Proposition de corrigé CCINP 2024 Informatique TSI

Partie I - Des algorithmes pour colorier un graphe

I.1 Introduction sur un exemple

Q.1

Pour construire la matrice d'adjacence, on procède ligne par ligne. Par exemple, la première ligne (correspondant au sommet 0), on met 1 zéro à la première colonne (le sommet 0 n'a pas d'arête commune avec le sommet 0), puis un 1 à la deuxième colonne (il y a une arête entre le sommet 0 et le sommet 1), puis 0 (pas d'arête de 0 à 2) et 1 (arête entre 0 et 3). En procédant de même à chaque ligne, on obtient la matrice d'adjacence suivante :

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

Q.2

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]]

Q.3

Un avantage de la matrice d'adjacence est qu'il facile de savoir si 2 sommets i et j sont reliés, il suffit d'avoir l'élément de matrice M_{ij}. L'inconvénient est que cette matrice demande un grand espace mémoire (si il y a n sommets, il faut n^2 coefficients), ce qui peut être inutile si le graphe présente peu d'arêtes (le nombre de coefficients non nuls est égal au double du nombre d'arêtes). Un avantage de la liste d'adjacence est son efficacité pour stocker les informations (seules les arêtes comptent), mais pour savoir si i et j sont reliés, il faut explorer toute la liste LA[i] pour vérifier la présence de j.

Q.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

Q.5

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

Q.6

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

ou pour éviter d'étudier en double chaque arête :

```
def coloration_valide(LA,C) :
  for i in range(len(LA)) :
    for j in range(i+1,len(LA)) :
      if (voisins (i,j, LA) & C[i] == C[j]) : return False
  return True
```

A noter que selon la nature du graphe, la première méthodes peut quand même être plus rapide, la première faisant deux fois plus de test que le nombre d'arêtes, alors que la deuxième vérifie toutes les paires de sommet. Ainsi, si l'on sait que le graphe dont on veut étudier la coloration présente un très faible nombre d'arêtes, la première version est plus rapide.

Q.7

Dans le pire des cas, la coloration est valide et tous les sommets sont reliés entre eux deux à deux. En ce cas, les $n(n-1)/2$ paires de villes sont testées (et même avec le premier programme, chacune est testée deux fois), donc la complexité temporelle est quadratique.

I.3 Un algorithme intuitif de coloration

Q.8

Pour créer une liste C composé de n fois la valeur -1 on fait

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

Q.9

```
def colore_sommet (C,s,LA) :
  coul_vois = []
  for i in LA[s] : coul_vois.append(C[i])
  num_coul = 0
  while num_coul in coul_vois : numcoul += 1
  C[s] = num_coul
```

Q.10

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

Q.11

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

Q.12

On trouve la lise des couleurs suivantes : C = [4,2,1,0,3,2,1,0] et donc 5 couleurs ont été utilisées.

I.4 Variante de Welsh-Powell

Q.13

```
def degre (LA) :
  return [len(i) for i LA]
```

Q.14

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

Q.15

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

Q.16

```
def renverse(L) :
  n = len(L)
  R = []
  for i in range(n) :
    R.append(L[n-1-i])
  return R
```

Q.17

```
def trier_sommets(LA) :
  R = renverse(ranger(LA))
  L = []
  for i in R : L.extend(i)
  return L
```

Q.18

Pour un graphe à n sommets, il faut d'abord appliquer la fonction `ranger`, qui appelle les fonctions `degre` et `init` qui sont en temps linéaires, puis effectuer une boucle n fois. La complexité de `ranger` est donc linéaire en n. Il en est de même pour la fonction `renverse` (dans le pire des cas, tous les sommets ont des degrés différents. Pour `trier_sommets` il y a ainsi un temps linéaire en n puis une boucle sur la liste R renversée, de taille maximale n. On obtient ainsi un algorithme de complexité linaire en n.

Q.19

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

Dans le pire des cas pour un graphe à n sommets, la complexité de `colorer3` est composée de celle de `trier_sommets` qui est linéaire et de `colorer2` qui est à étudier. Pour `colorer2` , il faut effectuer une boucle sur les n sommets de `colore_voisins` qui est dans le pire des cas (tous les sommets reliés entre eux) aussi linéaire en n. La complexité de `colorer2` est donc quadratique et écrase celle de `trier_sommets` : `colorer3` a une complexité quadratique.

