Devoir surveillé d'informatique
2ème année - Janvier 2023

Durée totale : *120mn*
Documents autorisés : *Toutes notes personnelles ou du cours.*

INSA

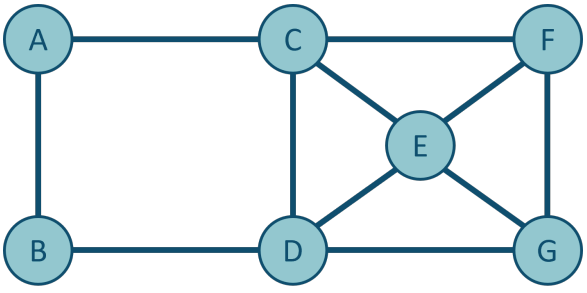
— Le barème est sur 20 points.
— Le sujet est sur 13 pages - il y a 4 exercices.

Consignes :

— Pour l'*Exercice 1. QCM*, ainsi que pour les questions 2.1 et 2.4, écrivez vos réponses directement sur le sujet.
— Les Exercices 1 et 2 du sujet sont à rendre. Pensez à signer le sujet en mettant vos nom, prénom et groupe
— **ATTENTION :** Dans l'*Exercice 1. QCM*, **la pénalité** de **-0.25** points sera attribuée pour toute réponse **fausse**. Cependant, si le total des points de l'*Exercice 1. QCM* est négatif, la note globale de cet exercice sera de **0**.

Exercice 1 QCM (3.25pts + bonus)

(Q1.1) Le graphe G est présenté ci-dessous. Considérez le parcours en largeur (BFS) de ce graphe à partir du sommet A. Pour chacune des séquences suivantes de sommets, dites si elles peuvent représenter un ordre dans lequel BFS parcourt les sommets du graphe.

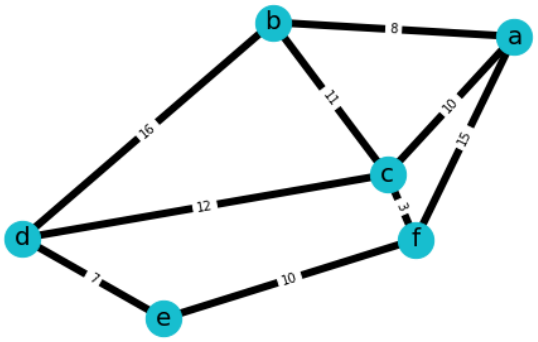


	Vrai	Faux
$ABDECFG$	☐	☐
$ACBFEDG$	☐	☐
$ABCGFED$	☐	☐
$ABCFGED$	☐	☐

Réponse : FVFF
Commentaire : A est un sommet-source (root).
(a) $ABDECFG$: $A \rightarrow B \rightarrow D$ est impossible, car la distance de A à D est 2 quand la distance de A à C est 1, donc C doit être découvert avant D. $\Rightarrow$ Faux
(b) $ACBFEDG$: Cette séquence est tout à fait possible $\Rightarrow$ Vrai

(c) $ABCGFED$: $A \to B \to C \to G$ est impossible, car la distance de A à G est 3, tandis que la distance de A à D ou E ou F est 2. $\Rightarrow$ Faux
(d) $ABCFGED$: $A \to B \to C \to F \to G$ est impossible, car la distance de A à G est 3, tandis que la distance de A à D est 2. $\Rightarrow$ Faux

(Q1.2) Pour le graphe ci-dessous, proposez les plus courts chemins avec leurs coûts (distances) en partant du sommet-source a à tous les autres sommets (sauf b qui est donné comme exemple), en utilisant l'algorithme de Dijkstra.



