DS1 - INFORMATIQUE

CORRIGE

Partie I. Des algorithmes pour colorer un graphe

I.1. Introduction sur un exemple

Question 1. La matrice d'adjacence est

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

$M[i][j] = 1$ si une arête relie le sommet i au sommet j. Dans les autres cas $M[i][j] = 0$.

Question 2. La liste d'adjacence du graphe donné en exemple est

```
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] ]
```

Question 3. Un avantage d'une représentation par matrice d'adjacence est que le test d'adjacence est en temps constant (donné par la valeur de $M[i][j]$). Un inconvénient est l'encombrement mémoire (n^2 emplacements mémoires occupés pour un graphe de n sommets). Un autre est que déterminer la liste des voisins nécessite de parcourir une ligne et est en $O(n)$.
Un avantage d'une liste d'adjacence est que déterminer la liste des voisins se fait en temps constant. Un inconvénient est que le test d'adjacence entre deux sommets nécessite de parcourir une liste d'adjacence potentiellement grande.

Question 4.

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

I.2. Tester si une coloration est valide

Question 5.

```
def voisins (i,j,LA) :
        return i in LA[j]
```

Question 6.

```
def coloration_valide(LA,C) :
    n=len(LA) # nombre de sommets
    for i in range(n):
        for j in LA[i]:
            if C[j]==C[i]:
                return False
    return True
```

Question 7. Dans le pire des cas, on fait n passages dans la boucle `for i...`.
Pour un sommet `i` donné, il peut y avoir `n-1` voisins et donc `n-1` comparaisons.
La complexité totale dans le pire des cas est ainsi en $O(n^2)$.

I.3. Un algorithme intuitif de coloration

Question 8.

```
L=[-1 for k in range(n)]
```

ou bien

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

Question 9.

```
def colore_sommet(C,s,LA) :
    n = len(LA)
    coul_vois = [] # couleurs des voisins colorés
     for i in LA[s] :
        if C[i] != -1 and C[i] not in coul_vois :
            coul_vois.append(C[i])
    # coul_vois est maintenant déterminée et on recherche la
    # plus petite couleur possible
    num_coul = 0
    while num_coul in coul_vois :
        num_coul += 1
    C[s] = num_coul
```

Question 10.

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

Question 11.

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

Question 12. L'exécution de `colorer2([7,6,7,5,4,3,2,1,0],LA)` pour le graphe G_{ex} renvoie `[4,2,1,0,3,2,1,0]`. On a donc utilisé 5 couleurs pour colorer de façon valide le graphe G_{ex}.

I.4. Variante de Welsh-Powell

Question 13.

```
def degre(LA):
    D=[len(elt) for elt in LA]
    return D
```

Question 14.

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

Question 15.

```
def ranger(LA):
    n=len(LA)
    D=degre(LA)
    R=init(n)
    for s in range(n):
        R[D[s]].append(s)
    return R
```

Question 16.

```
def renverse(L):
    return L[::-1]
```

Question 17.

```
def trier_sommets(LA):
    R=ranger(LA)
    TS=[]
    for SL in R: # pour chaque sous-liste de R
        TS.extend(SL)
    return renverse(TS)
```

Question 18. Pour une liste de taille `n`, la complexité est en $O(n)$. En effet la fonction `degre` fait un parcours de la liste d'adjacence (taille des listes des voisins), donc complexité en $O(n)$. La fonction `init` est elle même en $O(n)$ ainsi que le remplissage de `R`. Donc la fonction `ranger` est en $O(n)$. Enfin la fonction `trier_sommets` fait un appel à `ranger`, construit la liste `TS` par simple lecture de `R` ($O(n)$), la renverse ($O(n)$).

Question 19.

```
def colorer3(LA):
        ordre=trier_sommets(LA)
        return colorer2(ordre,LA)
```

La fonction `trier_sommets` a une complexité en $O(n)$ (pour le code proposé ci-dessus). La fonction `colorer2` a une complexité en $O(n^2)$. On a au total une complexité en $O(n^2)$.

Question 20. Pour le graphe G_{ex} l'appel à `colorer3` renvoie `C=[0,1,0,2,1,0,3,2]` (mais d'autres possibilités suivant l'ordre de traitement des sommets de même degré).

I.5. Algorithme DSATUR

Question 21.

```
def degre_satur(LA,s,C):
    coul_vois=dict() #couleurs des voisins
    voisins=LA[s]
    for v in voisins:
        if  C[v]!=-1 and C[v] not in coul_vois:
            coul_vois[C[v]]=1
    return len(coul_vois)
```

ou bien

```
def degre_satur(LA,s,C):
    coul_vois=[] #couleurs des voisins
    voisins=LA[s]
    for v in voisins:
        if  C[v]!=-1 and C[v] not in coul_vois:
            coul_vois.append(C[v])
    return len(coul_vois)
```

Il y a une différence de complexité entre ces deux versions.

Question 22.

```
def liste_satur(LA ,C):
    n=len(LA) #nb de sommets du graphe
    s0=0
    while C[s0]!=-1: # on cherche le premier sommet non coloré
        s0+=1
    dsmax=degre_satur(LA ,s0 ,C) # on calcule son degré de saturation
    #Ls sera la liste des sommets non colorés de degré de sat maximum
    Ls=[s0]
    for s in range(s0+1,n):
        if C[s]==-1: # sommet non coloré
            ds=degre_satur(LA ,s,C)
            if ds== dsmax:
                Ls.append(s)
            elif ds >dsmax:
                dsmax=ds
                Ls=[s]
    return Ls
```

Question 23.

```
def pas_fini(C):
    return -1 in C
```

ou bien

```
def pas_fini(C):
    for i in range(len(C)):
        if C[i]==-1:
            return True
    return False
```

Question 24.

```
def colorer4(LA):
    n=len(LA)
    D=degre(LA)
    C=[-1]*n
    while pas_fini(C):
        # liste des sommets non colorés de degré de saturation maximal
        Ls=liste_satur(LA,C)
        # en cas d'égalité, sommet de degré maximal
        s=Ls[0]
        dmax=D[s]
        for  s1 in Ls:
            if D[s1]>dmax:
                s=s1
                dmax=D[s1]
        # coloriage du sommet trouvé
        colore_sommet(C,s,LA)
    return C
```

I.6. Un minorant du nombre de couleurs nécessaires

Question 25. Notons n_c le "nombre de clique" du graphe. Cela veut dire qu'il y a n_c sommets tous reliés les uns aux autres. Fatalement il faut au moins n_c couleurs différentes pour colorer le graphe.
Par ailleurs si le "nombre de clique" est n_c, cela veut dire qu'il y a une clique de n_c sommets. Cela implique qu'au moins un sommet s du graphe est relié à au moins $n_c - 1$ autres sommets. Le degré de ce sommet est donc au moins égal à $n_c - 1$. Il en découle que $n_c - 1 \leq \deg(s)$ d'où $n_c \leq 1 + \deg(s) \leq 1 + \max\{\deg(s), s \in G\}$.

Question 26.

```
def est_clique(LA, C):
    n=len(C)
    for i in range(n):
        for j in range(i+1, n):
                if not voisins(C[i],C[j],LA):
                        return False
    return True
```

Question 27.

```
from itertools import combinations
def minoration_nb_couleurs(LA):
        n = len(LA)      # nombre de sommets du graphe
        S = list(range(n)) # liste des sommets du graphe
        k = 1+max(degre(LA))
        while True:
                for C in combinations(S,k):
                        if est_clique(LA,C):
                                return len(C)
                k=k-1
```