Q.20

Dans le cas du graphe de la figure 2, la liste des couleurs renvoyées par `colorer3` est [0, 1, 0, 2, 1, 0, 2, 3].

I.5Algorithme DSATUR

Q.21

```
def degre_satur(LA,s,C) :
  voisins = LA[s]
  liste_couleurs = [-1]
  for i in voisins :
    if C[i] not in liste_couleurs : liste_couleurs.append(C[i])
  return len(liste_couleurs) -1
```

Q.22

```
def liste_satur(LA,C) :
  liste_pascolorie = []
  for s in range(len(LA)) :
    if C[s] == -1 : liste_pascolorie.append(s)
  liste_sommet = []
  deg_max = degre_satur(LA,liste_pascolorie[0],C)
  for s in liste_pascolorie :
    if degre_satur(LA,s,C) > deg_max :
      deg_max = degre_satur(LA,s,C)
      liste_sommet = [s]
    elif degre_satur(LA,s,C) == deg_max : liste_sommet.append(s)
  return liste_sommet
```

Q.23

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

Q.24

Puisque l'énoncé ne précise pas comment choisir le sommet à colorier parmi ceux non coloriés, de saturation maximale et de degré maximal, on peut prendre le premier élément de la liste correspondant et on obtient alors :

```
def colorer4(LA) :
  n = len(LA)
  D = degre(LA)
  C = [-1 for i in range(n)]
  while pas_fini(C) :
    Ls = liste_satur(LA,C)
    i = 0
    for k in range(len(Ls)) :
      if D[Ls[k]] > D[Ls[i]] : i = k
    s = Ls[i]
    colore_sommet(C,s,LA)
  return C
```

I.6 Un minorant du nombre de couleurs nécessaires

Q.25

Pour colorier un graphe de n sommets, le nombre de couleurs est nécessairement supérieur ou égal à celui nécessaire pour colorier chaque clique, et en particulier à celui nécessaire à colorier la plus grande clique de G. Montrons que le nombre de couleurs nécessaire pour colorier la plus grande clique de G est n_c. Tout d'abord, puisqu'il y a n_c sommets dans cette clique, le nombre de couleurs est inférieur ou égal à n_c (il ne peut pas être strictement supérieur, sinon il y aurait plus de couleurs que de sommets). Si le nombre de couleurs utilisées était strictement inférieur à n_c, cela voudrait dire que deux sommets de la clique ont la même couleur, ce qui est impossible puisque par définition ils sont reliés. Ainsi, le nombre de couleurs nécessaire pour colorier la plus grande clique de G est n_c, et donc pour colorier le graphe entier il faut au moins n_c couleurs.

Considérons un sommet s appartenant à la plus grande clique de G de cardinal n_c. Ce sommet s est relié aux $n_c - 1$ autres éléments de la clique donc son degré est supérieur ou égal à $n_c - 1$. Ainsi, le maximum des degrés des sommets du graphe étant supérieur ou égal à celui de s, $\max\{\deg s,\ s \in G\} \geqslant n_c - 1$ d'où l'inégalité demandée.

Q.26

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

Q.27

Je n'ai pas réussi à respecter totalement l'indentation demandée, donc je ne suis pas totalement convaincu de la justesse de cette réponse, mais je propose :

```
def minoration_nb(LA) :
  n = len(LA)
  S = [k for k in range(n)]
  i = 1
  test = True
  while test :
    for K in combinations(S,i) :
      test = False
      if est_clique(LA,K) :
        test = True
        i += 1
```

```
12            break
13        if i == n+1 : test = False
14    i = i-1
15    return i
```

Partie II - Interrogation d'une base de données géographiques

Q.28

```
SELECT code_pays FROM Pays WHERE nom = "France";
```

Q.29

```
SELECT Nom FROM Pays JOIN Inclusion ON Pays.code_pays = Inclusion.
code_pays WHERE continent = 'Europe' ;
```

Q.30

```
SELECT SUM(longeur) FROM Frontieres HAVING code_pays1 = 'F' OR code_pays2
 = 'F' ;
```

Q.31

```
(SELECT Nom FROM Pays JOIN Frontieres ON code_pays = code_pays2 WHERE
code_pays1 ='F') UNION (SELECT Nom FROM Pays JOIN Frontieres ON code_pays
 = code_pays1 WHERE code_pays2 ='F')
```