Sommet	Plus court chemin depuis a	Distance
b	$a \to b$	8
c	$a \to$	
d	$a \to$	
e	$a \to$	
f	$a \to$	

CORRECTION :

Sommet	*Plus court chemin depuis a*	*Distance*
b	$a \to b$	8
c	$a \to c$	10
d	$a \to c \to d$	22
e	$a \to c \to f \to e$	23
f	$a \to c \to f$	13

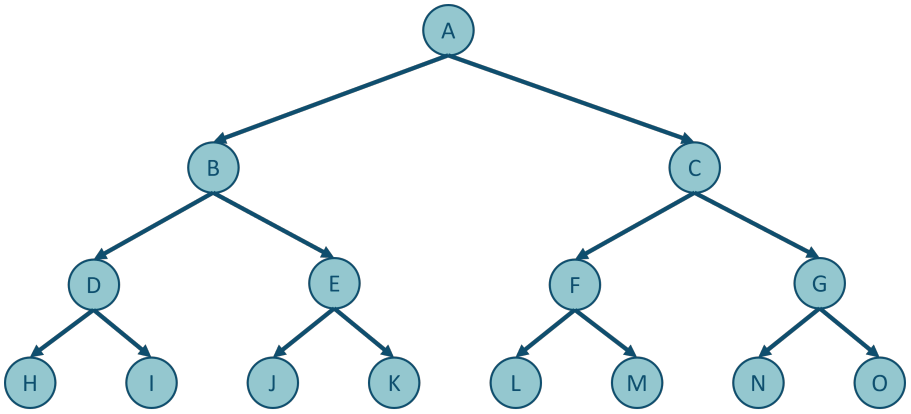
(Q1.3) Considérez le graphe ci-dessous ayant 15 sommets. Soit A un sommet-source (root). Soit X l'ensemble des 4 premiers sommets (y compris le sommet-source) découverts avec le parcours en largeur (BFS) depuis le sommet-source. Soit Y l'ensemble des 4 premiers sommets (y compris le sommet-source) découverts avec le parcours en profondeur (DFS) depuis le sommet-source. (1) Donnez les 4 premiers sommets visités par les parcours BFS et DFS, i.e. X et Y. (2) Quelle est la valeur de $|X - Y|$, où $X - Y$ est une soustraction de X par Y (opération sur les ensembles) et $|\cdot|$ est la cardinalité de l'ensemble $\cdot$ (le nombre d'éléments) ?

Remarque : pour le BFS les sommets adjacents sont considérés par ordre alphabétique ; pour le DFS, nous considérons le parcours de gauche à droite.

Les listes d'adjacences :

```
g = {A : [B, C],
     B : [D, E],
     C : [F, G],
```

```
    D : [H, I],
    E : [J, K],
    F : [L, M],
    G : [N, O],
    H : [],
    I : [],
    J : [],
    K : [],
    L : [],
    M : [],
    N : [],
    O : []}
```



Exemple : si $X = \{1, 2, 3, 4\}$ et $Y = \{6, 5, 1, 7\}$, alors $X - Y = \{2, 3, 4\}$ car le seul élément de X présent en Y est $\{1\}$. Donc, $|X - Y| = |\{2, 3, 4\}| = 3$.

(1) Dans chaque colonne, choisissez les 4 premiers sommets découverts lors de parcours BFS et DFS et mettez les nombres 1-4 correspondant à l'ordre de découverte de sommets par les parcours, i.e. l'ordre de l'ajout des sommets dans les ensembles X et Y :

	BFS (X)	DFS (Y)
A	☐	☐
B	☐	☐
C	☐	☐
D	☐	☐
E	☐	☐
F	☐	☐
G	☐	☐
H	☐	☐
I	☐	☐
J	☐	☐
K	☐	☐
L	☐	☐
M	☐	☐
N	☐	☐
O	☐	☐

(2) Quelle est la valeur de $|X - Y|$? :

☐ 0

☐ 1

☐ 2

☐ 3

☐ 4

CORRECTION :
Réponse : 1
Commentaire : $X = \{A, B, C, D\}$ et $Y = \{A, B, D, H\}$
Donc $X - Y = \{C\}$. Alors $|X - Y| = 1$.

