Projet 1 : Affectations masters

LU3IN025

Culevski Mattias & Doel Mumbobi Ndoluvwalu

Sommaire

I – Introduction
1) Présentation du sujet 3
2) Structures utilisées 3

II – Modélisation et fonctions
1) Objectif 4
2) Mise en place 4

III – Analyse des résultats théoriques
1) Objectif 6
2) Mise en place 6

IV – Equité et modélisation à l'aide de PLNE
1) Objectif 9
2) PLNE 9
3) Comparaisons 9

I – Introduction

1) Présentation du sujet

Le sujet de ce projet se porte sur un problème d'affectation d'étudiants dans les différents masters d'informatique de Sorbonne Université, qui comporte au total 9 parcours. On souhaite affecter au mieux possible tous les étudiants (tout d'abord 11 étudiants, puis dans la suite un nombre n aléatoire d'étudiants) aux 9 parcours.
Pour cela, on appliquera principalement un algorithme vu en cours, celui de Gale-Shapley des deux côtés : c'est-à-dire en fonction des préférences des étudiants, mais aussi en fonction des préférences des parcours. Afin d'optimiser au mieux possible l'application de l'algorithme sur nos exemples, on utilise des structures de données variées en Python.

2) Structures utilisées

Afin d'optimiser au mieux les opérations effectuées lors de nos tests, nous avons décidé d'utiliser les structures de données suivantes :

Listes et matrices (listes de listes) : il est demandé de retourner dans les fonctions de lecture des matrices de préférences. On utilise également une liste simple pour les capacités des différents parcours.
Ensembles (set) : pour éviter des répétitions dans certains calculs, on utilise des ensembles qui ne peuvent donc contenir qu'une seule fois une valeur donnée, ce qui est donc utile dans des affectations à des parcours.
Dictionnaires : afin de réduire le temps de certains calculs, on utilise des dictionnaires couplés d'ensembles (dict->set) pour retrouver et changer des affectations rapidement en fonction des valeurs dont on dispose.
Les différentes complexités des structures de données et des calculs seront évoqués plus tard faisant l'objet de différentes questions du sujet.

II – Modélisation et fonctions

1) Objectif

On souhaite appliquer l'algorithme de Gale-Shapley, qui est un algorithme qui résout le problème des mariages stables. Ceci est fait de 2 façons : tout d'abord du « côté étudiant », et ensuite du « côté parcours ». On applique ensuite nos deux fonctions à des fichiers textes donnés, contenant des préférences des étudiants, des parcours, mais aussi les capacités des parcours auquel on a préalablement appliqué des fonctions de lecture pour y extraire les données importantes.

2) Mise en place

Les fonctions **lectureEtu** et **lectureSpe** vont prendre en entrée un fichier texte, et retourner les matrices cE et cP tels que cE qui en ligne i contient les classements des parcours selon les préférences de l'étudiant i, et inversement avec cP.
Ensuite, la fonction **GaleShapleyEtu** et **GaleShapleyPrc** prend en entrée les listes de préférences des étudiants, parcours, et les capacités des parcours. On effectue 5 opérations majeures dans cette fonction comme décrit dans le sujet, dont voici les structures de données utilisées et leurs complexités pour que ces opérations soient les plus rapides et les moins coûteuses en espace mémoire :
1. Trouver un étudiant libre à chaque itération.
Structure utilisée : **etu_libres** (ensemble (set))**.** Complexité : O(1) car l'utilisation de pop() sur un set est constant.

