de copier-coller en modifiant une ligne ou deux dans le parcours vu en classe

2 Graphes exemples

On va considérer ici des graphes donnés sous la forme de listes d'adjacences.

On donne quelques exemples de graphes qui nous serviront à tester les fonctions demandées et qui correspondent aux graphes présentés en classe :

- `complet(n)` qui renvoie le graphe complet à n sommets K_n
- `cycle(n)` qui renvoie un cycle non orienté de n sommets et `cycle_oriente(n)` qui renvoie le cycle orienté dans l'ordre croissant $0 \rightarrow 1 \rightarrow 2 \dots \rightarrow 0$
- `grille(n, d)` qui renvoie la grille de dimension d où les coordonnées sont dans $[0; n-1]$. On donne également deux fonctions `vecteur_vers_nombre` et `nombre_vers_vecteur` permettant de passer d'un élément de $[0; n-1]^d$ à son identifiant dans $[0; n^d - 1]$.

On considère également un graphe pseudo-aléatoire :

- `graphe_alea(germe, n, p)` qui renvoie un graphe non orienté pseudo-aléatoire sur $[0; n-1]$ en utilisant le `germe` passé en argument pour que deux appels avec les mêmes valeurs produisent le même graphe (comme les *seeds* présentes dans certains jeux). Pour déterminer si $y \in v_+(x)$ on génère aléatoirement un nombre r dans $[0; 1[$ et on vérifie si $r \leq p$. Cela revient à donner une probabilité p à chaque arête.

Attention les fonctions suivantes ont été écrites de manière compactes pour ne pas prendre trop de place dans l'énoncé. Il n'est pas nécessaire de comprendre leur fonctionnement.

```
[38]: def complet(n):  
      return [ [i for i in range(n) if i != j] for j in range(n) ]
```

```

def cycle(n):
    return [ [ (i+1) % n, (i-1) % n ] for i in range(n) ]

def cycle_oriente(n):
    return [ [ (i+1) % n ] for i in range(n) ]

def vecteur_vers_nombre(v, n):
    # prend un vecteur [v0, .., v(d-1)] dans  $[0;n-1]^d$ 
    # et renvoie un entier entre 0 et  $n^d - 1$  l'identifiant uniquement
    return sum( v[i] * n ** i for i in range(len(v)) )

def nombre_vers_vecteur(N, n, d):
    # réalise l'opération inverse
    return [ (N // n**k) % n for k in range(d) ]

def dist(v1, v2):
    return sum( abs(v1[i]-v2[i]) for i in range(len(v1)) )

def grille(n, d):
    return [ [ k for k in range(n**d)
                if dist(nombre_vers_vecteur(k,n,d),
↪ nombre_vers_vecteur(i,n,d)) == 1 ]
                for i in range(n**d) ]

def hypercube(n):
    return grille(2, n)

import random
def graphe_alea(germe, n, p):
    random.seed(germe)
    ladj = [ [] for _ in range(n) ]
    for i in range(n):
        for j in range(i+1,n):
            if random.random() <= p:
                ladj[i].append(j)
                ladj[j].append(i)
    return ladj

```

```
[39]: grille(3, 2)
```

```

[39]: [[1, 3],
       [0, 2, 4],
       [1, 5],
       [0, 4, 6],
       [1, 3, 5, 7],
       [2, 4, 8],
       [3, 7],

```

```
[4, 6, 8],  
[5, 7]]
```

```
[41]: graphe_alea(42, 7, 0.25)
```

```
[41]: [[2, 4], [3, 5, 6], [0, 4, 5], [1, 5], [0, 2, 6], [1, 2, 3], [1, 4]]
```

3 Premiers pas avec le parcours en profondeur

On redonne ici le programme vu en classe pour le parcours en profondeur récursif.

```
[72]: INCONNU = 0  
      DECOUVERT = 1  
      TRAITE = 2  
  
      def parcours(ladj, x, etat):  
          if etat[x] != TRAITE:  
              print(x)  
              for y in ladj[x]:  
                  if etat[y] == INCONNU:  
                      etat[y] = DECOUVERT  
                      parcours(ladj, y, etat)  
          etat[x] = TRAITE  
  
      def lance_parcours(ladj, x):  
          # État initial inconnu pour tous  
          etat = [ INCONNU ] * len(ladj)  
          etat[x] = DECOUVERT  
          parcours(ladj, x, etat)
```

Pour bien comprendre ce qu'on vous demande, il faut imaginer que `parcours` est une sorte de boucle `for` qui va parcourir un graphe en partant d'un sommet.