(Q1.4) *(BONUS) Soit $G = (V, E)$ un graphe simple non-orienté et $s \in V$ un sommet-source. Pour tout $v \in V$, nous notons la plus courte distance de s à v sur G par $d(s, v) \geqslant 0$. Un parcours en largeur (BFS) est effectué sur G depuis le sommet s. Soit T un graphe des prédécesseurs (sous-graphe de G) obtenu comme résultat du BFS. Soit $(u, w) \in E$ une arête qui n'est pas présente dans T. Laquelle parmi les valeurs proposées ci-dessous NE peut PAS être la valeur de l'expression $d(s, u) - d(s, w)$?*

Pour répondre à cette question, vous pouvez vous servir du **lemme** suivant :

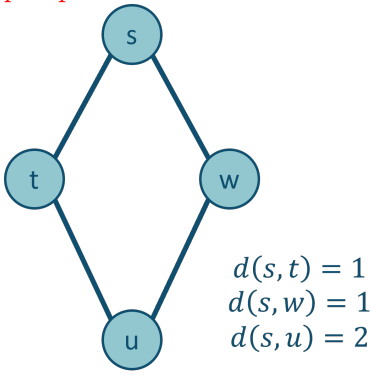
Lemma 1.1 *Soit $G = (V, E)$ un graphe orienté ou non-orienté, et $s \in V$ est un sommet choisi arbitrairement. Alors, pour toute arête $(u, v) \in E$:*

$$d(s, v) \leqslant d(s, u) + 1$$

où $d(s, v)$ est la distance du plus court chemin entre s et v.

- ☐ -1
- ☐ 0
- ☐ 1
- ☐ 2

Réponse : 2
Commentaire :
$d(s, u) - d(s, w) = 0 \rightarrow$ possible
s'il existe un sommet $x \in V$ tel que $(x, u) \in E$ et $(x, w) \in E$ et x est présent dans T, c.à.d. t se situe sur le plus court chemin de s à x et y
$d(s, u) - d(s, w) = 1 \rightarrow$ possible
w et t sont sur le plus court chemin de s à u, et sont du même niveau, i.e. la distance $d(s, w) = d(s, t)$. L'arête (t, u) est retenue, tandis que (w, u) non, c'est pour cela que l'arête (u, w) n'est pas présente dans T. Ex :



$d(s, u) - d(s, w) = -1 \rightarrow$ possible
Pareil au cas précédent, mais en inversant u et w.
$d(s, u) - d(s, w) = 2 \rightarrow$ impossible
Lors du parcours de BFS, soit le sommet u est visité en premier, soit le sommet v. Supposons que ça soit u qui est visité avant. Tous les sommets adjacent de u vont donc être ajouté dans la file de candidats, y compris v qui n'a pas été encore découvert (selon notre supposition). Alors, $d(s, v) = d(s, u) + 1$. Ceci est également vrai si on suppose que v a été découvert avant u.

Exercice 2 Graphes (8.25pts)

Contexte : Nous cherchons à résoudre le problème du câblage par fibre optique entre différents lotissements sur une commune hors agglomération. Plus précisément, il s'agît de trouver une façon de minimiser la longueur totale de câble pour relier (par des rues existantes) tous les lotissements d'une zone à partir d'un des lotissements qui est lui déjà "fibré". On ne considère pas le câblage interne de chaque lotissement, mais seulement le câblage entre lotissements.

Modélisation : Nous modélisons ce problème sous forme d'un graphe non-orienté $G = (V, E)$, où $V = \{0, 1, 2, 3, 4, 5, 6\}$ est un ensemble de sommets numérotés et E est un ensemble d'arêtes. Les lotissements sont les sommets d'un graphe. Les arêtes sont les liaisons directes existantes entre lotissements. Un nombre non-négatif reflétant la distance entre deux points (sommets) est associé à chaque arête. Le sommet correspondant au lotissement déjà "fibré" est numéroté 0.

