Sujet

Correction DS07

Graphe

Durée de l'épreuve : **01h55**

Correction

Exercice 1 - Un exemple (5 points)

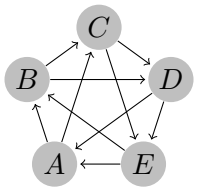

FIGURE 1 – Graphe 1

1. Donner la représentation du graphe 1 par une matrice d'adjacence. *1 pt*

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

2. Donner la représentation du graphe 1 par une liste d'adjacence. *1 pt*

A : [B , C]
B : [C , D]
C : [D , E]
D : [E , A]
E : [A , B]

3. Lister tous les parcours en profondeur sur le graphe 1. *1 pt*

ABCDE ABCED ABDEC ACDEB ACEBD
BCDEA BCDAE BCEAD BDEAC BDACE
CDEAB CDEBA CDABE CEABD CEBDA
DEABC DEACB DEBCA DABCE DACEB
EABCD EABDC EACDB EBCDA EBDAC

4. Lister tous les parcours en largeur sur le graphe 1. *1 pt*

ABCDE ACBDE ACBED
BCDEA BDCAE BDCEA
CDEAB CEDAB CEDBA
DEABC DAEBC DAECB
EABCD EBACD EBADC

5. Lister tous les cycles du graphe 1. *1 pt*

ABCDEA ABCDA ABCEA ABDEA ABDA ACDEA ACDA ACEBDA ACEA
BCDEAB BCDEB BCDAB BCEAB BCEB BDEAB BDEB BDACEB BDAB
CDEABC CDEAC CDEBC CDABC CDAC CEABC CEAC CEBDAC CEBC
DEABCD DEABD DEACD DEBCD DEBD DABCD DABD DACEBD DACD
EABCDE EABCE EABDE EACDE EACE EBCDE EBCE EBDACE EBDE

Exercice 2 - L'implémentation d'un graphe (5 points)

Implémenter une structure de graphe non orienté simple en paradigme objet avec une liste d'adjacence :

```
class Graphe:
    '''classe modélisant un graphe non orienté simple avec une liste d'adjacence'''

    def __init__(self):
        '''Constructeur de la classe'''
        self.sommets = []  # liste des sommets
        self.adjacence = {}   # dictionnaire d'adjacence {sommet:[voisins], ...}

    def ordre(self):
        '''Retourne l'ordre du graphe'''
        # l'ordre du graphe est le nombre de sommets
        return len(self.sommets)

    def taille(self):
        '''Retourne la taille du graphe'''
        # la taille du graphe est le nombre d'arêtes
        return sum([len(self.adjacence[sommet]) for sommet in self.adjacence ]) / 2

    def estVide(self):
        '''Retourne True si le graphe est vide, False sinon'''
        return  self.ordre() == 0

    def ajouteSommet(self, A):
        '''Ajoute, si nécessaire, le sommet A au graphe'''
        if A not in self.sommets:
            self.sommets.append(A)

    def voisins(self, A):
        '''Retourne la liste des voisins du sommet A'''
        return self.adjacence[A] if A in self.adjacence else []

    def sontVoisins(self, A, B):
        '''Retourne True si A et B sont voisins, False sinon'''
        return B in self.voisins(A)

    def aVoisin(self, A):
        '''Retourne True si A a au moins un voisin, False sinon'''
        return A in self.adjacence

    def ajouteArete(self, A, B):
        '''Ajoute l'arête A-B au graphe (ainsi que les sommets éventuellement)'''
        # ajoute les sommets s'ils ne sont pas encore dans la liste des sommets
        self.ajouteSommet(A) ; self.ajouteSommet(B)
        # s'ils ne sont pas déjà voisins
        if not self.sontVoisins(A, B):
            # si A n'a pas encore de voisin, l'ajoute dans le dictionnaire
            if not self.aVoisin(A):
                self.adjacence[A] = []
            # si B n'a pas encore de voisin, l'ajoute dans le dictionnaire
            if not self.aVoisin(B):
                self.adjacence[B] = []
            # Ajoute B en voisin de A et A en voisin de B
            self.adjacence[A].append(B) ; self.adjacence[B].append(A) ;
```


Exercice 3 - L'existence d'un chemin (5 points)