Ainsi, il faut comprendre le programme précédent comme :

```
def lance_parcours(ladj, x):  
    for y in accessibles depuis x dans ladj:  
        print(y)
```

La fonction récursive va s'appeler autant de fois que d'itérations dans la pseudo boucle précédente et `etat` n'est là que pour permettre à cette fonction de ne pas traiter trop de fois les sommets.

Dans la suite, les fonctions qu'on vous demande de faire vont correspondre à un pseudo-code :

```
def mafonction(ladj, x):  
    for y in accessibles depuis x dans ladj:  
        ## TRAITEMENT DU SOMMET y
```

```

## TRAITEMENT DES RÉSULTATS
return valeur calculée

```

qui sera retraduit sous la forme suivante :

```

def parcours_mafonction(ladj, x, etat):
    if etat[x] != TRAITE:
        ## TRAITEMENT DU SOMMET x
        for y in ladj[x]:
            if etat[y] == INCONNU:
                etat[y] = DECOUVERT
                parcours_mafonction(ladj, y, etat)
        etat[x] = TRAITE

def mafonction(ladj, x):
    # État initial inconnu pour tous
    etat = [ INCONNU ] * len(ladj)
    etat[x] = DECOUVERT
    parcours_mafonction(ladj, x, etat)
    ## TRAITEMENT DES RÉSULTATS
    return valeur calculée

```

Il est possible qu'on souhaite rajouter des arguments à la fonction réalisant le parcours : **exactement** comme lorsqu'on rajoute des variables qui seront mises à jour dans une boucle.

3.1 Compter les sommets traités

On va modifier le parcours précédent pour écrire une fonction `compte_traites(ladj, x)` qui renvoie le nombre de sommets traités dans un parcours en profondeur du graphe `ladj` en partant du sommet `x`. Vous ne devez modifier ces fonctions qu'en rajoutant du code dans la partie indiquée.

```

[1]: INCONNU = 0
    DECOUVERT = 1
    TRAITE = 2

def parcours_compte(ladj, x, etat):
    if etat[x] != TRAITE:
        for y in ladj[x]:
            if etat[y] == INCONNU:
                etat[y] = DECOUVERT
                parcours_compte(ladj, y, etat)
        etat[x] = TRAITE

def compte_traites(ladj, x):
    # État initial inconnu pour tous
    etat = [ INCONNU ] * len(ladj)
    etat[x] = DECOUVERT
    parcours_compte(ladj, x, etat)

```

```

# BEGIN SOLUTION
compte = 0
for i in etat:
    if i:
        compte += 1
return compte
# END SOLUTION

```

```

[75]: from civ import *
init()
test(compte_traites, cycle(5), 0, 5) # parcours de C5 en partant de 0
test(compte_traites, grille(3, 2), 0, 9) # parcours de G32 en partant de 0
test(compte_traites, graphe_alea(42, 10, 0.05), 0, 4)
test(compte_traites, graphe_alea(42, 10, 0.1), 0, 9)
valide()

```

OK!: `compte_traites([[1, 4], [2, 0], [3, 1], [4, 2], [0, 3]], 0)` vaut 5
 OK!: `compte_traites([[1, 3], [0, 2, 4], [1, 5], [0, 4, 6], [1, 3, 5, 7], [2, 4, 8], [3, 7], [4, 6, 8], [5, 7]], 0)` vaut 9
 OK!: `compte_traites([[2], [2, 5], [0, 1, 5], [], [], [1, 2], [9], [], [], [6]], 0)` vaut 4
 OK!: `compte_traites([[2, 8], [2, 5], [0, 1, 5], [6, 7], [], [1, 2], [3, 9], [3], [0, 9], [6, 8]], 0)` vaut 9
 Tous les tests sont valides

Si G est un graphe non orienté, on a vu qu'un parcours depuis x permettait d'explorer la composante connexe de x , c'est-à-dire $\bar{x}$ les sommet accessibles depuis x .

On en déduit que G est **connexe** $\iff \forall x \in S, \bar{x} = S$.

Ici, il suffit de vérifier que tous les sommets sont traités après un parcours depuis n'importe quel sommet.

On va donc en déduire une fonction `connexe(ladj)` qui renvoie un booléen indiquant si un graphe non orienté donnée en liste d'adjacence est connexe.

