DUPART Guillaume
TOUTAIN Yanis

Rapport Projet

Intelligence Artificielle et Jeux

2024 - 2025

LU3IN025

Sommaire

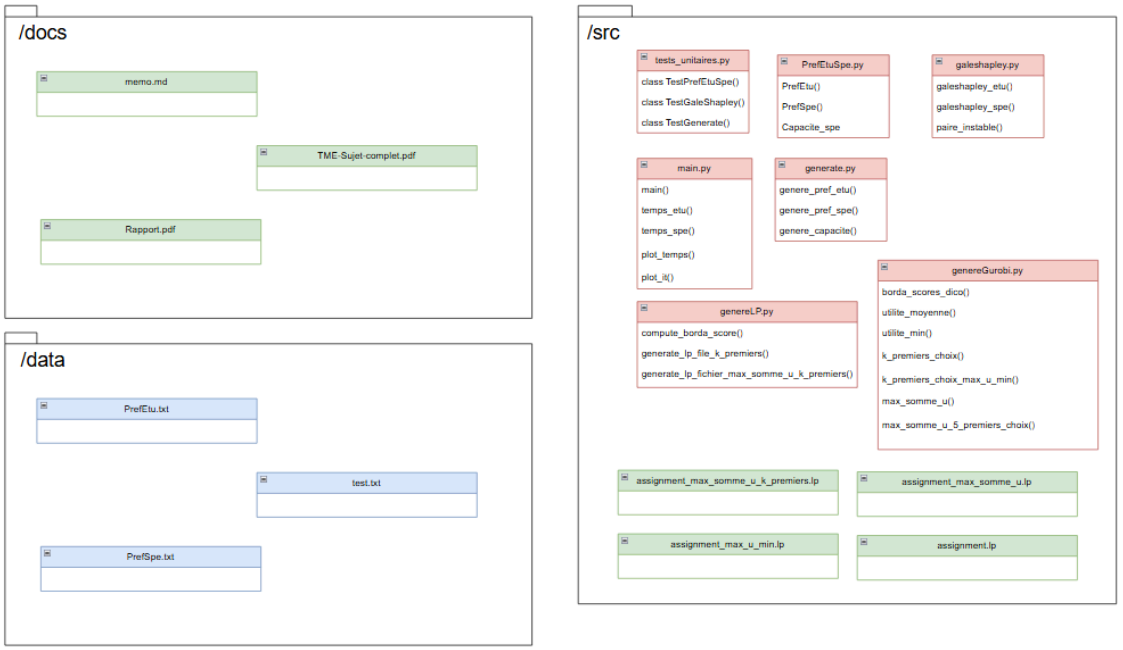
Reformulation du sujet :

Dans ce projet, nous allons explorer l'algorithme de **Gale-Shapley** pour résoudre un problème d'affectation stable. L'idée est d'**optimiser l'affectation des étudiants** aux différents parcours du Master Informatique de Sorbonne Université, en prenant en compte les **préférences** de chacun.

En utilisant **Python**, nous travaillerons à modéliser ces préférences, à implémenter l'algorithme, et à analyser les résultats pour garantir des solutions à la fois **stables et équitables**. Ce sera aussi l'occasion d'explorer des méthodes avancées, comme la programmation linéaire, pour répondre à des enjeux d'équité et de performance.

Ce projet, réalisé en binôme, mêle algorithmes, programmation et réflexion sur des problématiques concrètes, avec un bel équilibre entre théorie et pratique.

Description schématique du projet :



Fichiers comportant des unittest :

Pour **faciliter la compréhension et la correction de notre projet** par notre enseignant, nous avons effectué un **fichier "tests_unitaires.py"** contenant quelques tests de nos **principales fonctions.** (voir description schématique du projet)

Description des structures manipulées :

- **heapq** : Une bibliothèque pour **manipuler des tas** (heaps), qui sont des structures de données basées sur des listes où **l'élément minimal** est toujours à la racine, nous parlons donc ici de "tas min". Elle permet des opérations efficaces comme **l'insertion et l'extraction du minimum** en $\Theta(log(n))$.

- **deque** : Une file double (double-ended queue) fournie par le module **collections**. C'est une structure optimisée pour **ajouter ou retirer des éléments** des deux côtés en $\Theta(1)$, contrairement aux listes classiques, qui sont moins efficaces pour ces opérations.

Description globale du code :

Fichiers :

PrefEtuSpe.py :
Ce fichier regroupe différentes **fonctions de lecture et d'extraction de fichiers texte**, ces fonctions permettent d'organiser les **préférences des étudiants et des spécialités**, ainsi que les **capacités d'accueil des spécialités**.

- PrefEtu(s) : Crée une **matrice des préférences des étudiants** pour les spécialités.
- PrefSpe(s) : Génère **une matrice des préférences des spécialités** pour les étudiants.
- Capacite_spe(s) : Extrait les **capacités d'accueil des spécialités** à partir d'une ligne clé du fichier.

galeshapley.py :
Ce fichier Python implémente des algorithmes pour résoudre l'**algorithme de Gale-Shapley** d'affectation stable entre étudiants et spécialités.

- **galeshapley_etu** applique l'algorithme **côté étudiant**, c'est-à-dire que ce sont les étudiants qui font les demandes.
- **galeshapley_spe** applique l'algorithme de Gale Shapley **côté spécialités**, ce sont les spécialités qui font les demandes
- **paire_instable** regarde si les deux algorithmes fonctionnent bien en essayant de déterminer s' il y a des paires instables. Une paire instable est **une paire d'éléments** qui **se préfèrent** mutuellement alors que **ils sont associés l'un et l'autre à un autre éléments.**

generate.py
Ce fichier python sert à **générer aléatoirement une liste de préférences** de spécialités sur des étudiants et inversement.

