Devoir maison 2019 – version avec exemples

1 Impératifs

Travail à rendre par mail à `marc-michel.corsini@u-bordeaux.fr` en utilisant une adresse officielle de la forme
`prenom.nom@etu.u-bordeaux.fr`, vous indiquerez dans le corps du mail les noms et prénoms des membres du groupe et vous prendrez soin de mettre en copie vos camarades **CC**, afin que tous reçoivent l'accusé de réception du mail. Date limite du rendu le **Lundi 02 décembre à 08h00 (matin)**, tout retard sera assorti d'une pénalité. Pour ce DM vous travaillerez par groupe de 2 **exceptionnellement** 3[1], évidemment la note sera adaptée « plus on est de fous, moins il y a de riz » (Coluche).
Le rendu est constitué d'un document au format **PDF** pour la partie théorique (TdA, axiomes, complexité) et un code **python 3.6+**.

2 Présentation du sujet et des contraintes

Le projet de cette année vise à manipuler un réseau, dont le graphe support sera un graphe non orienté (GNO) muni d'une mesure sur les arêtes à valeur entière dans $-10..+10$, cette mesure est appelée `poids`.
On souhaite pouvoir faire certaines opérations sur cette structure telle que l'ajout de sommets ou d'arêtes. Mais aussi pouvoir accéder à différentes propriétés du graphe support – c'est-à-dire mettre en œuvre les définitions ou prorpiétés vues en cours.

2.1 Travail à effectuer

Pour chaque méthode, j'attends la signature, les axiomes ad-hoc ainsi qu'un calcul de complexité dépendant de **vos** algorithmes où $n = |V|$ représente le cardinal des sommets, $m = |E|$ le cardinal des arêtes.

2.1.1 Services

Il n'y a pas d'attributs publiques ou protégés – c'est-à-dire de valeur modifiable – dans la classe à réaliser, un petit programme (fourni avant le 01.11) **test_dmGNO.py** vérifie l'existence des méthodes demandées et l'absence d'attributs non protégés en écriture. Le nom de la classe et celui des méthodes à réaliser sont imposés.

2.1.2 Cas de méthodes non demandées

Vous avez toute liberté pour rajouter des méthodes que vous pensez utiles pour la manipulation des automates. Vous devrez, dans le cas où ces méthodes sont non privées – donc accessibles à l'utilisateur – fournir : signature, axiomes, complexité et justifier leurs raisons d'être. Pour les méthodes privées, le commentaire doit contenir des informations claires sur ce que fait la méthode.

2.2 Évaluation, remarques

L'évaluation globale du travail sera sur **12** points pour la partie « signatures, axiomes, compléxité » et sur **10** points pour la partie implémentation « codage et tests ». Le code sera impérativement fait avec une approche orientée objets pour le TdA Les tests mettent en évidence la validité de votre code, et permettent de certifier que l'application fournie est conforme à la demande (et à vos axiomes).

[1] Dans ce cas veuillez me contacter par mail **au plus tard le lundi 04 nov. 2019 12h00** afin que je donne mon avis à l'issue du cours du 05 nov.

Il vaut mieux implémenter quelques fonctionnalités de manière adéquate que tout faire n'importe comment.
Si votre code ne tourne pas, l'évaluation est 0 pour la partie code.
Si votre code tourne, mais qu'il n'y a pas de test, l'évaluation du code sera inférieure à 4.
Si votre code tourne, mais pas d'indication sur son utilisation, l'évaluation du code sera inférieure à 4.
Tout cas de tricherie sera durement sanctionné

2.3 Ressources

Le support de cours, et particulièrement la partie « Théorie des Graphes », vous servira de point de départ.

2.4 Complexité

Vous allez vous appuyer sur des structures de données pour construire votre programme.

