Devoir - Bilan 2nd semestre

Colorer un graphe

I - Des algorithmes pour colorer un graphe

1 Introduction sur un exemple

Q1. Un graphe peut être représenté par une matrice d'adjacence. Sur le document réponse, la matrice incomplète du graphe G_{ex} est donnée. La compléter et expliquer le processus de construction sur votre copie.

$$M = \begin{pmatrix} \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} & 1 & 0 & 1 & 1 \\ \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{1} & 0 & 0 & 0 & 0 \\ \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} & 0 & 0 & 0 & 0 \\ \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix}$$

Chaque ligne correspond à un sommet de départ et chaque colonne à un sommet d'arrivée. Un 0 à la case (i,j) représente l'absence d'arête partant du sommet i et allant vers le sommet j. Dans le cas d'un graphe non orienté comme ici, on a forcement une matrice symétrique : `M[i,j]=M[j,i]`.

Q2. Une autre manière de représenter en mémoire un graphe est d'utiliser une liste d'adjacence.
La liste d'adjacence d'un graphe (constitué de sommets numérotés de 0 à $n-1$) est une liste `LA`, telle que `LA[i]` est la liste des sommets voisins du sommet `i`. Par convention, les voisins énumérés dans `LA[i]` seront listés dans l'ordre croissant.
Donner la liste d'adjacence du graphe G_{ex} présenté en exemple.

```
LA=[[1,3,4,6,7], [0,2,3], [1,3], [0,1,2,4], [0,3,5,6,7], [4,6,7], [0,4,5,7], [0,4,5,6]]
```

Q3. Donner un avantage et un inconvénient d'une représentation par matrice d'adjacence.
Donner un avantage et un inconvénient d'une représentation par liste d'adjacence.

Avantage d'une représentation par matrice d'adjacence : calculs facilement systématisables et elle s'adapte facilement à l'ajout de pondérations sur les arcs ou les arêtes.

Inconvénient d'une représentation par matrice d'adjacence : gâchis de mémoire pour les graphes ayant peu d'arêtes au vu de leur nombre de sommets. Redondance d'information pour les graphes non orientés et nécessite la présence d'une liste de noms de sommets quand ceux-ci ne sont pas de simples numéros qui se suivent.

Avantage d'une représentation par liste d'adjacence : plus rapide à remplir car on rentre directement les successeurs. Pas de données inutiles en mémoire.

Inconvénient d'une représentation par matrice d'adjacence : s'adapte moins facilement à l'ajout de pondérations sur les arcs ou les arêtes (nécessité de faire des listes de tuple). Redondance d'information pour les graphes non orientés et nécessite la présence d'une liste de noms de sommets quand ceux-ci ne sont pas de simples numéros qui se suivent

---

Q4. On rappelle que le degré d'un sommet `s` d'un graphe G est le nombre de voisins du sommet `s`, c'est-à-dire le nombre de sommets reliés à `s`. Compléter, sur le document réponse, le tableau des degrés des différents sommets du graphe G_{ex}.

Sommet	0	1	2	3	4	5	6	7
Degré	5	3	2	4	5	3	4	4

---

2 Créer une liste d'adjacence à partir d'un fichier texte

Q5. Rappeler la définition d'ordre d'un graphe non orienté .

Le degré d'un sommet d'un graphe non orienté est le nombre d'arêtes reliées à ce sommet. Le degré d'un graphe est le plus grand degré de sommet. L'ordre d'un graphe correspond à son nombre de sommets.

Q6. À l'aide de l'annexe fournie en fin de sujet, donner un code permettant à partir du fichier `graphe.csv` (pas besoin d'indiquer son chemin d'accès) d'obtenir la liste d'adjacence `Ladj` du graphe représenté.

```
1 Ladj=[]
2 fichier=open('graphe.csv','r') # Ouverture du fichier en mode lecture
3 for ligne in fichier # On parcourt le fichier ligne à ligne
4     ligne=ligne.strip() # On supprime le nen fin de ligne
5     liste=ligne.split(',') # On transforme la ligne en liste, l'élément de sé
          paration est la virgule
6     temp=[int(i) for i in liste] # Attention à bien convertir les noeuds obtenus
          en nombre entier qui étaient pour l'instant en chaîne de caractères
7     Ladj.append(temp[1:]) # On récupère tous les voisins du noeud considéré.
```

---

3 Tester si une coloration est valide

Q7. Écrire une fonction `voisins` avec trois arguments, deux nombres entiers distincts `i` et `j` et une liste d'adjacence `LA` représentant un graphe, qui renvoie `True` si les sommets numérotés `i` et `j` sont reliés par une arête et `False` sinon.

