Rapport Technique

APB - Student First - ParcourSup

Team 23

Doan Quang Minh
Mahjoub Nidhal
Montoya Damien
Peres Richard

Analyse Technique

Structure du projet

Après la lecture et la compréhension du sujet nous avons réfléchi à la mise en place du projet. Nous avons donc étudié les différentes classes que nous devrons mettre en place, tel le parseur d'arguments, le parseur du fichier d'entrée, les deux classes Student et School et la classe d'algorithme.

Nous avons ensuite développé APB en utilisant l'exemple de mariage fourni. En réalisant APB nous avons réussi par la même occasion à développer Student First ce qui nous a permis de rattraper le léger retard accumulé sur APB. Ce qui a été dur avec ces deux algorithmes a été de réfléchir sur le fonctionnement de ces derniers. Bien que l'algorithme de mariage de base nous ai aidé il a fallu le mettre en place et le faire fonctionner en se basant sur les choix des écoles et des élèves en essayant d'avoir le moins d'élèves non affectés.

Lors de l'étape 2 du projet, et de la mise en commun d'un package, nous avons mis en commun les classes School, Student ainsi que les parseur car ces éléments seront réutilisés pour tous les algorithmes que nous pourrons créer. Si un algorithme diffère de tel ou tel manière, les classes School et Student pourront être étendues pour ajouter les nouveaux attributs et méthodes. Pour mettre en place ce package commun nous avons aussi dû réfléchir à comment faire pour avoir et utiliser des main commun utilisant les mêmes arguments mais pouvant en supporter des nouveaux. Nous avons donc créé la classe MainUtils qui permet de traiter les arguments simplement en l'appelant dans un main. Chaque module pourra donc l'utiliser ou en créer une nouvelle étendu si de nouveaux arguments apparaissent, notamment pour Parcours Sup. De même pour l'utilisation des algorithmes, ceux-ci implémentent une même interface contenant une méthode affecter et print. La méthode affecter permettra de lancer les algorithmes implémentant l'interface de manière rapide et grâce à la méthode print cela affichera sur la sortie standard le résultat de l'affectation (= l'application de l'algorithme). Cette méthode de conception permet d'ajouter de nouveaux algorithmes facilement en implémentant seulement une interface et en utilisant MainUtils. Il n'y aura pas de modification de classes à faire mais seulement des ajouts de classe ce qui respecte le principe ouvert fermé des méthodes SOLID.

Concernant la mise en place de l'algorithme Parcours Sup nous avons dû réfléchir sur plusieurs points. Comment implémenter les différents profils ainsi que l'algorithme. Nous avons donc créé des classes abstract SchoolWithProfile ainsi que StudentWithProfile elles-mêmes étendant respectivement School et Student. Cette classe abstract contenant la méthode abstract affectProfile permettant aux classes filles correspondants aux types de profil d'utiliser cette méthode pour accepter ou non les écoles et les étudiants.

Ce modèle permet l'ajout facilement de nouveaux types d'écoles et d'élèves sans avoir besoin de modifier l'algorithme. Le parseur du fichier input a également été étendu permettant maintenant de créer les écoles et élèves avec leur profil en fonction du pourcentage à avoir. Le principe d'ouverture n'est cependant pas très respecté étant donné que si on ajoute un

nouveau profil il faudra rajouter un case dans le switch du parseur qui crée les nouvelles classes.

Nous avons donc essayé au maximum de développer ce projet pour qu'il soit ouvert à l'extension et fermé à la modification. En effet pour ajouter des arguments pour une future implémentation d'un nouvel algorithme par exemple, il suffit de créer ce ou ces arguments en héritant de Argument (de manière direct si nouveau format d'argument ou en héritant de ArgumentMetric/ArgumentValue) et de les ajouter à la liste d'argument créer dans le Main correspondant au nouveau module. Afin de respecter le principe de Single Responsability, nos arguments sont traités et gérés dans une classe MainUtil qui va se charger de récupérer les valeurs de tous les arguments et lancer la ou les action(s) requise(s).