1. Le dictionnaire **dict()**
   - L'accès à la longueur se fait en temps constant
   - L'ajout et la suppression d'un élément se font en temps constant
2. La liste **list**
   - L'accès à la longueur se fait en temps constant
   - L'ajout (append) se fait en temps constant
   - L'ajout (insert) se fait en temps linéaire
   - La suppression (pop) du 1er et du dernier élément se fait en temps constant, les autres éléments se font en temps linéaire
   - L'accès au ième élément se fait en temps linéaire, si $i \neq 0$ et $i \neq -1$
3. L'ensemble **set**
   - L'accès à la longueur se fait en temps constant
   - Ajout (add) et suppression (discard) se font en temps linéaire

3 Description du TDA

La classe du TDA s'appellera **Reseau**.

3.1 constructeur

Le constructeur `__init__` aura un paramètre qui est une liste de triplets, ayant par défaut la valeur `[ ]`

- Si la liste est vide, le graphe support sera $G = \langle \emptyset, \emptyset \rangle$.
- S'il s'agit d'une liste de triplets, chaque triplet (x, y, w) représente une arête dont les extrémités sont x et y, si $w \in -10..10$ le poids est w, sinon $w = 1$. Il n'y a pas plusieurs arêtes de même poids entre deux extrémités. Si un n-uplet n'est pas de taille 3, il est ignoré sans dommage.

```
g = Reseau([(1,1), (2,1,3), (1,2,3)])
g.edges # [(1, 2, 3)]
g.nbVertices # 2
g.nbEdges # 1
g.dmin # 1
g.dmax # 1
```

3.2 reset

La méthode `reset` **initialise** toutes les variables internes manipulées, les valeurs sont celles d'un graphe vide. Le constructeur est donc un appel à `self.reset()` suivi d'une boucle sur les arêtes, ajoutée une à une après vérification de la syntaxe.

3.3 lecture/écriture dans un fichier

Deux méthodes permettant d'interagir avec un fichier

- `write_to` : écrit le réseau dans un fichier de description. Ce fichier peut contenir des commentaires, tout ce qui se situe après le symbole # est ignoré.

  1. La première ligne, qui n'est pas un commentaire contient 4 entiers
     - Le premier est le nombre de sommets du graphe $|V|$
     - Le second est le nombre d'arêtes du graphe $|E|$
     - Le troisième est le degré minimum du graphe δ_G
     - Le quatrième est le degré maximum du graphe Δ_G
  2. Les lignes suivantes, qui ne sont pas des commentaires contiennent deux ou trois informations numériques
     - Les deux premières sont les extrémités de l'arête et sont dans 1.. $|V|$. La première extrémité doit avoir une valeur inférieure ou égale à la seconde.
     - La troisième est le poids. Un poids de +1 peut être omis.

- `read_from` : lit un fichier de description de réseau, toute erreur de syntaxe est ignorée. En cas d'incohérence avec les informations, le réseau sera vide. Le fichier python **lecture.py** contient un code adapté et testé permettant de lire un fichier texte et de récupérer une liste de triplet (sommet, sommet, poids) respectant la syntaxe. On ne peut pas avoir plusieurs arêtes entre deux sommets ayant le même poids.
  Il suffit donc de récupérer la liste des arêtes et de traiter l'information. **Attention** l'importation depuis un fichier annule toute action préalable.

```
g = Reseau()
g.add_edge(1, 2, -1) # True
g.add_node(42) # True
g.dmin # 0
g.edges # [ (1, 2, -1) ]
g.weight # -1
g.read_from("good_file.txt")
g.edges # [(1, 2, 1), (1, 2, 3), (2, 3, -1), (3, 3, 7)]
g.read_from("bad_file.txt")
g.edges # [ ]
g.nbVertices # 0
```