2. Étant donné un étudiant libre, trouver le prochain parcours à qui faire une proposition.
Structures utilisées : **etu_pref** (liste de listes) et **propositions** (ensemble (set)). Complexité : O(m) dans le pire cas, si l'étudiant a déjà fait des propositions à tous les parcours. L'ensemble propositions évite de faire des répétitions. (avec m la longueur de la liste)

3. Étant donné un étudiant i et un parcours j, trouver la position de l'étudiant i dans le classement du parcours j.
Structure utilisée : **spe_pref** (liste de listes). Complexité : O(n) en moyenne car il faut comparer la position de chaque étudiant affecté dans la liste de préférences du parcours. (avec n la longueur de la liste)

4. Étant donné un parcours, trouver l'étudiant le moins préféré parmi ceux qui lui sont affectés
Structures utilisées : **prc_prefs** (dict) et **affectations** (dict->set)). Complexité : O(1) car on a précalculé les rangs des étudiants pour chaque parcours, et donc un accès immédiat dans la fonction.

5. Remplacer un étudiant par un autre dans l'affectation courante d'un parcours.
Structure utilisée : **affectations** (dict->set). Complexité : O(1) pour suppression, ajout et mise à jour car on utilise des ensembles, avec accès direct par clé de dictionnaire.

Les deux fonctions de Gale-Shapley sont théoriquement en O(n^2), car dans la boucle principale « while etu_libres » il y a au maximum n itérations puisque chaque étudiant doit être affecté, et dans le pire cas un étudiant peut être rejeté plusieurs fois et propose à tous les parcours dans la boucle « for prc in preferences_etu_courant ».
Le résultat retourné est le suivant :

```
##### Affectation obtenue (Parcours: {Etudiants}) avec le coté étudiants:
 {0: {3, 5}, 1: {4}, 2: {9}, 3: {8}, 4: {10}, 5: {0}, 6: {1}, 7: {7}, 8: {2, 6}}


##### Affectation obtenue (Parcours: {Etudiants}) avec le coté parcours:
 {0: {3, 5}, 1: {4}, 2: {9}, 3: {8}, 4: {10}, 5: {1}, 6: {0}, 7: {7}, 8: {2, 6}}
```

On remarque que les résultats sont très proches, à une différence près des parcours 5 et 6 qui ont échangé d'étudiant.

Finalement, nous avons la fonction **verifier_stabilite** qui prend en entrée une affectation et les listes de préférences des étudiants et parcours, et renvoie les paires instables. Dans les deux cas, aucune paire instable n'est renvoyée.

```
Paires instables étudiants:  []
Paires instables parcours:  []
```