- genere_pref_etu génère aléatoirement **n tableaux de préférences de spécialités** avec 9 spécialités (allant de 0 à 8)
- genere_pref_spe génère aléatoirement **9 tableaux de préférences** qui contient n étudiants.
- genere_capacite génère de façon homogène **une liste de capacité**, la fonction prend en paramètre le nombre de spécialités ainsi que le nombre d'étudiants. Elle divise nb_etu par nb_spe afin de pouvoir **équitablement distribués le nombre de place**, si il reste des places non attribués (car le résultat de la division n'était pas entière) alors **les dernières places sont attribués aléatoirement** dans la liste pour ne **pas créer de privilège entre les spécialités**.

main.py

- main sert à **exécuter toutes les fonctions** pour pouvoir avoir une **moyenne de temps selon les itérations**, la fonction regarde aussi si la constante **GRAPH est bien sur True**, si c'est le cas, alors le main appelle la fonction de **génération de graphe**.
- temps_etu execute **10 fois l'algorithme de Gale shapley coté étudiants** sur chacune des valeurs de **n** (allant de **200 à 2000** avec des sauts de 200) puis **calcule le temps**, qu'il enregistre ensuite dans une liste de temps.
- temps_spe execute **10 fois l'algorithme de Gale shapley coté spécialités** sur chacune des valeurs de **n** (allant de **200 à 2000** avec des sauts de 200) puis **calcule le temps**, qu'il enregistre ensuite dans une liste de temps.
- plot_temps sert à **représenter le graphique contenant les deux courbes de temps**. La courbe pour les **étudiants** est représentée en **bleu**, et la courbe en **rouge** est celle pour les **spécialités**.

tests_unitaires.py

- **Ensemble des tests** permettant de vérifier le **bon déroulement des fonctions**. Un print est affiché à chaque fois qu'un test est passé avec succès.

genereLP.py
La fonction generate_lp_file_k_premiers crée un fichier au format LP (Linear Programming) pour résoudre un problème d'affectation d'étudiants à des spécialités. Elle définit une fonction objectif visant à maximiser une constante, permettant ainsi de vérifier la faisabilité de l'affectation. Les contraintes incluent l'obligation pour chaque étudiant d'être affecté à exactement une spécialité parmi ses k premières préférences, tout en respectant les capacités maximales des spécialités. La fonction génère également des variables binaires qui indiquent si un étudiant est affecté à une spécialité spécifique. Le fichier généré est destiné à être utilisé par un solveur d'optimisation pour trouver une solution au problème.

genereGurobi.py

La fonction **borda_scores_dico** calcule les scores de Borda pour chaque étudiant et spécialité. Elle attribue un score basé sur la position des spécialités dans la liste des préférences de chaque étudiant, donnant plus d'importance aux spécialités les mieux placées dans la liste de préférences.

Ensuite, la fonction **utilite_moyenne** évalue l'utilité moyenne d'un arrangement entre les étudiants et les spécialités. Elle prend en compte la position des spécialités dans les préférences des étudiants et calcule la moyenne des scores de Borda obtenus, offrant une mesure de la satisfaction globale du groupe.

De son côté, la fonction **utilite_min** calcule l'utilité minimale dans un arrangement. Cela signifie qu'elle identifie le score le plus bas parmi tous les étudiants en fonction de leur spécialité attribuée, permettant de voir quel arrangement génère la plus grande insatisfaction.

La fonction **k_premiers_choix** est utilisée pour résoudre un problème d'affectation où chaque étudiant est assigné à une spécialité parmi ses *k* premières préférences. En plus de cette contrainte, elle respecte également les capacités des spécialités, cherchant à maximiser le nombre d'étudiants satisfaits tout en respectant les contraintes imposées.

En revanche, **k_premiers_choix_max_u_min** est similaire à la fonction précédente, mais avec un objectif supplémentaire : elle maximise l'utilité minimale des étudiants. Cela garantit que, même dans le pire cas, l'étudiant le moins satisfait ait une utilité aussi élevée que possible, tout en respectant la limite des *k* premiers choix.

La fonction **max_somme_u** vise à maximiser la somme totale des utilités des étudiants et des spécialités, en prenant en compte des scores spécifiques à chaque étudiant et à chaque spécialité. Elle cherche à maximiser la satisfaction globale de tous les étudiants tout en respectant les contraintes de capacité.

Enfin, **max_somme_u_5_premiers_choix** est une version de la fonction précédente, mais elle limite les choix des étudiants à leurs cinq premières préférences. Cela garantit que les affectations tiennent compte des choix les plus importants pour chaque étudiant, tout en maximisant la somme des utilités.

Réponses aux questions :

Question 1

La fonction **PrefEtu(s)** lit un fichier texte contenant des données et organise ces informations dans une **matrice 2D**. Elle commence par **ouvrir le fichier en mode lecture** et **stocke son contenu** dans une liste où chaque élément correspond à une ligne du fichier. Ensuite, **chaque ligne est transformée** en **une liste de mots** ou nombres grâce à la méthode **split().** À partir de la première valeur de la première ligne, la fonction **détermine le nombre d'étudiants** et initialise une matrice de dimensions correspondantes, avec **9 colonnes par étudiant**. Cette matrice est ensuite **remplie avec les valeurs extraites** des lignes du fichier. Enfin, la matrice, qui structure les données en fonction des préférences des étudiants, est retournée.

La fonction **PrefSpe(s)** lit un fichier texte contenant des données et organise ces informations dans une **matrice 2D**. Elle commence par **ouvrir le fichier en mode lecture** et **stocke son contenu** dans une liste où chaque élément correspond à une ligne du fichier. Ensuite, **chaque ligne est transformée** en **une liste de mots** ou nombres grâce à la méthode **split().** À partir de la première valeur de la première ligne, la fonction **détermine le nombre d'étudiants** et initialise une matrice de dimensions **9 x nb_etu**. Cette matrice est ensuite **remplie avec les valeurs extraites** des lignes du fichier. Enfin, la matrice, qui structure les données en fonction des préférences des étudiants, est retournée.