3.4 Attributs en lecture seule

Un certain nombre de valeurs sont définies par la directive `@property`

- `nbVertices` renvoie le nombre de sommet du graphe
- `nbEdges` renvoie le nombre d'arêtes du graphe
- `dmin` renvoie le degré minimum du graphe
- `dmax` renvoie le degré maximum du graphe
- `edges` renvoie une liste de triplets (x_1, x_2, w) où x_i est une extrémité de l'arête et w son poids. $x_1 \leq x_2$
- `weight` renvoie la somme des poids des arêtes du graphe

3.5 Ajouts

Deux méthodes d'ajout sont à réaliser

1. `add_node` prend en entrée une valeur, renvoie un booléen. Si cette valeur appartient à $\mathbb{N}^*$ et n'est pas déjà utilisée par le graphe, la méthode renvoie `True`. Sinon, la méthode renvoie `False`.
   Attention Seule méthode à faire un contrôle sur le type des paramètres

```
g = Reseau()
g.add_node('a') # False
g.add_node(-2) # False
g.add_node(42) # True
g.add_node(42) # False
```

2. `add_edge` prend en entrée trois paramètres, les deux premières sont les extrémités de l'arête, le troisième, **optionnel**, son poids. La méthode renvoie `True` si l'arête a été ajoutée, `False` sinon. Les 3 paramètres sont des entiers, les 2 premiers sont strictement positifs, le troisième, s'il est fourni, doit être un entier qui devrait être à valeur dans $-10..+10$. Si le troisième paramètre est absent ou hors des valeurs, sa valeur est 1. Entre deux mêmes sommets, il n'y a qu'une seule arête ayant un poids particulier.

```
g = Reseau()
g.add_edge("a", 1, 5) # undef
g.add_edge(1, 3) # True
g.add_edge(3, 1) # False
g.add_edge(3, 1, 1) # False
g.add_edge(3, 1, -2) # True
```

3.6 Suppressions

Plusieurs méthodes de suppression sont disponibles, suivant que l'on souhaite agir sur les sommets ou sur les arêtes.

3.6.1 Suppression des sommets

1. `del_nodes` : supprime tous les sommets et toutes les arêtes du graphe. Ne renvoie rien.
2. `del_node` : s'il existe, supprime le sommet et toutes les arêtes incidentes. Renvoie `True` si la suppression a été effectuée, `False` sinon.

3.6.2 Suppression des arêtes

1. `del_edges` : supprime toutes les arêtes, conserve les sommets. Ne renvoie rien.
2. `del_edge` : prend en entrée 2 sommets et 1 poids (**optionnel**). Si le poids n'est pas fourni, sa valeur est 1. Supprime l'arête dont le poids et les extrémités correspondent. Renvoie `True` si la suppression a pu se faire, `False` sinon.
3. `erase_edge` : prend en entrée 2 sommets. Supprime toutes les arêtes incidentes aux deux sommets. S'il y a eu, au moins une suppression, renvoie `True`, sinon `False`.

3.7 Méthode dépendant d'un sommet

Ces méthodes, prennent en entrée un sommet et renvoie différentes informations.

- `adj` Prend en entrée un sommet et renvoie une liste de sommets qui y sont adjacents. **Attention** chaque sommet n'est donné qu'une seule fois. La liste est triée par ordre croissant sur les sommets.
- `degre` En entrée un sommet, en sortie un entier qui vaut -1 si le sommet n'appartient pas au graphe.
- `composante` Prend en entrée un sommet et renvoie une liste de sommets triés par ordre croissant. La liste des sommets appartenant à la même composante connexe.

3.8 Méthodes sans paramètre

Plusieurs méthodes sans paramètre. Celles qui renvoient un booléen, commencent par `est`.