ATTENTION pas la peine de recoder un parcours, on peut utiliser la fonction précédente.

```

[76]: def connexe(ladj):
    # BEGIN SOLUTION
    return compte_traites(ladj, 0) == len(ladj)
    # END SOLUTION

```

```

[77]: from civ import *
init()
test(connexe, cycle(5), True)
test(connexe, grille(3, 2), True)
test(connexe, graphe_alea(42, 10, 0.05), False)
test(connexe, graphe_alea(42, 10, 0.1), False)

```

```
test(connexe, graphe_alea(42, 10, 0.25), True)
valide()
```

```
OK!: connexe([[1, 4], [2, 0], [3, 1], [4, 2], [0, 3]]) vaut True
OK!: connexe([[1, 3], [0, 2, 4], [1, 5], [0, 4, 6], [1, 3, 5, 7], [2, 4, 8], [3,
7], [4, 6, 8], [5, 7]]) vaut True
OK!: connexe([[2], [2, 5], [0, 1, 5], [], [], [1, 2], [9], [], [], [6]]) vaut
False
OK!: connexe([[2, 8], [2, 5], [0, 1, 5], [6, 7], [], [1, 2], [3, 9], [3], [0,
9], [6, 8]]) vaut False
OK!: connexe([[2, 4, 8], [2, 3, 5, 6, 9], [0, 1, 5, 9], [1, 6, 7], [0], [1, 2],
[1, 3, 9], [3, 8], [0, 7, 9], [1, 2, 6, 8]]) vaut True
Tous les tests sont valides
```

Dans le dernier exemple de test, on peut voir que la connexité d'un graphe aléatoire dépend de la probabilité d'existence d'une arête. La dernière partie de ce TP propose d'explorer cette propriété.

3.2 Composantes connexes

Pour obtenir les composantes connexes d'un graphe, on va commencer par modifier le parcours pour qu'il renvoie la liste des sommets traités. Pour cela, on va rajouter une liste en paramètre et y ajouter les sommets traités.

Écrire une fonction `composante(ladj, x)` qui renvoie la liste des sommets de la composante connexe de `x` **triés dans l'ordre croissant**.

Pour trier un tableau `t` en Python, il suffit d'appeler `t.sort()`. Cette fonction ne renvoie rien, elle va modifier `t` en place.

```
[123]: INCONNU = 0
        DECOUVERT = 1
        TRAITE = 2

def parcours_comp(ladj, x, etat, comp):
    if etat[x] != TRAITE:
        # BEGIN SOLUTION
        comp.append(x)
        # END SOLUTION
        for y in ladj[x]:
            if etat[y] == INCONNU:
                etat[y] = DECOUVERT
                parcours_comp(ladj, y, etat, comp)
        etat[x] = TRAITE

def composante(ladj, x):
    # État initial inconnu pour tous
    etat = [ INCONNU ] * len(ladj)
    etat[x] = DECOUVERT
    # BEGIN SOLUTION
```

```

comp = []
parcours_comp(ladj, x, etat, comp)
comp.sort()
return comp
# END SOLUTION

```

```

[127]: from civ import *
init()
test(composante, cycle(5), 0, [0,1,2,3,4])
test(composante, graphe_alea(42, 10, 0.1), 0, [0, 1, 2, 3, 5, 6, 7, 8, 9])
valide()

```

OK!: composante([[1, 4], [2, 0], [3, 1], [4, 2], [0, 3]], 0) vaut [0, 1, 2, 3, 4]

OK!: composante([[2, 8], [2, 5], [0, 1, 5], [6, 7], [], [1, 2], [3, 9], [3], [0, 9], [6, 8]], 0) vaut [0, 1, 2, 3, 5, 6, 7, 8, 9]

Tous les tests sont valides

Pour obtenir toutes les composantes connexes, il suffit de relancer le parcours depuis le prochain sommet non traités et ajouter la composante obtenue à la liste des composantes.

Écrire une fonction `composantes(ladj)` qui renvoie la liste des composantes connexes du graphe `ladj` où les composantes connexes sont triés dans l'ordre croissant de leur plus petit élément (normalement, c'est ainsi que vous devriez les calculer naturellement). On appellera `parcours_comp` et non `composante` pour conserver le même tableau `etat`.

```

[134]: def composantes(ladj):
# BEGIN SOLUTION
n = len(ladj)
etat = [ INCONNU ] * n
comps = []
for x in range(n):
    if etat[x] == INCONNU:
        comp = []
        etat[x] = DECOUVERT
        parcours_comp(ladj, x, etat, comp)
        comp.sort()
        comps.append(comp)
return comps
# END SOLUTION

```

```

[137]: from civ import *
init()
test(composantes, cycle(5), [[0,1,2,3,4]])
test(composantes, graphe_alea(42, 10, 0.1), [[0, 1, 2, 3, 5, 6, 7, 8, 9], [4]])
test(composantes, graphe_alea(42, 10, 0.05), [[0, 1, 2, 5], [3], [4], [6, 9],
↪ [7], [8]])
valide()

