



## Examen d'informatique

1<sup>er</sup> semestre AU : 2022 – 2023

Date : 06 Janvier 2023

Durée : 2H

Nombre de page : 6

*Le langage de programmation sera obligatoirement Python.*

## Ordonnancement de tâches de révision

## Introduction :

Voici un problème important qui se pose à chacun et chacune de nous tous les jours ! J'ai un certain nombre de tâches à exécuter aujourd'hui ; comment planifier à quelle heure chacune va être exécutée ? Par exemple, je peux avoir aujourd'hui à terminer le chapitre 2 en mathématiques, avancer un projet en informatique, aller à la bibliothèque. Chacune de ces tâches va me prendre une certaine durée, que je peux estimer. Même si ce n'est pas tout à fait le cas dans la vie courante, on peut ici supposer que les tâches ne sont pas interrompues ; je ne commence une nouvelle tâche que lorsque la tâche en cours est terminée.

**Dépendances.** Le plus souvent, il existe des dépendances entre les tâches, en ce sens qu'il est nécessaire d'exécuter une tâche A avant d'exécuter une tâche B. Par exemple, il est nécessaire d'aller à la bibliothèque avant de terminer le chapitre 2 en mathématique.

Ces dépendances peuvent être modélisées par un graphe orienté. Les nœuds sont les tâches, les arcs orientés sont les dépendances. Il y a un arc  $1 \rightarrow 2$  si la tâche 1 doit être terminée avant que la tâche 2 puisse commencer.



**Ordonnancement.** Un ordonnancement est la séquence d'exécution des tâches qui respecte cette contrainte de dépendance. La mesure intéressante est alors la durée d'exécution totale, c'est-à-dire la durée écoulée entre l'heure à laquelle l'exécution de la première tâche commence et l'heure à laquelle celle de la dernière tâche se termine.

**Défi.** Les exemples traités dans ce sujet sont bien sûr des illustrations simplifiées. En pratique, on va considérer des graphes de tâches de taille gigantesque, par exemple l'ensemble des actions nécessaires pour assembler un avion. L'objectif sera de trouver le meilleur ordonnancement possible pour l'assemblage selon les contraintes des ouvriers. On peut par exemple embaucher plus d'ouvriers pour réduire la durée d'exécution totale. Cela augmente le coût de production mais réduit les délais de livraison.

Trouver un ordonnancement optimal est alors un défi algorithmique majeur.

## Plan du sujet proposé

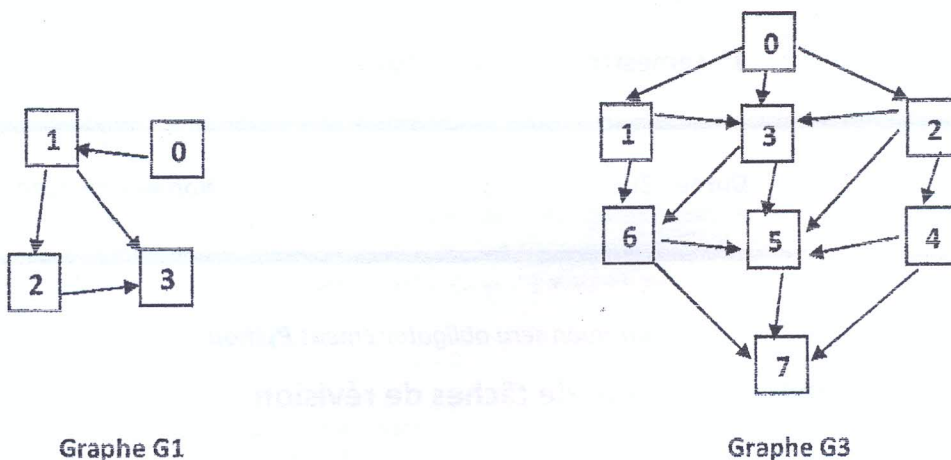
La partie 1 introduit la notion de graphe orienté. La partie 2 s'intéresse aux graphes orientés pondérés. La partie 3 étudie un cas d'ordonnancement de graphe de tâche d'un étudiant pendant la période de révision.