Le code à établir est assez simple. Pour savoir si `i` et `j` sont des voisins, il suffit de vérifier si `j` est présent dans la liste `LA[i]`. (Ou inversement vérifier si `i` est dans la liste `LA[j]`). Il s'agit donc simplement d'un code de recherche d'un élément dans une liste.

```
def voisins(i,j,LA):
    for voisin in LA[i]: # On parcourt les voisins de noeud i
        if voisin==j: # Si un des voisins est le noeud j alors ...
            return True # ... la fonction renvoie True
    return False # Si aucun voisin ne correspond au noeud j, la fonction renvoie
        False
```

Q8. Écrire la fonction `coloration_valide` avec pour arguments une liste d'adjacence `LA` et une liste de couleurs `C`, qui renvoie `True` si la coloration est valide, `False` sinon.

Pour répondre à cette question, il faut vérifier que pour chaque sommet, noté `pays1`, chacun de ses voisins (noté `pays2`) ait une coloration différente.

```
def coloration_valide(LA,C) :
    for pays1 in range(len(LA)) : # Parcours des différents pays du graphe (par
        ordre numérique)
        for pays2 in LA[pays1] : # Parcours des voisins du pays1
            if C[pays1] == C[pays2] # Si deux pays voisins ont la même couleur
                ...
                return False # ... alors on renvoie False.
    return True # Si aucun couple de voisin n'a de couleurs identiques, alors la
        fonction renvoie True.
```

Q9. Pour un graphe comportant n sommets, quelle est la complexité temporelle dans le pire des cas de la fonction `coloration_valide` ?

Dans le pire des cas, chaque sommet est voisin de tous les autres : la complexité sera donc en $\mathcal{O}(n \times n) = O(n^2)$ (la première boucle `for` a n itérations et la deuxième en a n aussi dans le pire des cas). Évidemment, pour que cette réponse soit valide, on précise que les lignes de codes situées au sein des boucles `for` imbriquées est à coût constant

---

4 Un algorithme intuitif de coloration

Q10. On suppose (dans cette question seulement) que n est une constante déjà définie. Écrire la ou les instruction(s) permettant de créer une liste initiale `C` composée de n éléments valant -1.

On peut tout simplement faire :

```
C=[-1] * n
```

ou encore :

```
C=[-1 for i in range(n)]
```

ou bien :

```
C=[]
for i in range(n) :
    C.append(-1)
```

La première solution est à privilégier (notamment lors de l'épreuve orale).

Q11. Compléter, sur le document réponse, la fonction `colore_sommet` ayant trois arguments, la liste `C` des couleurs attribuées, le numéro `s` du sommet à colorer et la liste d'adjacence `LA` caractérisant le graphe.
Cette fonction ne renvoie rien mais modifie la liste `C` en donnant à `C[s]` la plus petite couleur possible,

en fonction des couleurs des sommets voisins qui sont déjà colorés.
Par exemple, pour le graphe G_{ex}, prenons `C=[0,1,-1,-1,-1,-1,-1,-1]`. Les sommets 0 et 1 ont donc déjà été colorés avec les couleurs `C[0]=0` et `C[1]=1`.
L'appel `colore_sommet(C,2,LA)` modifie la liste `C` en `C=[0,1,0,-1,-1,-1,-1,-1]`. Cela veut dire que le sommet 2, adjacent au sommet 1 mais pas au sommet 0, a reçu la même couleur que le sommet 0.