La fonction **Capacite_spe(s)** lit un fichier texte et **extrait les capacités** spécifiées pour différentes spécialités. Elle **ouvre le fichier en mode lecture** et stocke chaque ligne dans une liste appelée contenu. Elle parcourt ensuite ces lignes et **cherche celle qui commence par le mot-clé "Cap"** grâce à la méthode **startswith().** Une fois la ligne trouvée, elle **découpe son contenu** en éléments à l'aide de **split(),** ignore le premier élément, et **convertit les éléments restants en entiers grâce à map()** avant de les **stocker dans une liste appelée capacites.** Cette liste est immédiatement retournée. Si aucune ligne ne commence par "Cap", une liste vide est renvoyée.

Question 2

Nous avons choisi d'utiliser **heapq** et **deque** pour optimiser l'algorithme de **Gale-Shapley**.

Dans **galeshapley_etu**, nous utilisons **heappush et heappop**, ce qui est pratique pour **ajouter ou enlever des éléments d'un tas** et nous permet de **vérifier** assez facilement si le **pire étudiant** déjà affecté est **plus ou moins bien qu'un étudiant qui fait sa candidature** dans la spécialité. La complexité de ces opérations est en $\Theta(log(n))$ ce qui est bien.

D'autre part, les **deque** sont idéales pour gérer **les files des spécialités**. Elles **permettent d'ajouter et de retirer efficacement des éléments** des deux côtés avec une complexité de $\Theta(1)$. Cela permet de **manipuler rapidement les préférences des spécialités**, en retirant l'étudiant de la file et en ajustant les listes sans coût supplémentaire.

Question 3

La fonction **galeshapley_etu** implémente l'algorithme de Gale-Shapley pour l'**affectation des étudiants aux spécialités.** Elle commence par **copier** les listes d'étudiants, de spécialités et de leurs capacités. Elle initialise ensuite une l**iste pour les couples étudiants-spécialité**, des **tas pour gérer les préférences des spécialités**, et une **liste des étudiants libres**. Tant qu'il y a des étudiants libres, **l'algorithme les fait postuler à leur spécialité préférée**. Si la spécialité a des **places disponibles**, l'étudiant est **affecté**. Si la **spécialité est pleine**, l'algorithme **vérifie si l'étudiant actuel est préféré à celui déjà affecté**, et si c'est le cas, il **remplace cet étudiant**. Les **étudiants rejetés sont réintégrés dans la liste des étudiants libres** pour postuler ailleurs. L'algorithme continue jusqu'à ce que tous les étudiants soient affectés à une spécialité.

Complexité de galeshapley_etu

Contexte	Complexité
Copier les listes liste_etu et liste_spe :	$\Theta(n)$
Créer la liste couple_etu_spe :	$\Theta(n)$
Créer la liste de dictionnaires prefSpeIndices :	$\Theta(n^2)$
Créer la liste de tas spe_tas :	$\Theta(n)$
Créer la liste des étudiants libres etu_libre :	$\Theta(n)$
Boucle principale: (while etu_libre) La boucle s'exécute au plus n^2 fois (chaque étudiant peut être traité n^2 fois au pire). Prendre le premier étudiant de etu_libre : $\Theta(1)$ Prendre la liste des spécialités préférées de l'étudiant : $\Theta(1)$ Prendre la première spécialité de la liste : $\Theta(1)$	$\Theta(n * m * log(n))$

Vérifier et mettre à jour les capacités et les couples : $\Theta log(n)$ pour les opérations sur le tas.	
Total	$\Theta(n * m * log(n))$

Où (n) est le nombre d'étudiants

Question 4

La fonction **galeshapley_spe** implémente **l'algorithme de Gale-Shapley** en faisant des spécialités les entités actives. Elle **associe progressivement chaque spécialité à des étudiants** selon leurs préférences mutuelles, tout en respectant les capacités disponibles. Si **une spécialité propose à un étudiant déjà affecté**, celui-ci **compare les deux options** et peut changer pour une spécialité mieux classée selon ses préférences. **Le processus se répète jusqu'à ce que toutes les spécialités soient satisfaites ou n'aient plus d'étudiants à qui proposer**. La fonction garantit une répartition stable entre étudiants et spécialités.

Complexité de galeshapley_spe

Contexte	Complexité
Créer la deque spe_libre :	$\Theta(m)$
Créer les deques spe_preferences :	$\Theta(m^2)$
Créer la liste couple_etu_spe :	$\Theta(n)$
Boucle Principale : (while spe_libre) La boucle while spe_libre s'exécute au plus (m) fois (chaque spécialité est traitée une fois). À chaque itération, les opérations suivantes sont effectuées : Prendre la première spécialité de spe_libre : (O(1)) Prendre le premier étudiant de spe_preferences : (O(1)) Vérifier et mettre à jour les capacités et les couples : (O(n)) pour les opérations de recherche et de mise à jour des listes.	$\Theta(m * n)$
Total	$\Theta(m * n)$

Où (n) est le nombre d'étudiants et (m) est le nombre de spécialités.

Question 5

Lorsque nous **exécutons l'algorithme de Gale Shapley** sur le côté **étudiants**, nous obtenons ce résultat :

[5, 6, 0, 8, 1, 0, 8, 7, 3, 2, 4]

L'étudiant 0 est affecté à la spécialité 5
L'étudiant 1 est affecté à la spécialité 6
L'étudiant 2 est affecté à la spécialité 0
L'étudiant 3 est affecté à la spécialité 8
L'étudiant 4 est affecté à la spécialité 1
L'étudiant 5 est affecté à la spécialité 0
L'étudiant 6 est affecté à la spécialité 8
L'étudiant 7 est affecté à la spécialité 7
L'étudiant 8 est affecté à la spécialité 3
L'étudiant 9 est affecté à la spécialité 2
L'étudiant 10 est affecté à la spécialité 4

Lorsque nous exécutons l'**algorithme de Gale Shapley** sur le côté **spécialités**, nous obtenons ce résultat :

[6, 5, 8, 0, 1, 0, 8, 7, 3, 2, 4]