Les parties peuvent être traitées indépendamment. Néanmoins, chaque partie utilise des notations et des fonctions introduites dans les parties précédentes.

## Partie 1. Graphe orienté

Un graphe est une structure de donnée constituée d'un ensemble fini de nœuds notés  $u, v$ , etc., et un ensemble d'arcs dirigés (nommés arrêtes) entre nœuds. L'existence d'un arc entre deux nœuds  $u$  et  $v$  ( $u \rightarrow v$ ) signifie que le nœud  $v$  dépend du nœud  $u$ . En terme de tâches, cela signifie que la tâche  $v$  doit être exécuté après la tâche  $u$ .

La figure 1 suivante propose quelques exemples de graphes orientés :



Travail demandé :

### I. Classe Arrete

#### Question1.

Ecrire une classe **Arrete** dotée de deux attributs u et v, et ajouter son constructeur qui prend en paramètre un tuple t de deux entiers pour initialiser les deux attributs  $u$  et  $v$ .

Exemple :

```
>>> a1 = Arrete( (1,3) )
>>> print(a1.v)
3
```

#### Question2.

Ajouter la méthode **\_\_str\_\_** qui renvoie une chaine de la forme ' $u - v$ '.

Exemple :

```
>>> print(a1)
1-3
```

#### Question3.

Ajouter la méthode **\_\_eq\_\_** qui prend en paramètre un autre objet **Arrete** other. Cette méthode renvoie **True** si les attributs des deux objets sont égaux, **False** sinon.

Exemple :

```
>>> a2 = Arrete( (2,3) )
>>> a1 == a2 # ou bien a1.__eq__(a2)
False
```



**Question 4.** Ecrire une classe Graphe dotée des attributs :

- arretes de type list d'objet d'Arrete
- nœuds de type list d'entiers

Ajouter le constructeur de cette classe qui prend en paramètre une liste de tuple de la forme (u,v).

**Exemple :**

```
>>> G1 = Graphe( [ (1,2) , (1,3) , (2,3) , (0,1) ] )

>>> print ( ' liste de noeuds = ', G1.noeuds )
liste de noeuds = [0, 1, 2, 3]
>>> print('arrete de G1 = ',str(G1.arretes[0]) )
arrete de G1 = 1-2
```

**Question 5.**

Ajouter la méthode \_\_str\_\_ qui renvoie l'affichage du graphe sous la forme d'un tableau A tels que les indices de lignes et de colonnes représentent les nœuds, et s'il existe une arrête entre les nœuds u et v, alors  $A[u, v] = 1$ . (Vous pouvez utiliser les tableaux numpy ou bien les liste de listes).

**Exemple :**

```
>>> print( G1 )
[[0 1 0 0]
 [0 0 1 1]
 [0 0 0 1]
 [0 0 0 0]]
```

**Question 6.**

Ajouter la méthode add\_arrete qui prend en paramètre deux nœuds u et v et qui ajoute une arrête dans le graphe. Cette méthode déclenche une exception si les nœuds u ou v n'existent pas dans le graphe, ou l'arrête existe déjà.

**Exemple :**

```
>>> G1.add_arrete(0 ,3)
>>> print(G1)
[[0 1 0 1]
 [0 0 1 1]
 [0 0 0 1]
 [0 0 0 0]]
```

**Question 7.**

Ajouter la méthode add\_noeud qui prend en paramètre un nœud n et une liste tuple représentant les arrêtes du nouveau nœud n. Cette méthode ajoute le nouveau nœud n ainsi ses arrêtes associées dans le graphe.