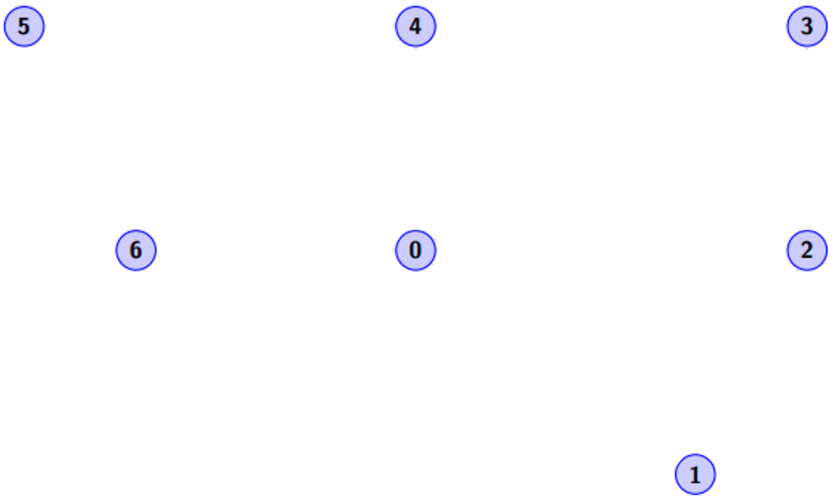
La matrice de distances $D_{|V|\times|V|}$ est donnée ci-dessous :

$$D = \begin{array}{c} \\ \mathbf{0} \\ \mathbf{1} \\ \mathbf{2} \\ \mathbf{3} \\ \mathbf{4} \\ \mathbf{5} \\ \mathbf{6} \end{array} \begin{array}{c} \begin{array}{ccccccc} \mathbf{0} & \mathbf{1} & \mathbf{2} & \mathbf{3} & \mathbf{4} & \mathbf{5} & \mathbf{6} \end{array} \\ \begin{pmatrix} 0 & 30 & 40 & 50 & 20 & 50 & 20 \\ 30 & 0 & 30 & 0 & 50 & 0 & 0 \\ 40 & 30 & 0 & 20 & 50 & 0 & 0 \\ 50 & 0 & 20 & 0 & 40 & 0 & 0 \\ 20 & 50 & 50 & 40 & 0 & 40 & 30 \\ 50 & 0 & 0 & 0 & 40 & 0 & 0 \\ 20 & 0 & 0 & 0 & 30 & 0 & 0 \end{pmatrix} \end{array}$$

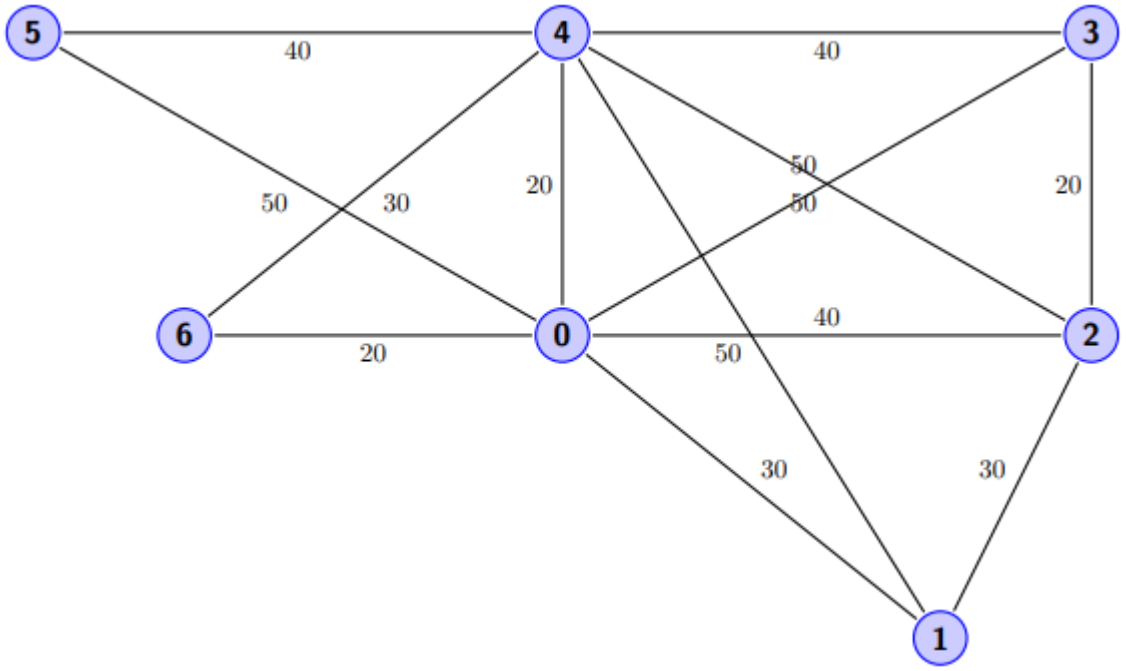
```python
# liste de sommets
v = [0, 1, 2, 3, 4, 5, 6]
# matrice de distance
d = [[0,  30, 40, 50, 20, 50, 20],
     [30, 0,  30, 0,  50, 0,  0],
     [40, 30, 0,  20, 50, 0,  0],
     [50, 0,  20, 0,  40, 0,  0],
     [20, 50, 50, 40, 0,  40, 30],
     [50, 0,  0,  0,  40, 0,  0],
     [20, 0,  0,  0,  30, 0,  0]]
```