L'étudiant 0 est affecté à la spécialité 6
L'étudiant 1 est affecté à la spécialité 5
L'étudiant 2 est affecté à la spécialité 8
L'étudiant 3 est affecté à la spécialité 0
L'étudiant 4 est affecté à la spécialité 1
L'étudiant 5 est affecté à la spécialité 0
L'étudiant 6 est affecté à la spécialité 8
L'étudiant 7 est affecté à la spécialité 7
L'étudiant 8 est affecté à la spécialité 3
L'étudiant 9 est affecté à la spécialité 2
L'étudiant 10 est affecté à la spécialité 4

Notre **liste d'association** fonctionne de la manière suivante :

Un **étudiant** correspond à **l'indice dans la liste**, si nous parlons de l'étudiant 0, alors l'indice correspondant sera l'indice 0. **La structure est la même pour les deux algorithmes**, l'étudiant est toujours l'indice, cela évite toute confusion au moment de la création de graphe.

Question 6

La fonction **paire_instable** a pour objectif de **vérifier l'absence de paires instables** dans un ensemble de **couples étudiants/spécialités.** Une paire instable se forme lorsqu'un étudiant et une spécialité préfèrent échanger entre eux plutôt que de rester dans leur affectation actuelle.

Elle prend en entrée trois arguments : couple_etu_spe, qui est une **liste associant chaque étudiant à sa spécialité actuelle**, liste_etu, qui contient les **préférences des étudiants pour les spécialités**, et liste_spe, qui contient les **préférences des spécialités pour les étudiants.**

Le processus commence par **la création d'une liste p_instable**, qui contiendra les paires instables détectées. Ensuite, **pour chaque étudiant** dans couple_etu_spe, **la fonction identifie la spécialité qui lui est associée**. Elle **parcourt les étudiants de cette spécialité** (en fonction de son classement) et **compare l'étudiant courant** à **l'étudiant à la position donnée** dans le classement.

Si l'étudiant courant préfère la spécialité de l'étudiant comparé à sa propre spécialité, et si **la spécialité de l'étudiant comparé préfère l'étudiant courant à l'étudiant qu'elle a actuellement affecté**, cela signifie qu'**une paire instable existe** et elle est ajoutée à la liste p_instable. Enfin, la fonction renvoie cette liste, qui devrait, en théorie, être vide si toutes les paires sont stables. Nous avons testé et la fonction renvoie bien une liste vide donc les paires sont stables.

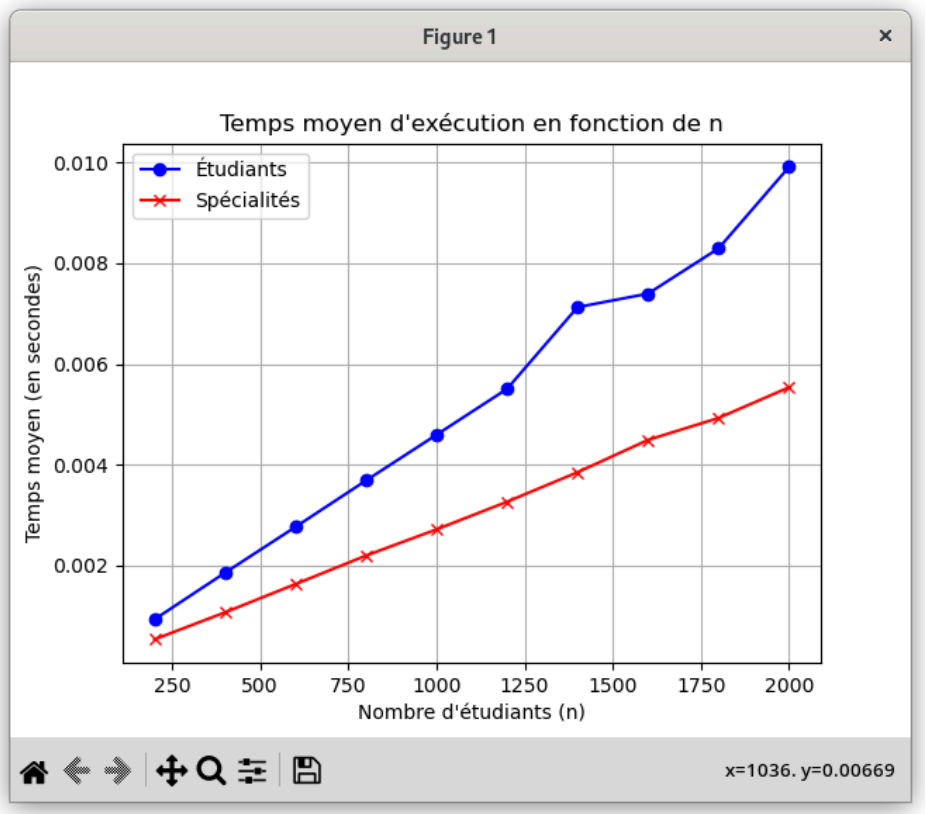
Question 7

La fonction **genere_pref_etu** génère un **dictionnaire où chaque étudiant**, représenté par **une clé, a une liste de spécialités**, mélangée aléatoirement. Elle utilise la fonction **random.shuffle** pour créer cet **ordre aléatoire à partir d'une liste de spécialités** allant de **0 à 8**, puis **assigne cette liste à chaque étudiant**. Le dictionnaire résultant contient donc les préférences aléatoires de chaque étudiant.

La fonction **genere_pref_spe** génère **un dictionnaire où chaque spécialité, représentée par une clé, a une liste d'étudiants** mélangée aléatoirement. Elle utilise **random.shuffle** pour créer un **ordre aléatoire des étudiants** et assigne cette **liste à chaque spécialité**. Le dictionnaire résultant contient **les préférences aléatoires des spécialités pour les étudiants.**

La fonction **genere_capacite** génère une **liste représentant la capacité de chaque spécialité** en fonction du nombre total d'étudiants et du nombre de spécialités. Elle **divise d'abord le nombre total d'étudiants par le nombre de spécialités** pour attribuer une capacité de base à chaque spécialité. Ensuite, **elle répartit le reste des places restantes de manière aléatoire** parmi les spécialités. La liste retournée indique la capacité de chaque spécialité, avec des ajustements aléatoires si nécessaire.