**Exemple :**

```
>>> G1.add_noeud(4 , [(2,4) , (4,3)])
>>> print(G1)
[[0 1 0 1 0]
 [0 0 1 1 0]
 [0 0 0 1 1]
 [0 0 0 0 0]
 [0 0 0 1 0]]
```

**Question 8.**

Ajouter la méthode suitant qui prend en paramètre un nœud n et qui renvoie la liste des nœuds adjacents à n.

Un nœud v est un successeur adjacent à un nœud u dans un graphe s'il existe une arrête  $u \rightarrow v$ .

**Exemple :**

```
>>> G1.suitant(1)
[2, 3]
```

## Question 9.

Cette question utilise une structure de donnée **Pile** implémentée dans un module nommé **structure.py**.

La classe Pile dispose d'un attribut **P** de type list et des méthodes suivantes :

- `__init__`
- `est_vide`
- `empiler`
- `depiler`

```
class Pile :
    def __init__(self) :
        self.P = []
    def est_vide(self):
        return len(self.P) == 0
    def empiler(self, s):
        self.P.append(s)
    def depiler(self) :
        assert not self.est_vide()
        return self.P.pop()
```

Ajouter, dans la classe Graphe, une méthode **chemin** qui prend en paramètres un nœud de départ et un nœud d'arrivée. Cette méthode renvoie un **objet Pile** ayant la liste des nœuds du graphe partant du nœud de départ jusqu'au nœud d'arrivée tout en respectant les arrêtes entre nœuds. Cette méthode suit l'algorithme suivant :

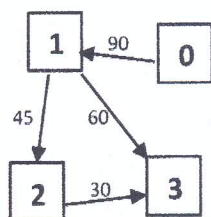
```
Créer une pile vide p
Empiler le nœud de départ dans p
Créer un ensemble vide S pour les nœuds interdits
Tant que la pile n'est pas vide faire
    Dépiler p dans un nœud u
    Chercher les nœuds successeur de u n'appartenant pas à S
    S'il n'y pas de successeurs alors ajouter u dans S
    Sinon
        Choisir un nœud aléatoire v des successeurs de u
        Empiler u dans la pile p
        Empiler v dans la pile p
    Si v égale au nœud d'arrivée alors on arrête et on renvoie la pile
```

## Partie 2. Graphe orienté pondéré

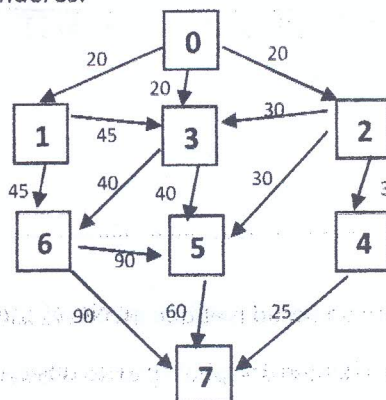
Un graphe est dit pondéré si ses arrêtes sont affectées de nombres nommés poids entre les nœuds.

Le poids d'un chemin dans un graphe pondéré est la somme des poids des arrêtes constituant le chemin.

La figure 2 suivante propose quelques exemples de graphes pondérés.



Graphe G2



Graphe G4

Figure 2. Exemple de graphes orientés et pondérés

### Question 10.

Ecrire une sous classe **GraphePondere** qui hérite de la classe Graphe et dotée en plus d'un attribut **dictG** de type dictionnaire tels que :

- Les clés sont les nœuds du graphe
- Les valeurs sont des dictionnaires tels que les clés sont les successeurs des nœuds clés et les valeurs sont leurs poids associés.

Ajouter le constructeur de cette classe qui prend en paramètre une liste de tuple de la forme (u, v, poids).

Exemple :

```
>>> G2 = GraphePondere( [(1,2,45) , (1,3,60) , (2,3,30), (0,1,90)] )
>>> print(G2.dictG)
{1: {2: 45, 3: 60}, 2: {3: 30}, 0: {1: 90}}
```

### Question 11.

Ajouter la méthode **getPoids** qui prend en paramètre un objet Arrete et qui renvoie le poids entre les deux nœuds de cet objet Arrete dans le graphe pondéré.