1. Implémenter une méthode de parcours en profondeur minimale (uniquement pour permettre de vérifier l'existence d'un chemin entre deux sommets) :

```
69 def parcoursProfondeur(self, A, visited=None):
70     '''Retourne un parcours en profondeur à partir du sommet A'''
71     # on ajoute A à la liste des sommets visités
72     # (en initialisant la liste uniquement la première fois)
73     if visited == None:
74         visited = [A]
75     else:
76         visited.append(A)
77 
78     # Pour chaque voisin de A
79     for B in self.voisins(A):
80         # s'il n'a pas encore été visité
81         if B not in visited:
82             # on fait appel récursif sur ce sommet:
83             self.parcoursProfondeur(B, visited)
84     return visited
```

2. Implémenter une méthode qui vérifie l'existence d'un chemin entre deux sommets du graphe :

```
87 def existeChemin(self, A, B):
88     '''Retourne True s'il existe un chemin entre A et B'''
89     return B in self.parcoursProfondeur(A)
```

Exercice 4 - La détermination d'un chemin (5 points)

1. Implémenter une méthode de parcours en profondeur (pour permettre cette fois de déterminer un chemin entre deux sommets) :

```
91 def parcoursProfondeurChemin(self, A, visited=None, ancetre=None):
92     '''Retourne un parcours en profondeur à partir du sommet A'''
93     # on ajoute A au dictionnaire des sommets visités  {sommet: ancetre}
94     # (en initialisant la liste uniquement la première fois)
95     if visited == None:
96         visited = {A: None}
97     else:
98         visited[A] = ancetre
99 
100    # Pour chaque voisin de A
101    for B in self.voisins(A):
102        # s'il n'a pas encore été visité
103        if B not in visited:
104            # on fait appel récursif sur ce sommet:
105            self.parcoursProfondeurChemin(B, visited, A)
106    return visited
```


2. Implémenter une méthode qui détermine un chemin entre deux sommets :

```
108 def chemin(self, A, B):
109     '''Retourne un chemin entre A et B s'il en exite un, None sinon'''
110     if not self.existeChemin(A, B):
111         return None
112     visited = self.parcoursProfondeurChemin(A)
113     chemin = [B]
114     ancetre = visited[B]
115     # depuis B, on parcourt le chemin à l'envers jusqu'à A
116     while ancetre != None:
117         chemin.append(ancetre)
118         ancetre = visited[ancetre]
119     # on le remet à l'endroit avant de le retourner
120     return chemin[::-1]
```

```
123 # tests
124
125 # g_1: sommets["A", "B", "C"]  ;  arêtes={"A":["B"], "B":["A"]}
126 g_1 = Graphe()
127 assert g_1.estVide()
128 g_1.ajouteArete("A","B") ; g_1.ajouteSommet("C")
129 assert g_1.ordre() == 3 ; assert g_1.taille() == 1
130 assert not g_1.estVide()
131 assert g_1.aVoisin("A") ; assert g_1.aVoisin("B") ;assert not g_1.aVoisin("C")
132 assert g_1.sontVoisins("A", "B") ; assert not g_1.sontVoisins("A", "C")
133
134 # g_1: ajoute C-D (et donc D)
135 g_1.ajouteArete("C", "D")
136 assert g_1.ordre() == 4 ; assert g_1.taille() == 2
137 assert g_1.aVoisin("A") ; assert g_1.aVoisin("B") ;assert g_1.aVoisin("C")
138 assert g_1.sontVoisins("C", "D") ; assert not g_1.sontVoisins("A", "C")
139
140 # g_1: ajoute B-E et B-F (et donc E et F)
141 g_1.ajouteArete("B", "E") ; g_1.ajouteArete("B", "F")
142
143 # parcours profondeur
144 assert g_1.parcoursProfondeur('A') == ['A', 'B', 'E', 'F']
145 assert g_1.parcoursProfondeur('E') == ['E', 'B', 'A', 'F']
146 assert g_1.parcoursProfondeur('D') == ['D', 'C']
147 assert g_1.existeChemin('A','E') ; assert g_1.existeChemin('F','E')
148 assert not g_1.existeChemin('A','C') ; assert not g_1.existeChemin('F','C')
149
150 # chemin
151 assert g_1.parcoursProfondeurChemin('A') == {'A': None, 'B': 'A', 'E': 'B', 'F': 'B'}
152 assert g_1.parcoursProfondeurChemin('C') == {'C': None, 'D': 'C'}
153 assert g_1.chemin ('A','F') == ['A', 'B', 'F']
154 assert g_1.chemin ('A','D') == None
155 assert g_1.chemin ('E','F') == ['E', 'B', 'F']
156 assert g_1.chemin ('F','E') == ['F', 'B', 'E']
```