Question 8



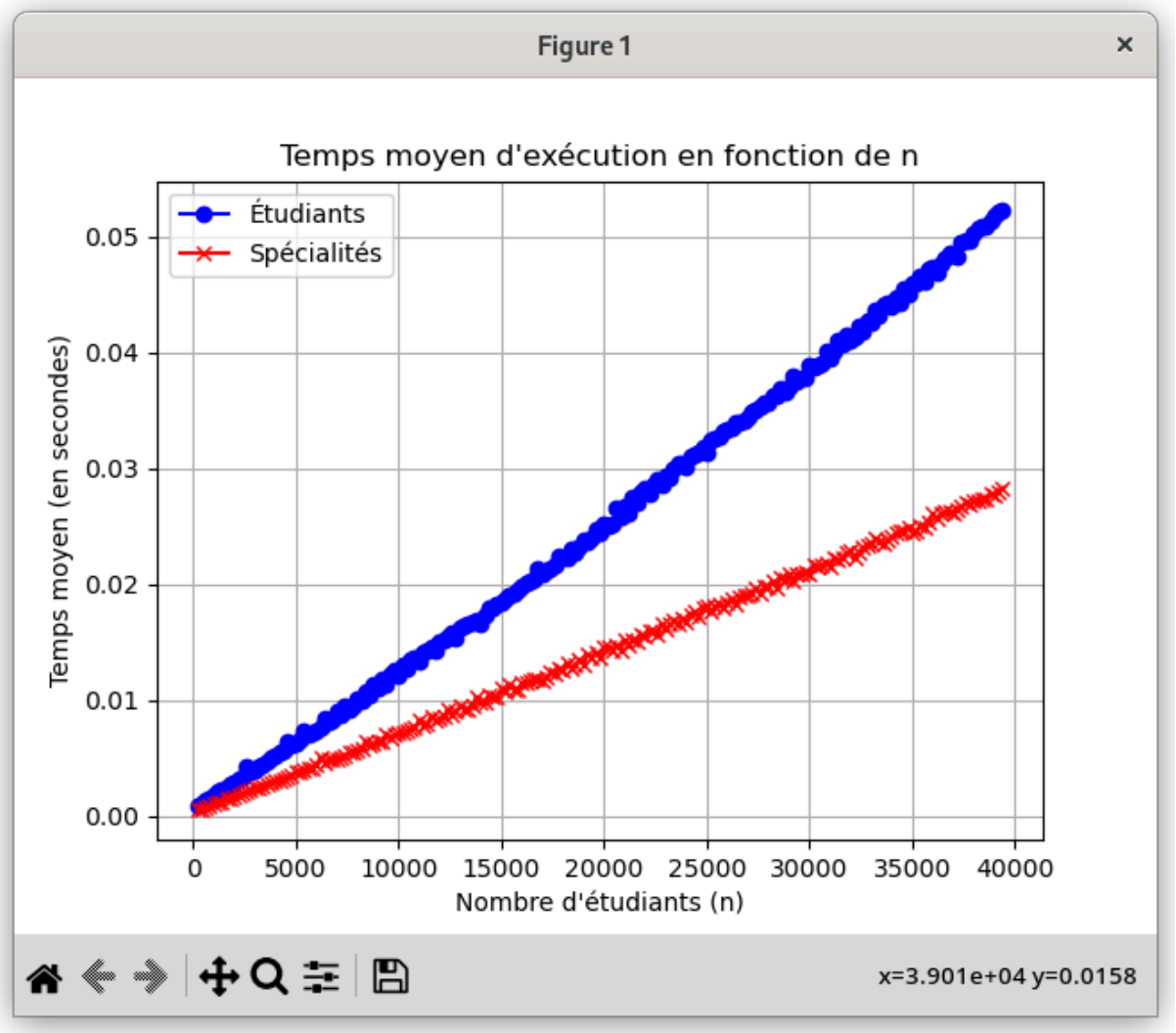
Voici la **courbe du temps de calcul** des deux algorithmes selon le nombre d'étudiants n.

Voici un tableau représentant le temps des algorithme

Nombre d'étudiants	Temps sur etu (en sec)	Temps sur spe (en sec)
200	0.000921630859375	0.000504302978515625
400	0.0018451213836669922	0.0009969949722290039
600	0.002790236473083496	0.0015484094619750977
800	0.0036888360977172852	0.002108263969421387
1000	0.004483079910278321	0.002632617950439453
1200	0.005370306968688965	0.003133845329284668
1400	0.007005763053894043	0.003674960136413574
1600	0.007295083999633789	0.004253196716308594
1800	0.008166074752807617	0.004828190803527832
2000	0.009624147415161132	0.0053375244140625