Dans un premier temps, on crée une liste `coul_vois` contenant la couleur de tous les voisins du sommet `s`. Pour cela, on parcourt la liste `LA[s]` et ajoute à `coul_vois` grâce à la liste `C`, la couleur de chacun des voisins. Ensuite, dans un second temps, il faut déterminer la couleur de valeur minimale non utilisée. Pour cela, on initialise une variable `mini` à 0. On teste ensuite si cette valeur est présente dans la liste `coul_vois`. Si c'est le cas, on incrémente `mini` et on recommence les tests jusqu'à trouver une valeur de `mini` qui ne soit pas présente dans `coul_vois`.

```
def colore_sommet(C,s,LA) :
    # on détermine la liste des couleurs des voisins de s déjà colorés
    coul_vois=[ ]
    for voisin in LA[s] :
        coul_vois.append(C[voisin])
    # coul_vois est maintenant déterminée et on recherche la plus petite couleur
      , notée num_coul, absente de coul_vois.
    mini , fin_recherche = 0 , False
    n = len(coul_vois)
    while fin_recherche == False:
        indice = 0
        while indice < n :
            if coul_voisin[indice] == mini : # On teste si mini dans la liste
               coul_vois ...
                indice = n+1 #... si c'est le cas, on sort de la boucle while
                   indice <n (avec un indice plus grand que n)
            else :
                indice +=1
        if indice == n : # Si on a fini de parcourir la liste coul_vois, on arrê
           te les différentes boucles.
            num_coul , fin_recherche = mini , True
        else : # Sinon (dans le cas où indice=n+1), on itère la valeur de mini
           et on recommence les différents tests.
            mini+=1
    # la valeur num_coul trouvée devient la couleur du sommet s
    C[s] = num_coul
```

On peut faire un code plus simple à lire grâce à la commande `in`.

```
def colore_sommet(C,s,LA) :
    # on détermine la liste des couleurs des voisins de s déjà colorés
    coul_vois=[ ]
    for voisin in LA[s] :
        coul_vois.append(C[voisin])
    # coul_vois est maintenant déterminée et on recherche la plus petite couleur
      , notée mini, absente de coul_vois.
    mini , fin_recherche = 0 , False
    while fin_recherche == False:
        if mini in coul_vois:
            mini=mini+1
        else:
            fin_recherche = True
    num_coul=mini
    # la valeur num_coul trouvée devient la couleur du sommet s
    C[s] = num_coul
```

Q12. À l'aide de **Q11**, écrire une fonction `colorer1` avec pour argument une liste `LA` caractérisant un graphe, qui crée et renvoie la liste `C` des numéros des couleurs attribuées en colorant les sommets un par un par ordre croissant de leurs numéros.

Par exemple, l'application de la fonction `colorer1` au graphe G_{ex} renverra la liste de couleurs `[0,1,0,2,1,0,2,3]`.

Pour répondre, à cette question, il faut initialiser une liste `C` (comme vu à la question **Q10**) et modifier cette même liste pour chaque sommet `s` à l'aide de la fonction établie à la question **Q11**.

```
1 def colorer1(LA):
2     n=len(LA)
3     C=[-1]*n
4     for s in range(n):
5         colore_sommet(C, s, LA)
6     return C
```

Q13. L'ordre de coloration imposé à la question précédente est arbitraire. On souhaite maintenant colorer le graphe en traitant les sommets selon un ordre arbitraire donné en argument. Écrire une fonction `colorer2` analogue à `colorer1` et avec un argument supplémentaire, une liste ordre fixant l'ordre de coloration des pays.
Par exemple `colorer2([0,2,4,6,1,3,5,7],LA)` colorera le graphe G_{ex} en commençant d'abord par le sommet 0, puis en continuant par les sommets 2, 4, 6, etc..

Cette question est très similaire à la précédente. Il suffit de modifier la ligne 4 du code précédent, en adaptant la boucle `for`.

```
1 def colorer2(ordre,LA):
2     n=len(LA)
3     C=[-1]*n
4     for s in ordre :
5         colore_sommet(C, s, LA)
6     return C
```

Q14. Donner la liste des couleurs renvoyée par `colorer2` pour colorer le graphe G_{ex} donné en exemple en prenant `ordre=[7,6,5,4,3,2,1,0]`.
Combien de couleurs ont-elles été utilisées ?

On aurait `C=[4, 2, 1, 0, 3, 2, 1, 0]` avec l'ordre donné. 5 couleurs serait utilisées (soit une de plus que pour l'ordre croissant dont le résultat est donné **Q12)**

---

5 Variante de Welsh-Powell

Q15. Écrire une fonction `degre` avec pour argument la liste d'adjacence `LA` d'un graphe quelconque, qui renvoie la liste des degrés des sommets du graphe.
Par exemple, pour un graphe de liste d'adjacence `LA=[[1,2],[0,2,3],[0,1,3],[1,2,4],[3]]`, la fonction `degre` renverra la liste `[2,3,3,3,1]`.