- `estSimple` renvoie `True`, si le graphe est sans boucle ni arêtes multiples, `False` sinon
- `estConnexe` renvoie `True` s'il n'y a qu'une seule composante connexe, `False` sinon
- `estComplet` renvoie `True` si le graphe est complet
- `estEulerien` renvoie `True` si le graphe est Eulérien
- `estArbre` renvoie `True` si le graphe est un arbre
- `cconnexe` renvoie un dictionnaire, les clefs sont des sommets, la valeur est le sommet de numéro le plus petit appartenant à la même composante connexe.
- `minimal_subtree` renvoie un sous-graphe partiel qui est un arbre de poids minimal. Si le graphe de départ n'est pas connexe renvoie `None`
- `maximal_subtree` renvoie un sous-graphe partiel qui est un arbre de poids maximal. Si le graphe de départ n'est pas connexe renvoie `None`
- `minimisation` renvoie un sous-graphe partiel qui est **simple** et de poids minimal.
- `maximisation` renvoie un sous-graphe partiel qui est **simple** et de poids maximal.
- `matrice_adjacence` renvoie la matrice booléenne du graphe s'il est simple, la matrice entière sinon. La matrice est une liste de listes.
- `matrice_incidence` renvoie la matrice booléenne du graphe s'il est simple, la matrice entière sinon. La matrice est une liste de listes. Les arêtes sont utilisées dans l'ordre de l'attribut `edges`.