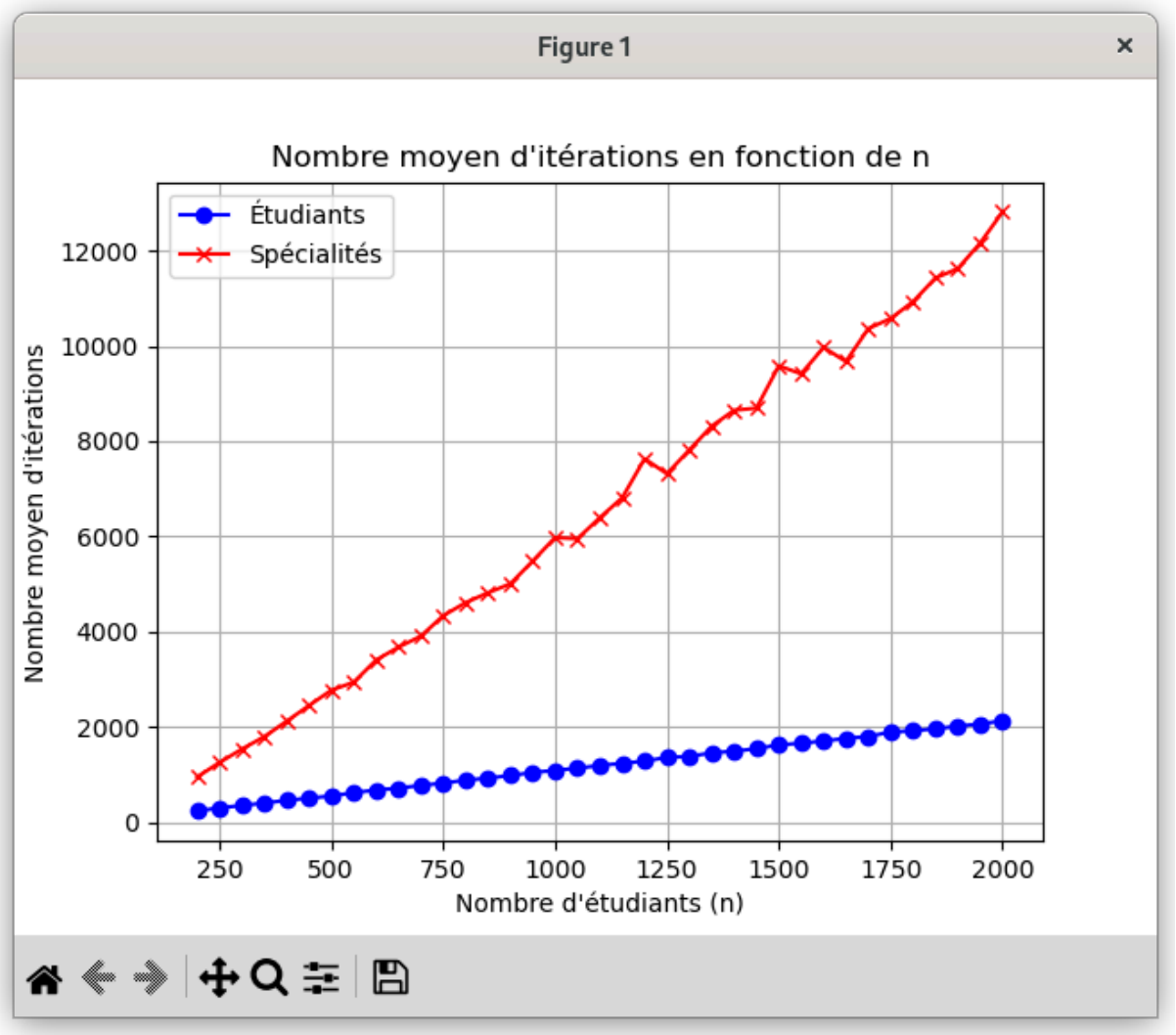
Question 9

Nous avons tracé un graphique jusqu'a n = 10 000 et des pas de 50 et nous trouvons sur ce graphique :



Théoriquement nous avions dit que les complexités étaient $\Theta(n * m * log(n))$ pour Étudiants et $\Theta(m * n)$ pour Spécialités.

Question 10



Voici la courbe des itérations sur nos deux algorithmes. Le nombre d'itérations est assez bas pour Étudiant et plutôt élevé pour Spécialités.

Voici le code associé aux nombre d'itérations. Nous avons par la suite enlevé les modifications apportées aux itérations car elles faussent les graphes de temps.

Question 11 :

Variables : $x_{ij} \in \{0, 1\},\ \forall\, i \in \{1, 2,..., n\},\ \ j \in \{1, 2,..., m\}$ (x_{ij} vaut 1 si l'étudiant i est affecté à la spécialité j, sinon 0.)

Fonction objectif : max $\sum_{i=1}^{n} \sum_{j=1}^{m} x_{ij} * i(s_{ij} \geq\ m\ -\ k)$

où $i(s_{ij} \geq\ m\ -\ k)\ =\ 1$ et $i(s_{ij} <\ m\ -\ k)\ =\ 0$

(s_{ij} est le score de Borda de l'étudiant i pour la spécialité j)

Contraintes :

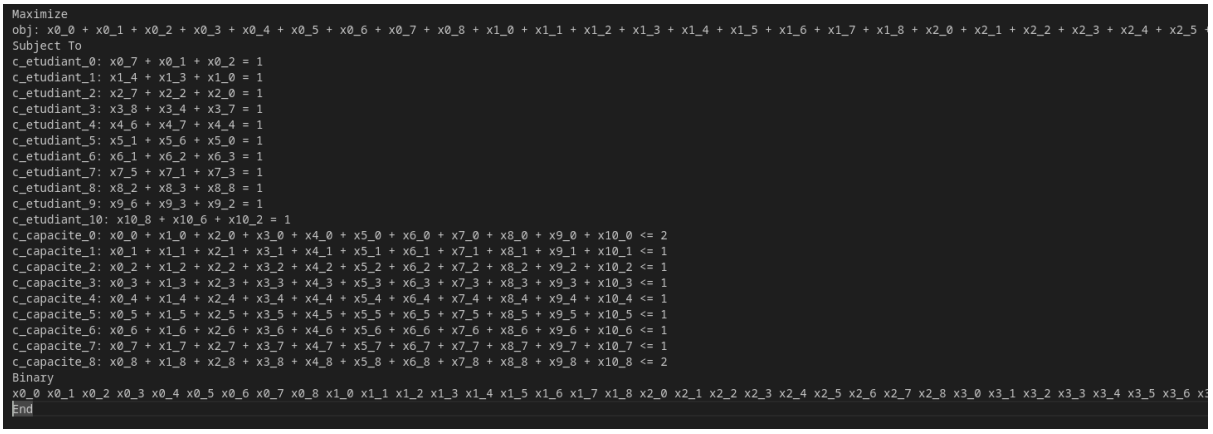
- $\sum_{j=1}^{m} x_{ij}$, $\forall\, i \in \{1, 2,..., n\}$ (Un étudiant i ne peut être affecté qu'à une seule spécialité j)
- $x_{ij}\ =\ 0\ si\ s_{ij} < m\ -\ k,\ \forall\, i \in \{1, 2,..., n\},\ \ j \in \{1, 2,..., m\}$ (On interdit l'affectation d'un étudiant i à une spécialité j si elle n'est pas dans ses k premiers choix.)
- $\sum_{i=1}^{n} x_{ij}\ \leq C_j$, $\forall\, j \in \{1, 2,..., m\}$ (Le nombre d'étudiants affectés à une spécialité j ne dépasse pas sa capacité Cj.)