(Q2.1) Donnez la représentation graphique du graphe décrit.

Votre réponse :



CORRECTION :



(Q2.2) Nous souhaitons convertir la matrice des distances `d` *en un dictionnaire* `weighted_edges` *ayant comme clés les couples d'arêtes* $(origine, destination)$ *et comme valeurs les poids (distances) non nuls. Écrivez la fonction* `get_weighted_edges(vertices, mat_dist)` *prenant en paramètres la liste de sommets* `vertices` *et la matrice de distances* `mat_dist` *et permettant d'effectuer une telle transformation en renvoyant comme résultat le dictionnaire construit.*

Soit `weighted_edges` un dictionnaire obtenu comme ceci :

```
weighted_edges = get_weighted_edges(v, d)
```

Voici un extrait d'affichage possible de `weighted_edges` :

```
{(0, 1): 30,
 (0, 2): 40,
 (0, 3): 50,
 (0, 4): 20,
```

```
 (0, 5): 50,
 (0, 6): 20,
...}
```

CORRECTION :

```python
def get_weighted_edges(vertices, mat_dist):
    weighted_edges = {}
    for i in vertices:
        for j in vertices:
            if mat_dist[i][j] != 0:
                weighted_edges[(i, j)] = mat_dist[i][j]
    return weighted_edges
```

*(Q2.3) Écrivez la fonction `find_min_edge(some_vertices, weighted_edges)`. Elle prend en paramètres : (1) une liste de **certains** sommets `some_vertices`, (2) un dictionnaire contenant les arêtes pondérées `weighted_edges`. Les clés de `weighted_edges` sont des tuples contenant une paire de sommets formant une arête, les valeurs sont les poids des arêtes. La fonction trouve une arête de poids minimal telle qu'un seul sommet de cette arête soit présent dans la liste `some_vertices` et l'autre ne le soit pas, c'est-à-dire pour une arête (A, B), soit A est présent dans `some_vertices` et B ne l'est pas, ou B est présent dans `some_vertices` et A ne l'est pas. La fonction renvoie l'arête minimale et son poids.*

CORRECTION :

```python
import math
def get_min_edge(some_vertices, weighted_edges):
    min_val = math.inf
    for (v1, v2) in weighted_edges.keys():
        if  (v1 in vertices and v2 not in vertices) or \
            (v2 in vertices and v1 not in vertices):
            if weighted_edges[v1, v2] < min_val:
                min_val = weighted_edges[v1, v2]
                selected_edge = (v1, v2)
    return selected_edge, min_val
```

(Q2.4) Donnez le résultat d'exécution de l'algorithme décrit ci-dessous en partant du sommet 0. Le résultat doit être présenté sous forme graphique.

Algorithme de Prim de recherche d'un arbre couvrant minimal, ACM (en. *Minimum Spanning Tree, MST*) dans un graphe connexe pondéré et non orienté

Un *arbre couvrant minimal* est un arbre (graphe connexe sans les cycles) qui connecte tous les sommets ensemble et dont la somme des poids des arêtes est minimale. Notons que cet arbre couvrant minimal contient tous les sommets du graphe initial et un sous-ensemble de ses arêtes tel que la somme des poids de ces arêtes est minimale.

Principe : L'arbre couvrant minimal est construit depuis un sommet source `root`. Dans le code, `selected_vertices` représente l'ensemble des sommets progressivement ajoutés à l'arbre.