Il faut retourner la liste des longueurs de chacune des sous-liste de `LA`.

```
1 def degre(LA):
2 return [len(voisins) for voisins in LA]
```

Q16. Écrire une fonction `init` avec pour argument un entier `n`, qui renvoie une liste de listes `R` de taille n, telle que `R[i]` soit une liste vide.
Par exemple, `init(3)` renverra `[[],[],[]]`.

```python
def init(n):
    R=[]
    for i in range(n):
        R.append([])
    return R
```

ou

```python
def init(n):
    return [[] for i in range(n)]
```

Q17. Écrire une fonction `ranger` avec pour argument une liste d'adjacence `LA`, qui renvoie une liste `R` de même taille que `LA`, telle que `R[i]` soit la liste des sommets de degré `i`.
Ainsi, pour l'exemple de la question **Q15**, l'appel `ranger(LA)` renverra la liste `[[],[4],[0],[1,2,3],[]]`.

```python
def ranger(LA):
    n=len(LA)
    R=init(n)
    D=degre(LA)
    for s in range(len(LA)): # Noeud s du graphe
        deg=D[s] # Récupération du degré du sommet s
        R[deg].append(s) # Ajout à la sous-liste R[deg] du noeud s
    return R
```

Q18. Écrire une fonction `renverse` avec pour argument une liste `L`, qui crée et renvoie une nouvelle liste obtenue en lisant `L` dans l'ordre inverse.
Par exemple, `renverse([1,2,3,4])` renverra `[4,3,2,1]`.

Savoir renverser une liste est une chose à savoir faire. Nous l'avons vu dès les premières semaines. Le point important est que l'on vous demande une nouvelle liste. Une méthode par extraction est donc tout à fait possible.

```python
def renverse(L):
    return L[::-1] # Dans les faits, on renvoie une nouvelle liste
```

Si l'on vous demande, d'inverser une liste sans en construire une nouvelle. Voici, le code à établir :

```python
def renverse_bis(L):
    for i in range(len(L)//2):
        temp=L[i]
        L[i]=L[len(L)-i-1]
        L[len(L)-i-1]=temp
    # Pas besoin de return, car on ne crée pas nouvelle liste .
```

Q19. Écrire une fonction `trier_sommets` avec pour argument une liste d'adjacence `LA`, qui renvoie la liste des sommets triés dans l'ordre décroissant de leur degré.
Par exemple, pour un graphe de liste d'adjacence `LA=[[1,2],[0,2,3],[0,1,3],[1,2,4],[3]]`, la fonction `trier_sommets` renverra la liste de sommets `[1,2,3,0,4]`.

Il faut d'abord récupérer la liste des sommets de degré `i`. On utilise pour cela, la fonction `ranger`. Cette fonction renvoie la liste des sommets de degré `i` de façon croissante. Il faut alors l'inverser. Ensuite, il suffit d'ajouter chaque sommet de cette liste à la liste solution demandée.

```
def trier_sommets(LA):
    L=ranger(LA) # On récupère la liste des sommets de degré i
    L=renverse(L) # On inverse cette liste, pour obtenir celle des sommets de
        degré i dans l'ordre décroissant
    res=[] # On initialise la list résultat.
    for l in L: # Pour chaque sous liste de L ...
        if l!=[] : # Si la liste l n'est pas vide, alors il y a au moins un
            sommet de degré i
            res+=l # on ajoute alors les éléments de la sous-liste à res.
    return res
```

La fonction `renverse` n'est pas obligatoire.

```
def trier_sommets(LA):
    L=ranger(LA) # On récupère la liste des sommets de degré i
    res=[] # On initialise la list résultat.
    for i in range(len(L)): # Pour chaque sous liste de L ...
        if L[-i-1]!=[] : # Si la liste l n'est pas vide, alors il y a au moins
            un sommet de degré i
            res+=L[-i-1] # on ajoute alors les éléments de la sous-liste à res.
    return res
```

Q20. Pour un graphe à n sommets, quelle est la complexité temporelle de la fonction `trier_sommets` dans le pire des cas ?