Exemple :

```
>>> G2.getPoids( Arrete( (1,3) ) )
60
```

### Question 12.

Surcharger la méthode **\_\_str\_\_** pour avoir un affichage sous la forme d'un tableau de poids

Exemple :

```
>>> print(G2)
[[ 0 90 0 0]
 [ 0 0 45 60]
 [ 0 0 0 30]
 [ 0 0 0 0]]
```

### Question13.

Surcharger la méthode **chemin** pour renvoyer le chemin entre les nœuds de départ et d'arrivée ainsi le poids total du chemin (la somme des poids des arrêtes constituant le chemin trouvé).

Exemple :

```
>>> poids, p2 = G2.chemin(0,3)
>>> print("chemin = ",p2.P)
chemin = [0, 1, 2, 3]
>>> print("poids = ",poids)
poids = 165
```



Un graphe de tâches est un graphe pondéré dont les nœuds sont des tâches et les arrêtes sont les dépendances entre tâches. Le poids d'une arrête entre deux tâches  $u \rightarrow v$  signifie la durée d'exécution de la tâche  $u$  avant de commencer la tâche  $v$ .

L'ordonnancement de tâche est tout simplement chercher un chemin entre la tâche de début et la tâche de fin.

Les tâches sont enregistrées dans un fichier texte, où chaque ligne représente une tâche de la forme :

Numéro\_de\_tâche      désignation      durée      successeur 1      successeur2      ...

Les colonnes sont séparées par une tabulation ('\t').

La première tâche à exécuter est la tâche de la première ligne.

La dernière tâche à exécuter est la tâche de la dernière ligne (pas de successeur).

| Numéro_de_tâche | désignation  | durée | successeur 1 | successeur2 | ... |
|-----------------|--------------|-------|--------------|-------------|-----|
| 0               | aller_biblio | 20    | 1            | 2           | 3   |
| 1               | analys_chap1 | 45    | 3            | 6           |     |
| 2               | inform_chap3 | 30    | 5            | 4           | 3   |
| 3               | physiq_chap2 | 40    | 6            | 5           |     |
| 4               | chimie_chap4 | 25    | 7            |             |     |
| 5               | inform_chap4 | 60    | 7            |             |     |
| 6               | analys_chap2 | 90    | 5            | 7           |     |
| 7               | Fin_revision | 00    |              |             |     |

Un exemple de fichier des tâches modélisé avec le graphe 4 de la figure2

#### Question14.

Ecrire une fonction **charger\_taches** qui prend en paramètre un nom de fichier ayant la liste des tâches à exécuter et qui renvoie un objet **GraphePondere** ayant comme nœuds les numéros des tâches avec leurs arrêtes et leurs poids.

Exemple :

```
>>> G4 = charger_taches('taches.txt')
>>> print(G4)
[[ 0 20 20 20 0 0 0 0]
 [ 0 0 0 45 0 0 45 0]
 [ 0 0 0 30 30 30 0 0]
 [ 0 0 0 0 0 40 40 0]
 [ 0 0 0 0 0 0 0 25]
 [ 0 0 0 0 0 0 0 60]
 [ 0 0 0 0 0 90 0 90]
 [ 0 0 0 0 0 0 0 0]]
```

#### Question15.

Ecrire une fonction **ordonancement** qui prend en paramètre un nom de fichier et renvoie une séquence d'exécution des tâches avec la durée qu'il faut pour bien l'accomplir.

Exemple :

```
>>> duree, chemin = ordonancement('taches.txt')
>>> print("durée d'exécution des tâches = ",duree)
durée d'exécution des tâches = 165

>>> print("séquence des tâches exécutées = ",chemin.P)
séquence des tâches exécutées = [0, 1, 3, 5, 7]
```