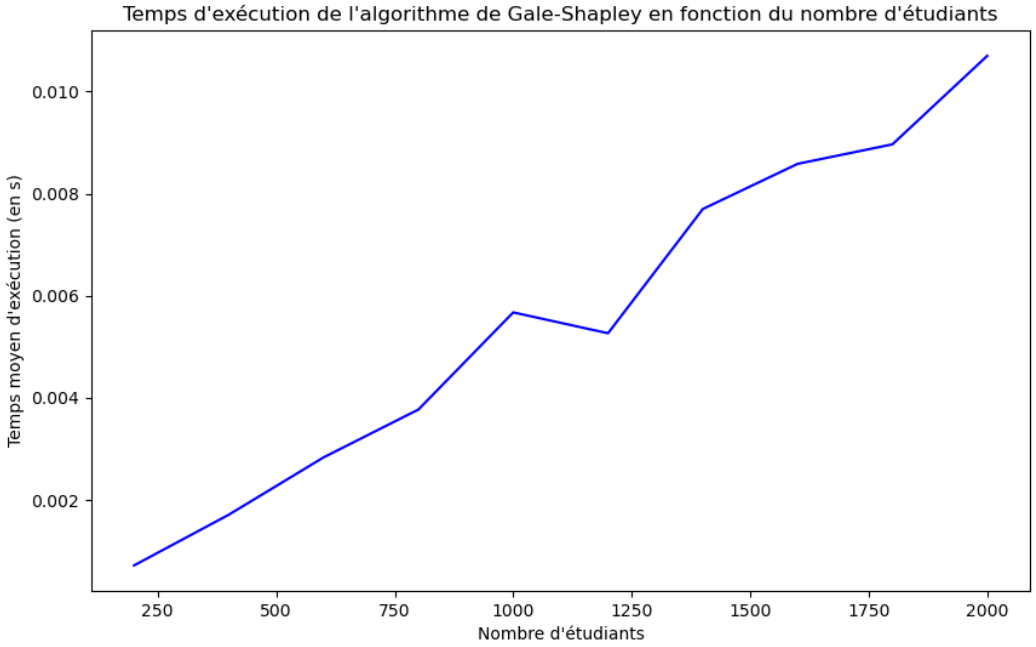
III – Analyse des résultats théoriques

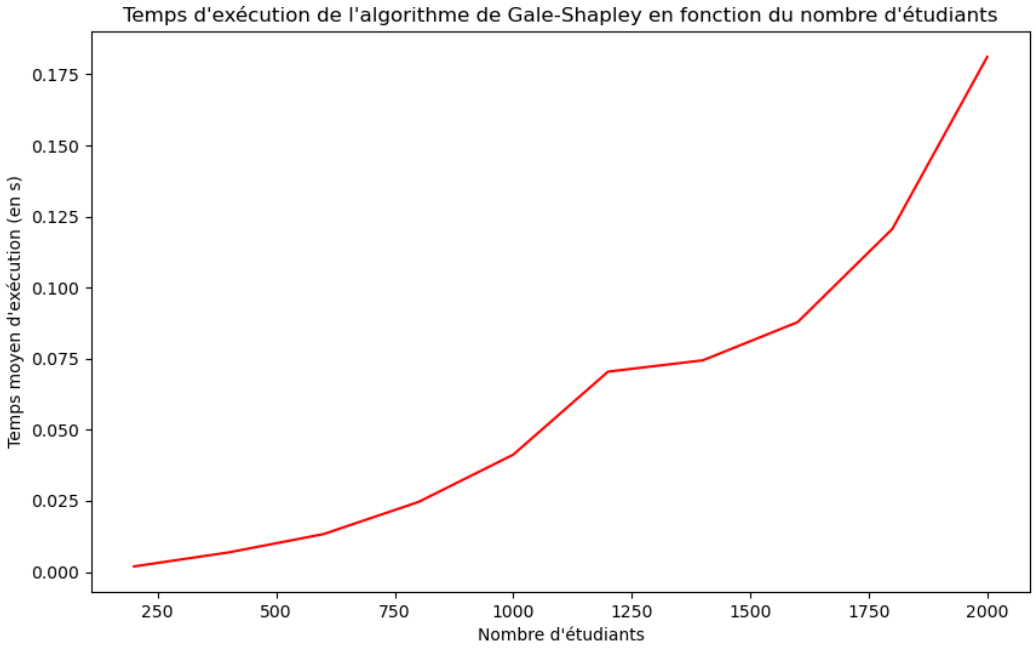
1) Objectif

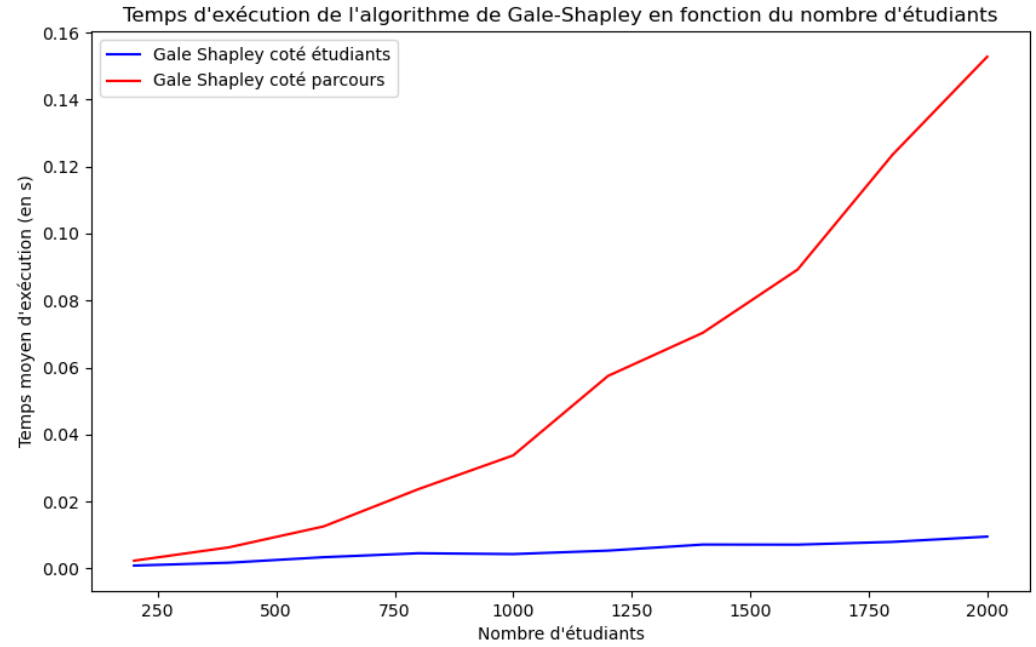
Dans cette deuxième partie, on souhaite analyser les algorithmes côté étudiant et côté parcours en générant des matrices de préférences aléatoires avec un nombre n d'étudiants, et toujours 9 parcours différents. Avec ces matrices, on souhaite analyser le temps de calcul des 2 algorithmes de Gale-Shapley en fonction de n, avec n ∈ [200,2000] avec un pas de 200. On souhaite également que les capacités d'accueil des parcours soient définies de façon à ce que la somme des capacités de chaque parcours soit égale à n, pour une affection pour chaque étudiant.

2) Mise en place

Les fonctions **matrice_cE** et **matrice_cP** génèrent des matrices de préférence aléatoires de taille n passée en entrée à l'aide de la biblio thèque random, et l'utilisation de la fonction shuffle qui va prendre les éléments dans une liste, et les réorganiser aléatoirement.
La fonction **generate_integer_list_v2** génère des listes de capacités aléatoire pour les 9 différents parcours, dont la somme est égale à n.
Les fonctions **time_calculator_etu** et **time_calculator_prc** sont 2 fonctions différentes afin de pouvoir appliquer la fonction GaleShapleyEtu ou GaleShapleyPrc respectivement. Ces deux fonctions vont être lancés l'une après l'autre. Dans le jeu de test, nous utilisons ces différentes fonctions pour tracer deux courbes représentant le temps de calcul moyen en fonction de n. Les voici :