```

```

OK!: composantes([[1, 4], [2, 0], [3, 1], [4, 2], [0, 3]]) vaut [[0, 1, 2, 3, 4]]
OK!: composantes([[2, 8], [2, 5], [0, 1, 5], [6, 7], [], [1, 2], [3, 9], [3], [0, 9], [6, 8]]) vaut [[0, 1, 2, 3, 5, 6, 7, 8, 9], [4]]
OK!: composantes([[2], [2, 5], [0, 1, 5], [], [], [1, 2], [9], [], [], [6]])
vaut [[0, 1, 2, 5], [3], [4], [6, 9], [7], [8]]
Tous les tests sont valides

```

3.3 Arbre du parcours et chemins

Comme vu dans le cours, il est possible de définir des liens de parentés entre deux sommets. On notera `parent[y] = x` quand c'est l'arête de $x \rightarrow y$ qui a permis de découvrir y .

Question Écrire une fonction `arbre(ladj, x)` qui renvoie le tableau `parent` ainsi défini. Quand un sommet n'a pas de parent défini, on mettra `None` comme valeur. En particulier, comme on part de x , on a forcément `parent[x] == None`.

```

[1]: INCONNU = 0
    DECOUVERT = 1
    TRAITE = 2

    def parcours_parent(ladj, x, etat, parent):
        if etat[x] != TRAITE:
            # BEGIN SOLUTION
            for y in ladj[x]:
                if etat[y] == INCONNU:
                    etat[y] = DECOUVERT
                    parent[y] = x
                    parcours_parent(ladj, y, etat, parent)
            # END SOLUTION
            etat[x] = TRAITE

    def arbre(ladj, x):
        # État initial inconnu pour tous
        etat = [ INCONNU ] * len(ladj)
        etat[x] = DECOUVERT
        # BEGIN SOLUTION
        parent = [ None ] * len(ladj)
        parcours_parent(ladj, x, etat, parent)
        return parent
        # END SOLUTION

```

```

[147]: from civ import *
    init()
    test(arbre, cycle(5), 0, [None, 0, 1, 2, 3])
    test(arbre, graphe_alea(42, 10, 0.1), 0, [None, 2, 0, 6, None, 1, 9, 3, 0, 8])
    valide()

```

```

OK!: arbre([[1, 4], [2, 0], [3, 1], [4, 2], [0, 3]], 0) vaut [None, 0, 1, 2, 3]

```

OK!: arbre([[2, 8], [2, 5], [0, 1, 5], [6, 7], [], [1, 2], [3, 9], [3], [0, 9], [6, 8]], 0) vaut [None, 2, 0, 6, None, 1, 9, 3, 0, 8]

Tous les tests sont valides

Question Écrire une fonction `chemin(y, parent)` qui prend un sommet `y` dans la composante connexe de `x`, un tableau `parent` rendu par la fonction précédente et renvoie un chemin de `x` vers `y` donné sous la forme d'une liste de sommets.

Note : on pourra utiliser `t.reverse()` pour renverser l'ordre des éléments dans un tableau. Car ici, c'est plus simple de remplir en mettant `y` en premier dans un premier temps.

```
[148]: def chemin(y, parent):  
        # BEGIN SOLUTION  
        l = [ y ]  
        while parent[y] != None:  
            y = parent[y]  
            l.append(y)  
        l.reverse()  
        return l  
        # END SOLUTION
```

```
[152]: from civ import *  
init()  
test(chemin, 3, [None, 0, 1, 2, 3], [0, 1, 2, 3])  
test(chemin, 5, [None, 2, 0, 6, None, 1, 9, 3, 0, 8], [0, 2, 1, 5])  
valide()
```

OK!: `chemin(3, [None, 0, 1, 2, 3])` vaut `[0, 1, 2, 3]`

OK!: `chemin(5, [None, 2, 0, 6, None, 1, 9, 3, 0, 8])` vaut `[0, 2, 1, 5]`

Tous les tests sont valides

4 Affichage du parcours sur une grille

Cette partie est optionnelle.

Si on applique le parcours initial au graphe d'une grille, ce n'est pas très lisible car on obtient le numéro d'un vecteur et pas le vecteur lui-même.