Le dictionnaire contenant des arêtes choisies avec leur poids est stocké dans `selected_edges`. Les clés de `selected_edges` représentent des arêtes, i.e. des tuples de 2 sommets, e.g. (A, B). Les valeurs de `selected_edges` correspondent aux poids de ces arêtes.

Initialisation : `selected_vertices` est initialisé à `root` (ligne 3) et `selected_edges` est vide (ligne 4).

A chaque itération, on cherche une arête (A, B) de poids minimal dont l'un des sommets appartient à `selected_vertices` et l'autre pas. Cette arête est stockée dans `edge_min_weight` et son poids dans `min_weight`. Dans le code ci-dessous le choix de l'arête s'effectue via la fonction `find_min_edge()` (ligne 9) qui prend en paramètre : un ensemble des sommets ajoutés `selected_vertices` et un dictionnaire `weighted_edges` contenant les valeurs de distances attribuées aux arêtes. Cette fonction est implémentée dans la question précédente (Q2.3) et nous considérons le dictionnaire `weighted_edges` obtenu précédemment comme valeur de `weighted_edges`.

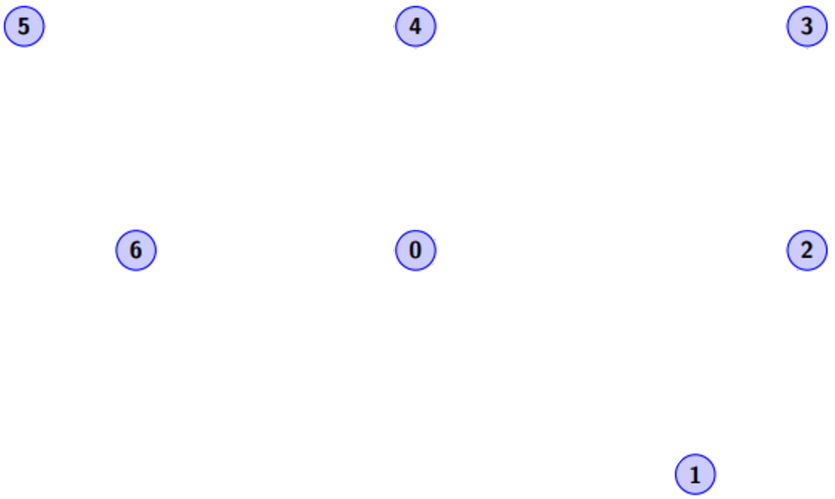
A la fin d'une itération, `selected_vertices` est augmenté du sommet de (A, B) (dans le code `edge_min_weight`) qui n'y était pas (lignes 13-16). `selected_edges` associe à l'arête (A, B) le poids min trouvé (ligne 11).

Le processus est répété tant que tous les sommets du graphe ne sont pas dans `selected_vertices` (ligne 7).