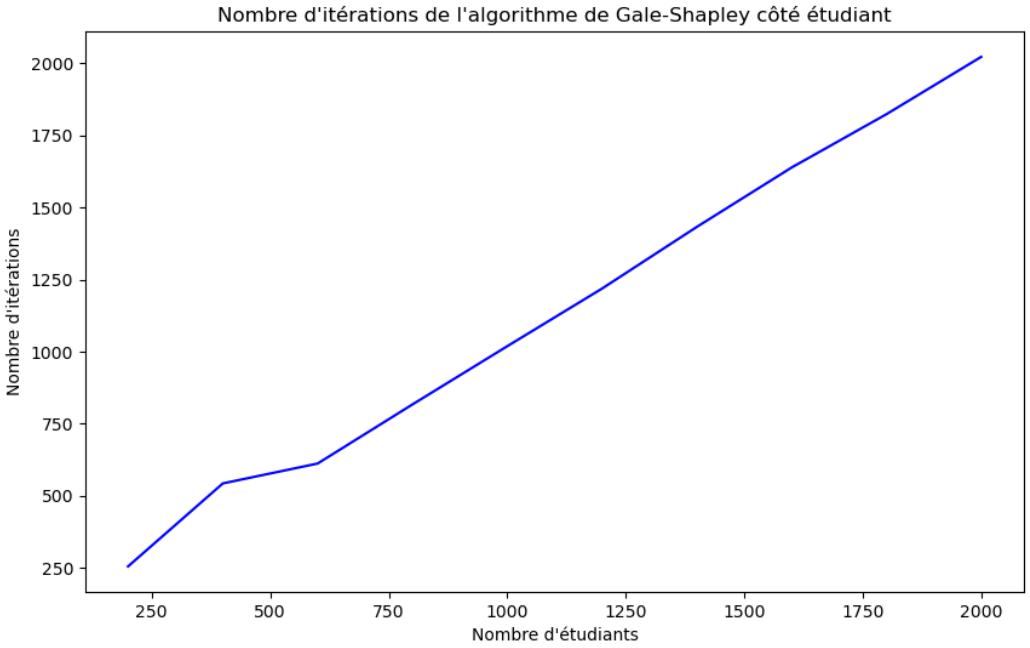


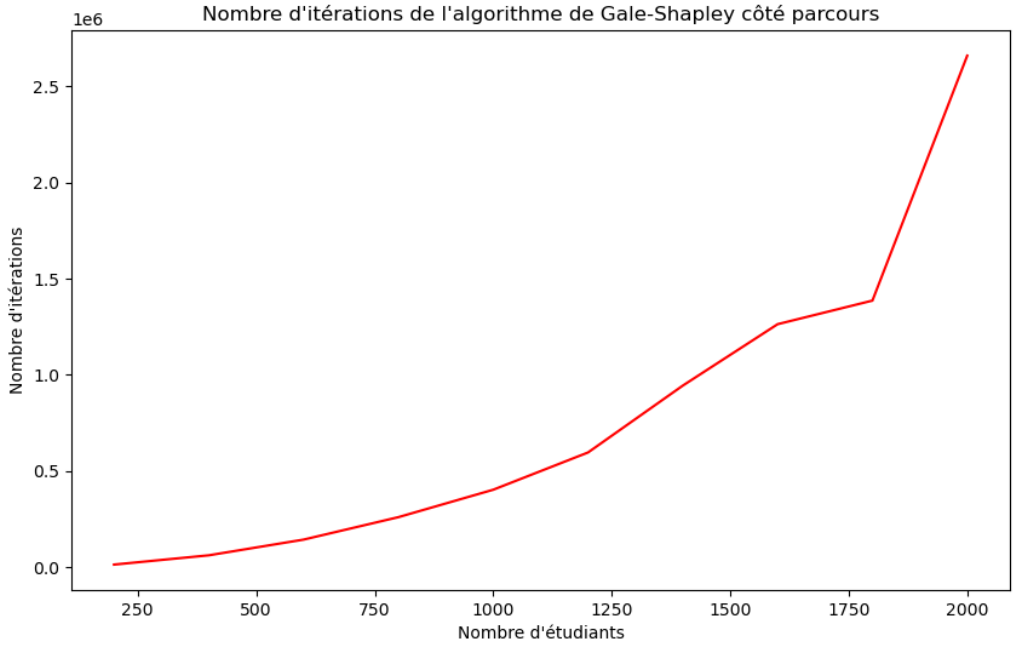




On remarque que l'algorithme côté parcours prend plus de temps que l'algorithme côté étudiant pour les valeurs de n grandes. L'algorithme côté parcours prends en moyenne 0.1 seconde (100 ms), et l'algorithme côté étudiant lui prend 10 fois moins de temps, donc environ 0.01 seconde (10 ms). Donc, l'algorithme côté étudiant est plus efficace en complexité temporelle. Les complexités ici ne sont pas totalement cohérente avec les résultats théoriques, mais peuvent être facilement expliqués : l'algorithme de Gale-Shapley est en moyenne de complexité O(n) et non O(n^2) pour le pire cas. Cela fais donc plus sens avec nos résultats observés pour le côté étudiant, tandis que le parcours côté parcours est plus proche de la complexité théorique.