3.8.1 Exemple

Il ne s'agit en aucun cas du travail demandé (vérification des axiomes) il s'agit d'un exemple avec le résultat attendu, qui vous permet

```
g = Reseau()
g.add_edge(1, 2, 3) # True
g.add_edge(3, 2 -1) # True
g.add_edge(4, 2) # True
g.add_edge(2, 1, 3)# False
g.nbVertices # 4
g.nbEdges # 3
g.dmin # 1
g.dmax # 3
g.edges # [(1, 2, 3), (2, 3, -1), (2, 4, 1)]
g.weight # 3
g.del_edge(2, 4, -1) # False
g.nbEdges # 3
g.del_edge(2, 4) # True
g.nbEdges # 2
g.add_edge(2, 4) # True
g.add_edge(2, 5) # True
g.add_node(5) # False
g.add_node(7) # True
g.add_edge(2, 3, -5) # True
g.add_edge(2, 3, 3) # True
g.dmin # 0
g.dmax # 6
g.nbVertices # 6
g.nbEdges # 6
g.edges # [(1, 2, 3), (2, 3, 3), (2, 3, -5), (2, 3, -1), (2, 4, 1), (2, 5, 1)]
g.degre(2) # 6
g.add_edge(2, 2, 7) # True
g.degre(2) # 8
h = g.subgraph_vertices([1, 2])
h.edges # [(1, 2, 3), (2, 2, 7)]
g.degre(2) # 8
h.degre(2) # 3
```

```
  p = g.subgraph_edges([(1,2), (2,3), (2,5)])
  p.edges # [(1, 2, 3), (2, 3, 3), (2, 3, -1), (2, 3, -5), (2, 5, 1)]
  p.degre(2) # 5
  p.del_node(2) # True
  p.degre(2) # -1
  p.edges # []
  p.nbVertices # 3
  p.nbEdges # 0
  g.edges
# [(1, 2, 3), (2, 2, 7), (2, 3, 3), (2, 3, -5),
#  (2, 3, -1), (2, 4, 1), (2, 5, 1)]
  g.composante(1) # [1, 2, 3, 4, 5]
  g.adj(1) # [2]
  g.cconnexe() # {1: 1, 2: 1, 3: 1, 4: 1, 5: 1, 7: 7}
  p.cconnexe() # {1: 1, 3: 3, 5: 5}
  p.reset()
  p.nbVertices # 0
  p.nbEdges # 0
  p.dmin # 0
  p.dmax # 0
  p.edges # []
  g.estConnexe() # False
  g.degre(4) # 1
  g.dmin # 0
  g.degre(5) # 1
  g.add_edge(4, 6, 10) # True
  g.add_edge(5, 8, 3) # True
  g.add_edge(6, 7, -7) # True
  g.add_edge(7, 8, -2) # True
  g.nbVertices # 8
  g.nbEdges # 11
  g.dmin # 1
  g.dmax # 8
  g.weight # 13
  g.estSimple() # False
  g.estEulerien() # True
  g.estConnexe() # True
  min_tree = g.minimal_subtree()
  min_tree.weight # -6
  min_tree.nbVertices # 8
  min_tree.nbEdges # 7
  min_tree.estSimple() # True
  min_tree.estEulerien() # False
  min_tree.estConnexe() # True
  max_tree = g.maximal_subtree()
  max_tree.weight # 19
  max_tree.nbVertices # 8
  max_tree.nbEdges # 7
  max_tree.estSimple() # True
  max_tree.estEulerien() # False
  max_tree.estConnexe() # True
  mini = g.minimisation()
  mini.estSimple() # True
  mini.nbVertices # 8
  mini.nbEdges # 8
  mini.weight # 4
  mini.edges
# [(1, 2, 3), (2, 3, -5), (2, 4, 1), (2, 5, 1),
#  (4, 6, 10), (5, 8, 3),  (6, 7, -7), (7, 8, -2)]
  maxi = g.maximisation()
  maxi.estSimple() # True
  maxi.nbVertices # 8
  maxi.nbEdges # 8
  maxi.weight # 12
  maxi.edges
# [(1, 2, 3),  (2, 3, 3), (2, 4, 1),  (2, 5, 1),
#  (4, 6, 10), (5, 8, 3), (6, 7, -7), (7, 8, -2)]
  g.nbVertices # 8
  g.nbEdges # 11
  g.weight # 13
  g.edges
# [(1, 2, 3),  (2, 2, 7), (2, 3, 3),  (2, 3, -5),
#  (2, 3, -1), (2, 4, 1),  (2, 5, 1), (4, 6, 10),
#  (5, 8, 3),  (6, 7, -7), (7, 8, -2)]
  maxi.matrice_adjacence()
# [ [False, True,  False, False, False, False, False, False],
#   [True,  False, True,  True,  True,  False, False, False],
#   [False, True,  False, False, False, False, False, False],
#   [False, True,  False, False, False, True,  False, False],
#   [False, True,  False, False, False, False, False, True],
#   [False, False, False, True,  False, False, True,  False],
#   [False, False, False, False, False, True,  False, True],
#   [False, False, False, False, True,  False, True,  False]]
  g.matrice_adjacence()
# [ [0, 1, 0, 0, 0, 0, 0, 0],
#   [1, 2, 3, 1, 1, 0, 0, 0],
#   [0, 3, 0, 0, 0, 0, 0, 0],
#   [0, 1, 0, 0, 0, 1, 0, 0],
#   [0, 1, 0, 0, 0, 0, 0, 1],
#   [0, 0, 0, 1, 0, 0, 1, 0],
#   [0, 0, 0, 0, 0, 1, 0, 1],
#   [0, 0, 0, 0, 1, 0, 1, 0] ]
  maxi.matrice_incidence()
# [ [True,  False, False, False, False, False, False, False],
```

```
#   [True,  True,  True,  True,  False, False, False, False],
#   [False, True,  False, False, False, False, False, False],
#   [False, False, True,  False, True,  False, False, False],
#   [False, False, False, True,  False, True,  False, False],
#   [False, False, False, False, True,  False, True,  False],
#   [False, False, False, False, False, False, True,  True],
#   [False, False, False, False, False, True,  False, True]]
  g.matrice_incidence()
# [ [1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
#   [1, 2, 1, 1, 1, 1, 0, 0, 0, 0],
#   [0, 0, 1, 1, 1, 0, 0, 0, 0, 0],
#   [0, 0, 0, 0, 0, 1, 0, 1, 0, 0],
#   [0, 0, 0, 0, 0, 0, 1, 0, 1, 0],
#   [0, 0, 0, 0, 0, 0, 0, 1, 0, 1],
#   [0, 0, 0, 0, 0, 0, 0, 0, 1, 1],
#   [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1]]
```