`lance_parcours(grille(3, 2), 0)`

va afficher :

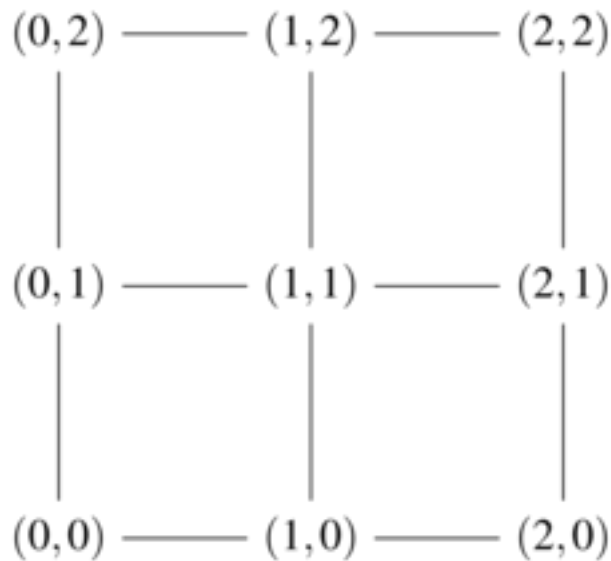
```
0  
1  
2  
5  
4  
3  
6  
7
```

En utilisant la fonction `nombre_vers_vecteur(5, 3, 2)` on peut obtenir l'étiquette `[2, 1]` à la place du nombre 5, ce qui est plus lisible.

Question Modifiez le parcours ci-dessous pour définir une fonction `parcours_grille(n, v)` qui, étant donné un tableau `v` représentant un vecteur, effectue l'affichage des vecteurs rencontrés. Par exemple, `parcours_grille(3, [1,2])` devrait afficher :

```
[1, 2]
[1, 1]
[1, 0]
[0, 0]
[0, 1]
[0, 2]
[2, 0]
[2, 1]
[2, 2]
```

Rappel, le graphe $G_{3,2}$ correspondant à la représentation :



ATTENTION ici on vous recommande de renommer la fonction `parcours` pour que `connexe` continue de fonctionner.

```
[80]: # BEGIN SOLUTION
def parcours2(ladj, x, n, d, etat):
    if etat[x] != TRAITE:
        print(nombre_vers_vecteur(x, n, d))
        for y in ladj[x]:
```

```

        if etat[y] == INCONNU:
            etat[y] = DECOUVERT
            parcours2(ladj, y, n, d, etat)
        etat[x] = TRAITE

def parcours_grille(n, v):
    d = len(v)
    g = grille(n, d)
    x = vecteur_vers_nombre(v, n)
    etat = [ INCONNU ] * len(g)
    etat[x] = DECOUVERT
    parcours2(g, x, n, d, etat)
# END SOLUTION

parcours_grille(3, [1,2])

```

```

[1, 2]
[1, 1]
[1, 0]
[0, 0]
[0, 1]
[0, 2]
[2, 0]
[2, 1]
[2, 2]

```

Question Ici, on a effectué un affichage lorsqu'un sommet est traité, mais on veut plutôt indiquer les arêtes suivies. Modifiez le parcours précédent pour afficher les arêtes d'un parcours de grille.

L'appel à `parcours_grille(3, [1,2])` devra afficher :

```

[1, 2] -> [1, 1]
[1, 1] -> [1, 0]
[1, 0] -> [0, 0]
[0, 0] -> [0, 1]
[0, 1] -> [0, 2]
[1, 0] -> [2, 0]
[2, 0] -> [2, 1]
[2, 1] -> [2, 2]

```

```

[81]: # BEGIN SOLUTION
def affiche(x, y, n, d):
    print(nombre_vers_vecteur(x, n, d),
          '->',
          nombre_vers_vecteur(y, n, d))

def parcours3(ladj, x, n, d, etat):
    if etat[x] != TRAITE:
        for y in ladj[x]:

```

```

        if etat[y] == INCONNU:
            affiche(x, y, n, d)
            etat[y] = DECOUVERT
            parcours3(ladj, y, n, d, etat)
    etat[x] = TRAITE

def parcours_grille(n, v):
    d = len(v)
    g = grille(n, d)
    x = vecteur_vers_nombre(v, n)
    etat = [ INCONNU ] * len(g)
    etat[x] = DECOUVERT
    parcours3(g, x, n, d, etat)
# END SOLUTION

parcours_grille(3, [1,2])

```

```

[1, 2] -> [1, 1]
[1, 1] -> [1, 0]
[1, 0] -> [0, 0]
[0, 0] -> [0, 1]
[0, 1] -> [0, 2]
[1, 0] -> [2, 0]
[2, 0] -> [2, 1]
[2, 1] -> [2, 2]