On veut maintenant faire de même pour le nombre d'itérations de nos algorithmes :





On remarque une différence notable entre les 2 algorithmes : le côté parcours effectue bien plus d'itérations que le côté étudiant. Ce résultat peut être expliqué par le fait que dans l'algorithme pour les parcours, on peut effectuer beaucoup plus de propositions en fonction du nombre d'étudiants, et beaucoup de changement avant d'arriver à un résultat final stable. Ces résultats sont cohérents avec les complexités, le côté étudiant effectue environ 2000 itérations et donc une complexité en O(n), tandis que le côté parcours effectue jusqu'à 2,5x10e6 itérations, qui est plus proche d'une complexité en O(n^2).

IV – Equité et modélisation à l'aide de PLNE

1) Objectif

On veut désormais avoir une équité entre les étudiants dans leurs affectations. Pour cela, nous utilisons la programmation linéaire, qui permet d'optimiser des fonctions par rapport à des contraintes données. Dans notre problème, on cherche à trouver une solution optimale pour les étudiants où chacun a un de ses k premiers choix. Après avoir déterminé les PLNE, nous allons utiliser l'optimisateur Gurobi pour trouver des solutions.

2) PLNE

Dans notre exemple donné, on a :

n=11 étudiants
m=9 parcours
Chaque parcours a une capacité maximale. On veut attribuer chaque étudiant à un de ses k premiers choix.
Soit Xij la variable binaire, valant 1 si l'étudiant i est affecté au parcours j, 0 sinon.
Contraintes :

$$\sum_{j \in P_k(i)} x_{i,j} = 1, \quad \forall i$$ où Pk(i) est l'ensemble des k premiers choix de l'étudiant i.

$$\sum_{i} x_{i,j} \leq C_j, \quad \forall j$$ où Cj représente la capacité du parcours j.

$$x_{i,j} \in \{0,1\}, \quad \forall i,j$$ les variables binaires.

Après génération du fichier affectation.lp, on résout le PL avec Gurobi et on trouve remarque qu'avec la valeur **k=3**, on ne trouve aucune solution. Il faut donc augmenter la valeur de k, ce qui revient à dire que les étudiants ne peuvent pas avoir un parcours dans leurs 3 premiers choix.
On trouve que la valeur minimale de k pour laquelle on obtient une solution est **k=5**.
{0 : {7,10}, 1 : {4}, 2 : {6}, 3 : {9}, 4 : {5}, 5 : {0}, 6 : {1}, 7 : {3}, 8 : {2,8}}
Dans le pire cas, les étudiants 2 et 9 obtiennent leur 5ème choix.
On souhaite désormais trouver une solution qui maximise la somme des utilités des étudiants, mais aussi des parcours. Pour cela, on doit modifier la fonction objectif en une maximisation, soit :

$$\max \sum_{i} \sum_{j} (m - \text{rang}_{i,j}) \cdot x_{i,j}$$

On obtient une solution
{0 : {2,5}, 1 : {7}, 2 : {10}, 3 : {3}, 4 : {4}, 5 : {0}, 6 : {8}, 7 : {6}, 8 : {1,9}}

```
# Solution for model obj
# Objective value = 91
```

Gurobi nous donne la somme des utilités, que l'on va diviser par le nombre d'étudiants pour trouver l'utilité moyenne :
Umean=91/11≃8,27
L'utilité minimale de cette solution est :
Umin=4.

Pour finir, on veut combiner les deux contraintes et trouver une solution où les étudiants a un de leur **k=5** premiers choix, et maximiser la somme des utilités. Pour cela, on a un PLNE

{0 : {2,5}, 1 : {4}, 2 : {7}, 3 : {0}, 4 : {10}, 5 : {1}, 6 : {8}, 7 : {6}, 8 : {3,9}}