Question 12

On écrit dans le fichier en respectant les contraintes:

- un étudiant i ne peut être affecté qu'à une seule spécialité j parmi ses k premiers choix
- Le nombre d'étudiants affectés à une spécialité j ne dépasse pas sa capacité Cj.

Contenu du fichier .lp généré :



```
Maximize
obj: x0_0 + x0_1 + x0_2 + x0_3 + x0_4 + x0_5 + x0_6 + x0_7 + x0_8 + x1_0 + x1_1 + x1_2 + x1_3 + x1_4 + x1_5 + x1_6 + x1_7 + x1_8 + x2_0 + x2_1 + x2_2 + x2_3 + x2_4 + x2_5 +
Subject To
c_etudiant_0: x0_7 + x0_1 + x0_2 = 1
c_etudiant_1: x1_4 + x1_3 + x1_0 = 1
c_etudiant_2: x2_7 + x2_2 + x2_0 = 1
c_etudiant_3: x3_8 + x3_4 + x3_7 = 1
c_etudiant_4: x4_6 + x4_7 + x4_4 = 1
c_etudiant_5: x5_1 + x5_6 + x5_0 = 1
c_etudiant_6: x6_1 + x6_2 + x6_3 = 1
c_etudiant_7: x7_5 + x7_1 + x7_3 = 1
c_etudiant_8: x8_2 + x8_3 + x8_8 = 1
c_etudiant_9: x9_6 + x9_3 + x9_2 = 1
c_etudiant_10: x10_8 + x10_6 + x10_2 = 1
c_capacite_0: x0_0 + x1_0 + x2_0 + x3_0 + x4_0 + x5_0 + x6_0 + x7_0 + x8_0 + x9_0 + x10_0 <= 2
c_capacite_1: x0_1 + x1_1 + x2_1 + x3_1 + x4_1 + x5_1 + x6_1 + x7_1 + x8_1 + x9_1 + x10_1 <= 1
c_capacite_2: x0_2 + x1_2 + x2_2 + x3_2 + x4_2 + x5_2 + x6_2 + x7_2 + x8_2 + x9_2 + x10_2 <= 1
c_capacite_3: x0_3 + x1_3 + x2_3 + x3_3 + x4_3 + x5_3 + x6_3 + x7_3 + x8_3 + x9_3 + x10_3 <= 1
c_capacite_4: x0_4 + x1_4 + x2_4 + x3_4 + x4_4 + x5_4 + x6_4 + x7_4 + x8_4 + x9_4 + x10_4 <= 1
c_capacite_5: x0_5 + x1_5 + x2_5 + x3_5 + x4_5 + x5_5 + x6_5 + x7_5 + x8_5 + x9_5 + x10_5 <= 1
c_capacite_6: x0_6 + x1_6 + x2_6 + x3_6 + x4_6 + x5_6 + x6_6 + x7_6 + x8_6 + x9_6 + x10_6 <= 1
c_capacite_7: x0_7 + x1_7 + x2_7 + x3_7 + x4_7 + x5_7 + x6_7 + x7_7 + x8_7 + x9_7 + x10_7 <= 1
c_capacite_8: x0_8 + x1_8 + x2_8 + x3_8 + x4_8 + x5_8 + x6_8 + x7_8 + x8_8 + x9_8 + x10_8 <= 2
Binary
x0_0 x0_1 x0_2 x0_3 x0_4 x0_5 x0_6 x0_7 x0_8 x1_0 x1_1 x1_2 x1_3 x1_4 x1_5 x1_6 x1_7 x1_8 x2_0 x2_1 x2_2 x2_3 x2_4 x2_5 x2_6 x2_7 x2_8 x3_0 x3_1 x3_2 x3_3 x3_4 x3_5 x3_6 x
End
```

Pour résoudre avec Gurobi, à partir de cette question et jusqu'à la fin, on a intégré Gurobi dans les codes Python. Certes, on n'utilise pas les fichiers .lp mais au moins il y a une double vérification sur les résultats.

On utilise Gurobi pour vérifier s'il y a une solution. Gurobi renvoie qu'une solution existe à partir de k=5, qui est :
- {0: 3, 1: 0, 2: 8, 3: 6, 4: 1, 5: 2, 6: 7, 7: 4, 8: 8, 9: 5, 10: 0}

Question 13
Comme dit juste avant, le k minimum pour avoir une solution est k = 5.
Pour maximiser l'unité maximale, on doit prendre le PLNE suivant :

Variables : $x_{ij} \in \{0, 1\},\ \forall\, i \in \{1, 2,..., n\},\ \ j \in \{1, 2,..., m\}$ (x_{ij} vaut 1 si l'étudiant i est affecté à la spécialité j, sinon 0.)