```

Rappel il est possible d'afficher une matrice de nombres, c'est-à-dire un tableau de tableaux de nombres tous de même longueur, comme une image avec la commande suivante :

```

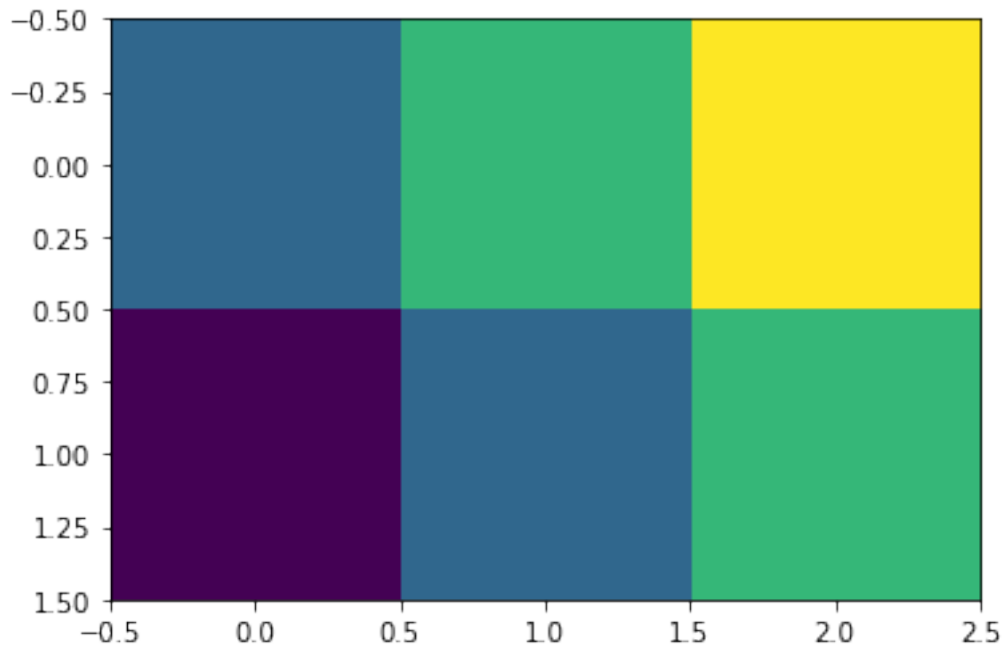
[64]: import matplotlib.pyplot as plt
      plt.imshow( [ [ 1, 2, 3], [0, 1, 2] ] )

```

```

[64]: <matplotlib.image.AxesImage at 0x7fcbea547320>

```



Attention les lignes sont affichées comme en mathématiques de haut en bas.

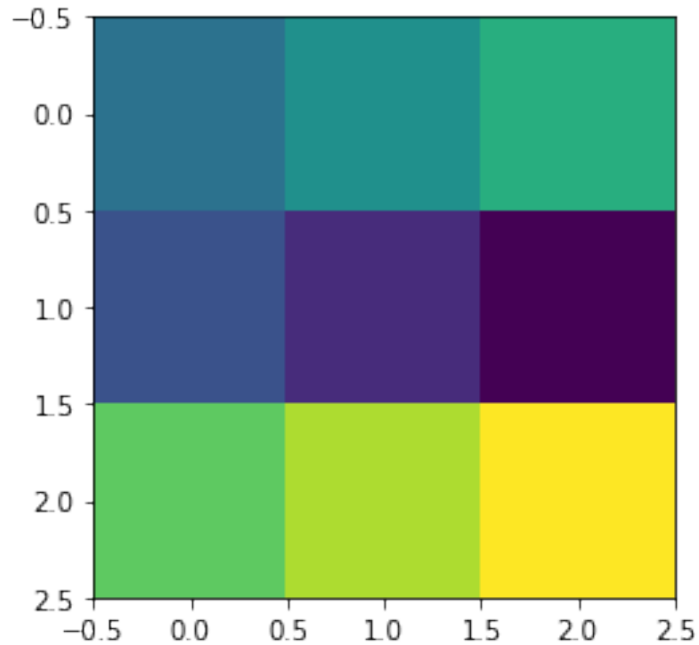
Question écrire une fonction `parcours_grille2d(n, v)` qui effectue un parcours de G_n^2 à partir du vecteur v et crée une matrice M où $M[i][j]$ est le numéro de $[i, j]$ dans l'ordre du parcours.

Par exemple, sur la grille G_3^2 on a vu que les sommets traités depuis $[1, 2]$ sont : $[1, 2]$, $[1, 1]$, $[1, 0]$, $[0, 0]$, $[0, 1]$, $[0, 2]$, $[2, 0]$, $[2, 1]$, $[2, 2]$.