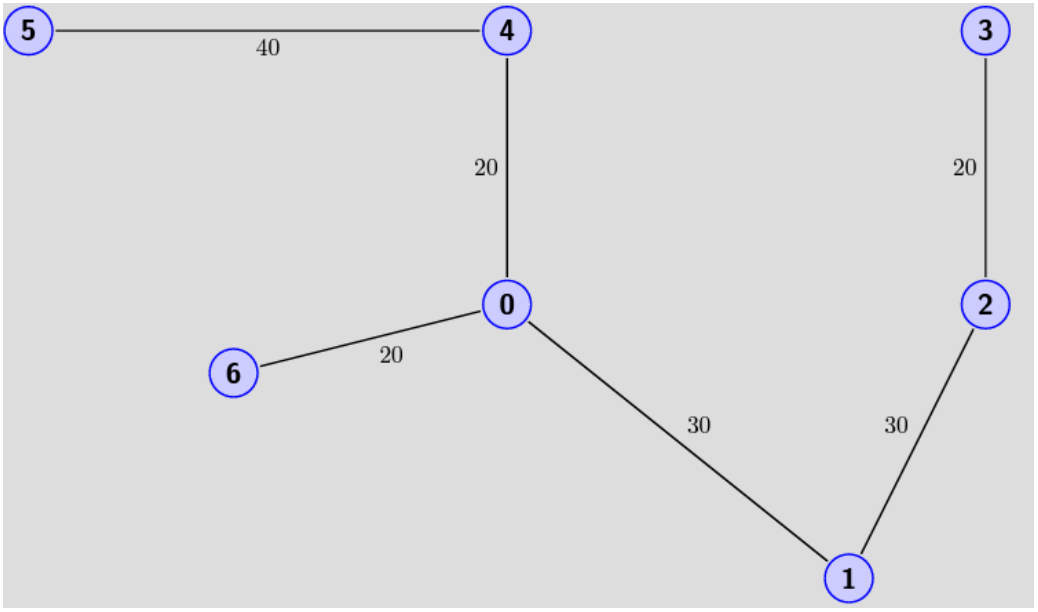
Code ***Python*** *:*

```
1  def prims_algorithm(v, weighted_edges, root):
2      # une liste des sommets deja choisis, initialisee a root
3      selected_vertices = [root]
4      selected_edges = {} # dict des aretes choisis avec leur poids
5
6      # tant que tous les sommets ne sont pas ajoutes dans
       selected_vertices
7      while len(selected_vertices) != len(v):
8          # choisir une arete au poids min
9          edge_min_weight, min_weight = find_min_edge(
       selected_vertices, weighted_edges)
10         # ajouter cette arete dans selected_edges
11         selected_edges[edge_min_weight] = min_weight
12         # ajouter le deuxieme cote de l'arete edge_min_weight dans
       selected_vertices, s'il n'y etais pas
13         if edge_min_weight[0] in selected_vertices:
14             selected_vertices.append(edge_min_weight[1])
15         else:
16             selected_vertices.append(edge_min_weight[0])
17
18     return selected_edges
```

Votre réponse :



CORRECTION :



(Q2.5) Écrivez le code python assurant l'affichage de l'arbre `mst`, obtenu après l'exécution de la ligne suivante :

```
mst = prims_algorithm(v, weighted_edges, 0) # construction de l'ACM
```

Le format du résultat attendu est : `start -> end : poids` où `start` est le sommet de départ de l'arête `(start, end)` dans `mst`, `end` est le sommet d'arrivée de l'arête `(start, end)` dans `mst`, `poids` est le poids (distance) correspondant à l'arête `(start, end)`.

Exemple : `0 -> 1 : 30`

CORRECTION :

```
for (k1, k2), val in mst.items():
    print(f"{k1} -> {k2} : {weighted_edges[(k1, k2)]}")
```

Exercice 3 Manipulation et création de dictionnaires (1,5pt)

On considère le dictionnaire `etudiants` suivant qui associe une note à chaque étudiant.e.

```
etudiants = {"etudiant_1" : 13, "etudiant_2" : 17, "etudiant_3" :
   9,
    "etudiant_4" : 15, "etudiant_5" : 8, "etudiant_6" : 14,
    "etudiant_7" : 16, "etudiant_8" : 12, "etudiant_9" : 13,
    "etudiant_10" : 15, "etudiant_11" : 14, "etudiant_112" : 9,
    "etudiant_13" : 10, "etudiant_14" : 12, "etudiant_15" : 13,
    "etudiant_16" : 7, "etudiant_17" : 12, "etudiant_18" : 15,
    "etudiant_19" : 9, "etudiant_20" : 17}
```

(Q3.1) Proposez un programme Python qui, à partir de ce dictionnaire initial, crée deux nouveaux dictionnaires :

- `etudiantAdmis` dont les clés sont les étudiant.e.s admis.es et les valeurs des clés sont les notes obtenues (note supérieure ou égale à 10),
- `etudiantNonAdmis` dont les clés sont les étudiant.e.s non admis.es et les valeurs des clés sont les notes obtenues (note strictement inférieure à 10).