Fonction objectif : max $\sum_{i=1}^{n} \sum_{j=1}^{m} u$
où u est l'utilité minimale parmi celle de tous les étudiants

Contraintes :

- $\sum_{j=1}^{m} x_{ij}\ ,\ \forall\, i \in \{1, 2,..., n\}$ (Un étudiant i ne peut être affecté qu'à une seule spécialité j)
- $\sum_{i=1}^{n} x_{ij} \leq C_j\ ,\ \forall\, j \in \{1, 2,..., m\}$ (Le nombre d'étudiants affectés à une spécialité j ne dépasse pas sa capacité Cj.)
- $\sum_{j=1}^{m} x_{ij} * s_{ij} \geq u\ ,\ \forall\, i \in \{1, 2,..., n\}$

On a avait besoin d'"une fonction pour obtenir les scores de Borda donc là voilà :

```python
def compute_borda_scores(preferences, m):
    """
    Calcule les scores de Borda pour chaque étudiant et spécialité.
    :param preferences: Liste de listes, où chaque liste correspond aux préférences d'un étudiant,
                        avec la spécialité à la position i représentant le ième choix.
    :param m: Nombre total de spécialités
    :return: Liste
    """
    n = len(preferences)  # Nombre d'étudiants
    borda_scores = [[0] * m for _ in range(n)]
    for i, prefs in enumerate(preferences):
        for rank, j in enumerate(prefs):  # Rank = position dans la liste (0 = meilleur choix)
            borda_scores[i][j] = m - rank  # Score de Borda
    return borda_scores
```

De plus, on utilise l'intégration de Gurobi dans Python pour définir la fonction pour maximiser l'unité minimale.

On obtient comme solution : u_min = 0

Question 14

Variables : $x_{ij} \in \{0, 1\},\ \forall\, i \in \{1, 2,..., n\},\ \ j \in \{1, 2,..., m\}$ (x_{ij} vaut 1 si l'étudiant i est affecté à la spécialité j, sinon 0.)

Fonction objectif : max $\sum_{i=1}^{n} \sum_{j=1}^{m} x_{ij} * (u_{ij} + u_{ji})$

où u_{ij} est l'utilité de l'étudiant i pour la spécialité j (ici u_{ij}= s_{ij} qui est le score de Borda de l'étudiant i pour la spécialité j)

et u_{ji} est l'utilité de la spécialité j pour l'étudiant i (ici u_{ji}= s_{ji} qui est le score de Borda de la spécialité j pour l'étudiant i)

On multiplie dans la double somme x par u car on veut la somme des utilités mais seulement celles pour lesquelles les étudiants i ont été affectés à la spécialité j.

Contraintes :

- $\sum_{j=1}^{m} x_{ij}\ ,\ \forall\, i \in \{1, 2,..., n\}$ (Un étudiant i ne peut être affecté qu'à une seule spécialité j)
- $\sum_{i=1}^{n} x_{ij} \leq C_j\ ,\ \forall\, j \in \{1, 2,..., m\}$ (Le nombre d'étudiants affectés à une spécialité j ne dépasse pas sa capacité Cj.)

De plus, on utilise l'intégration de Gurobi dans Python pour définir la fonction pour maximiser la somme des utilités des étudiants et des parcours.
Le résultat obtenu est de 178 avec la configuration : {0: 8, 1: 5, 2: 8, 3: 6, 4: 1, 5: 0, 6: 7, 7: 0, 8: 3, 9: 2, 10: 4}.
À savoir que le maximum obtenable est de 198, 9 pour chaque étudiant(il y en a 11) et 11 pour chaque spécialité (il y en a 9).
L'utilité moyenne obtenue est de 7.545454545454546.
L'utilité minimale obtenue est de 4.

Question 15

Variables : $x_{ij} \in \{0, 1\},\ \forall\, i \in \{1, 2,..., n\},\ \ j \in \{1, 2,..., m\}$ (x_{ij} vaut 1 si l'étudiant i est affecté à la spécialité j, sinon 0.)

Fonction objectif : max $\sum_{i=1}^{n} \sum_{j=1}^{m} x_{ij} * (u_{ij} + u_{ji})$

où u_{ij} est l'utilité de l'étudiant i pour la spécialité j (ici u_{ij}= s_{ij} qui est le score de Borda de l'étudiant i pour la spécialité j)

et u_{ji} est l'utilité de la spécialité j pour l'étudiant i (ici u_{ji}= s_{ji} qui est le score de Borda de la spécialité j pour l'étudiant i)

On multiplie dans la double somme x par u car on veut la somme des utilités mais seulement celles pour lesquelles les étudiants i ont été affectés à la spécialité j

Contraintes :

- $\sum_{j=1}^{m} x_{ij}$, $\forall i \in \{1, 2,..., n\}$ (Un étudiant i ne peut être affecté qu'à une seule spécialité j)
- $x_{ij} = 0 \; si \; s_{ij} < m - k$, $\forall i \in \{1, 2,..., n\}$, $j \in \{1, 2,..., m\}$ (On interdit l'affectation d'un étudiant i à une spécialité j si elle n'est pas dans ses k premiers choix.)
- $\sum_{i=1}^{n} x_{ij} \leq C_j$, $\forall j \in \{1, 2,..., m\}$ (Le nombre d'étudiants affectés à une spécialité j ne dépasse pas sa capacité Cj.)

De plus, on utilise l'intégration de Gurobi dans Python pour définir la fonction pour maximiser la somme des utilités des étudiants et des parcours parmi les solutions où chaque étudiant a un de ses k*, soit 5 premiers choix.
Le résultat obtenu est de 176 avec la configuration : {0: 8, 1: 5, 2: 8, 3: 6, 4: 1, 5: 0, 6: 7, 7: 0, 8: 2, 9: 3, 10: 4}.

Question 16

Nous avons fait des fonctions qui calculent l'utilité moyenne et l'utilité minimale pour un résultat sous forme de dictionnaires.

Solutions obtenues	Stabilité	Utilité moyenne(/10)	Utilité minimale des étudiants(/10)	Nombre de paires instables
GS étu	stable	7.636363636363637	4	0
GS spé	stable	7.181818181818182	4	0
Q13	instable	6.818181818181818	5	8 -> [(0, 5), (0, 6), (1, 5), (1, 6), (7, 0), (7, 7), (9, 2), (10, 4)]
Q14	stable	7.545454545454546	4	0
Q15	stable	7.363636363636363	5	0