Ce qui correspond à une matrice d'ordre : $\begin{bmatrix} 3 & 4 & 5 \\ 2 & 1 & 0 \\ 6 & 7 & 8 \end{bmatrix}$ et donc à l'affichage :

```
[65]: plt.imshow([ [ 3, 4, 5 ], [ 2, 1, 0 ], [6, 7, 8] ])
```

```
[65]: <matplotlib.image.AxesImage at 0x7fcbea457588>
```



Remarque pour avoir le numéro du sommet, on utilisera une liste de sommets traites dont la longueur indiquera le numéro.

```
[82]: # BEGIN SOLUTION
def parcours4(ladj, x, n, etat, traites, mat):
    if etat[x] != TRAITE:
        a, b = nombre_vers_vecteur(x, n, 2)
        mat[a][b] = len(traites)
        traites.append(x)
        for y in ladj[x]:
            if etat[y] == INCONNU:
                etat[y] = DECOUVERT
                parcours4(ladj, y, n, etat, traites, mat)
        etat[x] = TRAITE

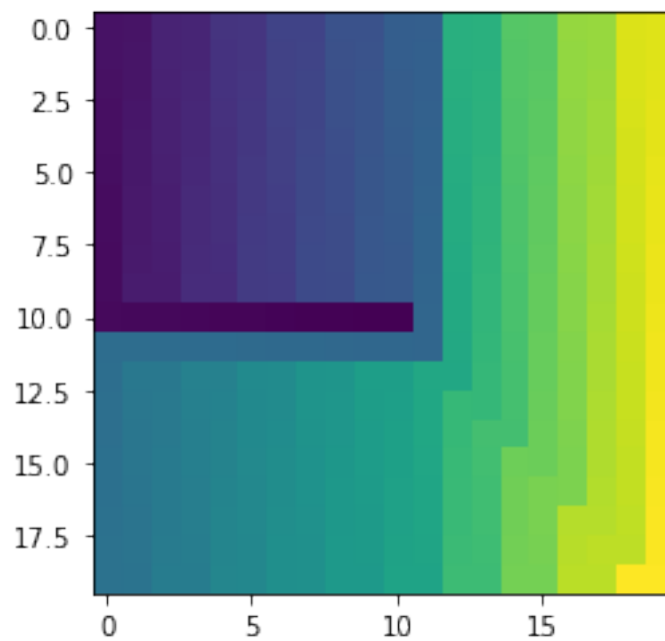
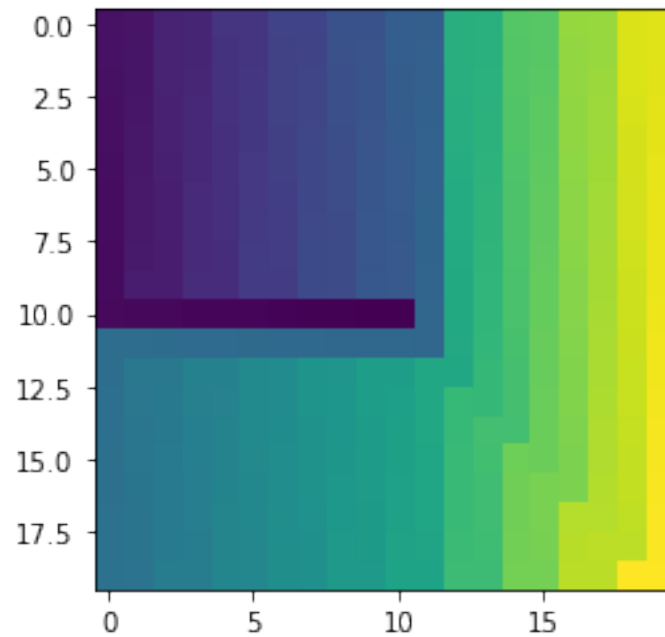
def parcours_grille2d(n, v):
    g = grille(n, 2)
    x = vecteur_vers_nombre(v, n)
    etat = [ INCONNU ] * len(g)
    etat[x] = DECOUVERT
    mat = [ ]
    for _ in range(n):
        l = [ ]
        for _ in range(n):
            l.append( -1 )
        mat.append(l)
```

```

    parcours4(g, x, n, etat, [], mat)
    plt.imshow(mat)
# END SOLUTION

parcours_grille2d(20, [10, 10])

```



L'appel précédent doit afficher :

5 Propriétés du graphe aléatoire

Cette partie est optionnelle.

On va explorer ici des propriétés du graphe aléatoire en fonction du nombre de sommets et de la probabilité d'avoir une arête.

Question écrire une fonction `table_connexe(germe, n, N)` qui renvoie un tableau `t` de `N` cases telle que `t[i]` contient un booléen indiquant si le graphe `graphe_alea(germe, n, i/N)` est connexe.