Il faut ici étudier la complexité des différentes fonctions impliquées (uniquement la première fonction donnée dans le corrigée de la **Q17**). En prenant $n = len(LA)$:

- `degre` est en $O(n)$;
- `ranger(LA)` est en $O(n+n) = O(n)$ (complexité de `degre` + boucle `for`) ;
- `renverse` est en $O(n)$ également ;
- La boucle `for` de la fonction `trier_sommet` est en $O(n)$.

Donc au final, la fonction `trier_sommet` est donc en $O(3n) = O(n)$.

Q21. Écrire la fonction `colorer3` avec pour argument une liste d'adjacence `LA`, qui crée et renvoie une liste de couleurs `C`, telle que `C[i]` soit la couleur à attribuer au sommet numéro `i`, les sommets étant colorés dans l'ordre décroissant de leur degré.
Quelle est la complexité de `colorer3` dans le pire des cas pour un graphe à n sommets ?

Cette fonction est similaire à `colorer2`, il suffit ici de créer la variable `ordre` grâce à la fonction `trier_sommets`.

```
def colorer3(LA):
    ordre=trier_sommets(LA)
    n=len(LA)
    C=[-1]*n
    for s in ordre:
        colore_sommet(C, s, LA)
    return C
```

La complexité de `colore_sommet` est en $O(n^2)$ dans le pire des cas (les deux boucles `while` sont imbriquées et sont parcourues n fois dans le pire des cas), la complexité de `colorer3` est donc en $O(n^3)$ à cause de `colore_sommet` dans la boucle `for`. La fonction `trier_sommets(LA)` n'a pas de d'impact sur la complexité, puisqu'elle est en $O(n)$.

Q22. Pour le graphe G_{ex} de la FIGURE 2, donner la liste `C` des couleurs renvoyée par la fonction `colorer3`.

```
C=[0, 1, 0, 2, 1, 0, 2, 3]
```

6 Algorithme DSATUR

Q23. Écrire une fonction `degre_satur` avec 3 arguments, une liste d'adjacence `LA`, un sommet `s` du graphe, une liste `C` de couleurs. Cette fonction renvoie le degré de saturation du sommet `s`. On rappelle que le sommet `i` est coloré si et seulement si `C[i]` est différent de -1.

```
def degre_satur(LA,s,C):
    v = LA[s] # Récupération des voisins du sommet s
    couleur_compte = [] # Création d'une liste contenant la couleur des voisins de s
    for i in v: # Pour chaque voisin i de v
        if C[i] != -1 and C[i] not in couleur_compte: # Si le noeud est coloré et que la couleur n'existe pas dans la liste couleur_compte ...
            couleur_compte.append(C[i]) #... on l'ajoute
    return len(couleur_compte)
```

Q24. Écrire une fonction `liste_satur` avec deux arguments, une liste d'adjacence `LA`, la liste `C` des couleurs des sommets, qui renvoie la liste des sommets non colorés du graphe ayant un degré de saturation maximum parmi les sommets non colorés.
On notera qu'il s'agit d'une liste car plusieurs sommets peuvent avoir le même degré de saturation. On supposera de plus qu'il reste au moins un sommet non coloré.

```
def liste_satur(LA,C):
    res=[] # Initialisation d'une liste qui contiendra les sommets encore non colorés.
    maxi=0 # Initialisation de la recherche du maximum. Celui-ci sera forcément plus grand que 0.
    for s in range(len(LA)): # Parcours des sommets du graphe
        if C[s]==-1: # Si le sommet s est non coloré
            degre=degre_satur(LA, s, C) # On calcule son degré de saturation
            if degre==maxi: # Si celui est égal au maxi connu, on l'ajoute à la liste res
                res.append(s)
            elif degre>maxi: # Sinon, si le degré de saturation est plus grand que la maxi connu, on réinitialise une nouvelle liste res
                res=[s]
    return res
```

Q25. Écrire une fonction `pas_fini` avec pour argument une liste `C`, qui renvoie `True` si cette liste contient la valeur -1, `False` sinon.

```
def pas_fini(C):
    for couleur in C:
        if couleur==-1:
        return True
    return False
```

Q26. Compléter, sur le document réponse, la fonction `colorer4` ayant pour argument une liste d'adjacence `LA`, qui renvoie une liste `C` constituant une coloration du graphe. Cette fonction procède de la façon suivante.
Tant qu'il reste un sommet non coloré :