CORRECTION

```
etudiantAdmis = dict()
etudiantNonAdmis  = dict()
for k,val in etudiants.items():
    if val < 10 :
        etudiantNonAdmis[k] = val
    else :
        etudiantAdmis[k] = val
```

Exercice 4 Manipulation et création de dictionnaires (3pts)

Le dictionnaire `matches` regroupe pour plusieurs clubs de rugby la listes des résultats des derniers matches, avec 'V' pour victoire (4 pts), 'N' pour match nul (2 pts) et 'D' pour défaite (1 pt)

```
matches = {'RCT' : ['V', 'V', 'N', 'D', 'D', 'V'],
        'LOU' : ['N', 'D', 'N', 'V', 'V', 'D', 'V'],
        'ASM' : ['D', 'D', 'N', 'D', 'V'],
        'UBB' : ['V', 'V', 'V', 'V', 'D', 'V'],
        'RM92' : ['V', 'D', 'V', 'D', 'D'] }
```

(Q4.1) Le score d'un club au championnat est donné par la somme des points obtenus dans les matches qu'il a joués. Écrivez un programme qui construit le dictionnaire `scores` dont les clés sont les clubs et les valeurs sont les scores au championnat

CORRECTION

```python
scores = dict()
for k,val in matches.items():
    score = 0
    for v in val:
        if v == 'V':
            score += 4
        elif v == 'N':
            score += 2
        else : score += 1
    scores[k] = score
print(scores)
```

(Q4.2) Comme tous les clubs n'ont pas joué le même nombre de matches, écrivez le programme qui construit le nouveau dictionnaire `average` qui associe à chaque club son score moyen.

CORRECTION

```python
average = dict()
for k,val in scores.items():
    nb = len(matches[k])
    average[k] = val/nb
print(average)
```

Exercice 5 Création d'une classe simple (4pts)

Dans cet exercice, nous allons faire un peu de géométrie 2D.

(Q5.1) Créez la classe `Point` en python, définie ainsi :

```
Attributs:
    - x : abscisse du point
    - y : ordonnée du point
Méthodes:
    - distance_point : retourne la distance euclidienne avec un
   point donné en paramètre
Formatage des affichages (exemple):
    - avec la méthode __str__ : Point (1.0, 2.0)
```

Remarque : la distance euclidienne en 2D entre deux points $P_1(x_1, y_1)$ et $P_2(x_2, y_2)$ se calcule ainsi : $d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$.

Remarque : il n'est pas nécessaire d'écrire l'import de la fonction `sqrt`.

CORRECTION

```python
class Point :
    def __init__(self, xi, yi):
```

```
        self.x = xi
        self.y = yi

    def __str__(self) :
        return f"Point ({self.x}, {self.y})"

    def distance_point(self, p):
        return sqrt((self.x-p.x)**2 + (self.y-p.y)**2)
```

Le barycentre $B(x_b, y_b)$ de coefficient λ de deux points $P_1(x_1, y_1)$ et $P_2(x_2, y_2)$ se définit ainsi :

$$x_b = \lambda x_1 + (1 - \lambda)x_2$$
$$y_b = \lambda y_1 + (1 - \lambda)y_2$$

(Q5.2) Complétez la classe `Point` en ajoutant la méthode `barycentre` qui détermine le barycentre entre le point courant et un point donné en paramètre, avec le coefficient λ donné en paramètre.

CORRECTION

```
    def barycentre(self, p, coeff):
        xb = coeff*self.x + (1-coeff)*p.x
        yb = coeff*self.y + (1-coeff)*p.y
        return Point(xb, yb)
```

De la même manière, on peut déterminer le barycentre $B(x_b, y_b)$ entre plusieurs points P_i :

$$x_b = \frac{\sum_i (\lambda_i x_i)}{\sum_i \lambda_i}$$
$$y_b = \frac{\sum_i (\lambda_i y_i)}{\sum_i \lambda_i}$$

Dans la suite, on se limitera à déterminer l'isobarycentre où $\forall i, \lambda_i = 1$

(Q5.3) Complétez la classe `Point` en ajoutant la méthode `isobarycentre` qui détermine le barycentre entre le point courant et une liste de points donnée en paramètre.

CORRECTION

```
    def isobarycentre(self, l):
        xb = self.x
        yb = self.y
        for p in l:
            xb += p.x
            yb += p.y
        xb = xb / (len(l)+1)
        yb = yb / (len(l)+1)
        return Point(xb, yb)
```