ATTENTION pour pouvoir utiliser `connexe`, il faut s'assurer que vous n'avez pas redéfini la fonction `parcours` qu'elle appelle.

```
[83]: # BEGIN SOLUTION
def table_connexe(germe, n, N):
    return [ connexe(graphe_alea(germe, n, i/N)) for i in range(N) ]
# END SOLUTION
```

```
[91]: from civ import *
init()
test(table_connexe, 42, 10, 10, [False, False, False, True, True, True, True,
↪True, True, True])
valide()
```

OK!: `table_connexe(42, 10, 10)` vaut `[False, False, False, True, True, True, True, True, True, True]`

Tous les tests sont valides

On remarque que les tableaux renvoyés par la fonction précédente semblent avoir tous la même forme : une suite de `False` suivi d'une suite de `True`. Cela signifierait qu'il existe une probabilité p tel que les graphes sont connexes avec une probabilité $p' \geq p$ et non connexes avec $p' < p$.

En admettant que dce soit le cas, pour obtenir une estimation de p , on peut réaliser une dichotomie. Écrire une fonction `proba_seuil(germe, n, eps)` qui renvoie à `eps` près la probabilité seuil.

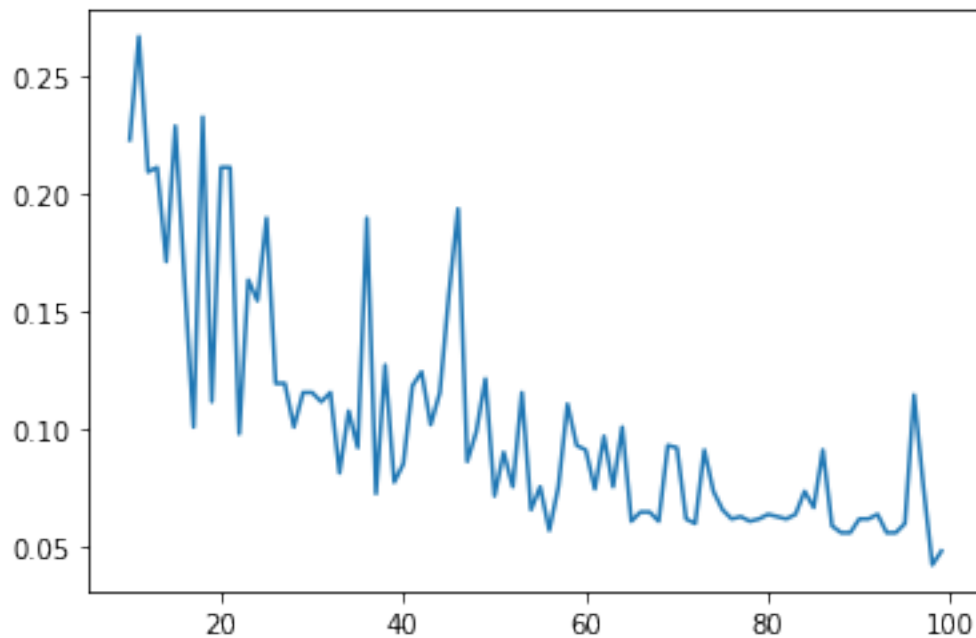
```
[100]: # BEGIN SOLUTION
def proba_seuil(germe, n, eps):
    a, b = 0, 1
    while b-a > eps:
        m = (a+b)/2
        if connexe(graphe_alea(germe, n, m)):
            b = m
        else:
            a = m
    return (a+b)/2
# END SOLUTION
```

```
[104]: p = proba_seuil(42, 20, 0.01)
assert( abs(p-0.214) < 0.01 )
```

On peut tracer un graphe indiquant, pour un germe donné, ici 42, les probabilités seuils :

```
[111]: sommets = []
       proba = []
       for n in range(10, 100):
           sommets.append(n)
           proba.append(proba_seuil(42, n, 0.001))
       plt.plot( sommets, proba )
```

```
[111]: [<matplotlib.lines.Line2D at 0x7fcbe9c6f080>]
```



Le graphe précédent est très bruité, afin de le lisser, écrire une fonction `proba_seuil_lisse(n)` qui moyenne la fonction précédente pour les germes 0 à 99.

```
[116]: # BEGIN SOLUTION
       def proba_seuil_lisse(n):
           s = 0
           N = 100
           for k in range(N):
               s += proba_seuil(k, n, 0.001)
           return s / N
       # END SOLUTION
```

```
[117]: sommets = []
       proba = []
       for n in range(10, 50):
```

```
sommets.append(n)
proba.append(proba_seuil_lisse(n))
plt.plot( sommets, proba )
```

[117]: [<matplotlib.lines.Line2D at 0x7fcbe9bc15f8>]