- déterminer parmi les sommets non colorés ceux de degré de saturation maximale ;
- si plusieurs sommets non colorés ont un degré de saturation maximale, en choisir un parmi ceux-ci qui soit de degré maximal ;
- colorer le sommet choisi en lui attribuant la couleur disponible ayant la plus petite valeur.

```
1  def colorer4 (LA):
2      n = len(LA) # nombre de sommets du graphe
3      D = degre(LA) # liste des degrés des sommets du graphe
4      C=[-1]*n # initialisation de la liste des couleurs
5      while pas_fini(C):
6          # liste des sommets non colorés de degré de saturation maximal
7
8          Ls = liste_satur(LA,C)
9
10         # en cas d'égalité, recherche du sommet de degré maximal
11
12         if len(Ls)>1:
13             smax=Ls[0]
14             deg_max=D[Ls[0]]
15             for s in Ls[1:]:
16                 if D[s]>deg_max:
17                     deg_max=D[s]
18                     smax=s
19         else:
20             smax=Ls[0]
21         # coloration du sommet prioritaire
22         colore_sommet(C, smax, LA)
23     return C
```

7 Un minorant du nombre de couleurs nécessaires

Q27. Justifier que le nombre minimum de couleurs pour colorer un graphe est supérieur ou égal au nombre n_c.
Montrer également que : $n_c \leqslant 1 + \max\{\deg s\,,\, s \in G\}$ où deg s désigne le degré du sommet s dans le graphe G.

Pour chaque clique, le nombre de couleurs différentes nécessaire est égale au cardinal de la clique. Si deux sommets sont d'une clique différente, il n'y a aucune raison pour qu'ils aient une couleur différente (sauf si un autre sommet de leur clique l'a déjà prise). Ainsi, le nombre minimum de couleurs nécessaires est bien le cardinal de la plus grande clique.

Dans une clique, le cardinal n de celle ci est égale au nombre de sommets de la clique et le degré de chacun est forcement supérieur ou égal au cardinal de la clique moins un sommet (le sommet considéré). Ainsi, $n - 1 \leq deg\ s$ quelque soit s dans la clique.

Pour la plus grande clique, on peut avoir $n_c - 1 \leq \max\{deg\ s, s \in clique\}$. Ce qui donne $n_c \leq 1 + \max\{deg\ s, s \in clique\}$. Si on a autre sommet $s2$, hors de la plus grande clique, à un degré plus grand, on a alors :

$n_c \leq 1 + \max\{deg\ s, s \in clique\} \leq 1 + deg\ s2$. Ainsi, on peut dire que $n_c \leq 1 + \max\{deg\ s, s \in G\}$.

Q28. Écrire une fonction `est_clique` avec deux arguments, la liste d'adjacence `LA` d'un graphe et un tuple `K` de sommets de ce graphe, qui renvoie `True` si `K` est une clique, `False` sinon.

```
def est_clique(LA,K):
    for s1 in K: # Pour un sommet s1 de K
        for s2 in K: # On va tester si les autres sommets de la clique...
            if s1!=s2:
                if s2 not in LA[s1]: #... ne sont pas voisins
                    return False # S'ils ne sont pas voisins la clique est
                        invalide
    return True
```

Un autre code "plus rapide" (mais de même complexité) est le suivant :

```
def est_clique(LA,K):
    for s1 in range(len(K)):
        for s2 in range(s1+len(K)):
            if K[s2] not in LA[K[s1]]:
                return False
    return True
```

Q29. Sur le document réponse, compléter la fonction `minoration_nb_couleurs` ayant pour argument la liste d'adjacence `LA` d'un graphe, qui renvoie le cardinal de la plus grande clique du graphe considéré.

```
def maxi(L):
    ''' fonction retournant le maximum d'une liste de nombre '''
    maxi=L[0]
    for i in range(1,len(L)) :
        if maxi<L[i]:
            maxi=L[i]
    return maxi
```

```
from itertools import combinations
def minoration_nb_couleurs (LA):
    n = len(LA)# nombre de sommets du graphe
    S = [ k for k in range(n) ] # liste des sommets du graphe
    i = 1+maxi(degre(LA))
    test = True
    while test:
        for K in combinations (S,i):
            if est_clique(LA,K):
                test=False
        i=i-1
    return i+1
```