De ce fait, étant donné que ce n'est pas les arguments qui lance leur actions assignés si appelé mais bien le main, il faut également créer un nouveau MainUtil, héritant du 1er pour les arguments communs, afin de dicter la procédure à adopter en fonction de telle ou telle valeur. Le principe de responsabilité est respecté cependant l'ouverture du code est semi-respecté dans le sens où il est possible mais quelque peu ardue d'ajouter des arguments à traiter (3 ajouts à faire : dans le main, 1 classe par arguments et enfin un main les contrôlant).

Un autre point important à propos de l'ouverture de notre code est que tous nos résultats (affectation, métriques, profils et graph) sont tous modifiables à souhait du fait que ce ne soit pas des types primitifs ou semi-primitifs tel que des int ou des String mais bien des objets Result déclinés sous plusieurs formes avec un affichage séparé de la méthode de calcul.

Cette séparation à par exemple permis l'introduction de la représentation sous forme graphique des résultats de parcourSup très simple et sans rien modifier du module commons au préalable.
Ainsi, si demain on ne souhaite plus avoir un fichier contenant à la suite les numéros d'attributions des étudiants mais par exemple une page HTML ou encore envoyer un mail à chacun des étudiants (si l'info est renseigné et parser correctement au préalable), il est possible de changer l'action par défaut du stockage et de l'affichage du résultat (dans la méthode print() de tous les objets Result) dans ce sens, en renseignant les infos nécessaires dans l'objet Result associé.

De plus, si de nouveaux algorithmes voient le jour la classe MainUtil pourra directement les traiter mais si de nouveaux arguments doivent être créés, il faudra simplement créer un nouveau MainUtil qui étend la classe mère. Il en est de même pour parser les fichiers en entrée, si jamais il diffère d'une quelconque manière il faudra simplement override la méthode ReadStudent et ReadSchool. De même pour les fichiers de sortie, pour modifier l'affichage il faudra créer une classe fille qui étend la classe ResultAffectation et override la méthode print.

Concernant l'ajout de nouveaux profils, que ce soit pour la partie Student ou School, il faudra ajouter une enum dans les classes respectives, créer la nouvelle classe en question et rajouter le cas dans le parseur. Cette partie semble être le point faible du projet car cela ne respecte pas la fermeture à la modification.
Par rapport à la structure du projet nous avons réussi à séparer les codes en petites classes. Nous avons fait en sorte que l'ajout de nouvelles fonctionnalités soit faites facilement ce qui permet une extension assez facile.

Si notre projet devrait être codé procéduralement il y aurait quelques modifications à faire. En effet nous avons utilisés des objets tels que School et Student qui devraient être transformés en structure pour correspondre au C par exemple. Il faudrait également changer les HashMap et les List en réels tableau ce qui serait plus difficile à mettre en place.

APB

Nb Étudiants	10	100	1.000	10.000	100.000	1.000.000
Temps total (s)	0.007	0.028	0.117	0.836	18	1881
Temps parser (s)	0.006	0.018	0.058	0.230	1.535	25
Temps algo (s)	0.001	0.010	0.059	0.606	16	1856
Nb Étudiants non-affecté	0	1	23	436	8284	41115

Student First

Nb Étudiants	10	100	1.000	10.000	100.000	1.000.000
Temps total (s)	0.006	0.024	0.068	0.269	1.829	47.595
Temps parser (s)	0.005	0.020	0.055	0.205	0.973	23.217
Temps algo (s)	0.001	0.004	0.013	0.064	0.856	24.378
Nb Étudiants non-affecté	1	1	32	494	8797	67545

ParcourSup (Profils Ranked/Caution)

Nb Étudiants	10	100	1.000	10.000	100.000	1.000.000
Temps total (s)	0.0076	(~)0.039	(~)0.579	(~)3.658	(~)32	(~)3759
Temps parser (s)	0.006	0.02	0.06	0.234	1.6	24.952
Temps algo (s)	0.0016	0.019	0.519	3.424	30.4	3734.048
Nb Étudiants non-affecté	0	(~)1	(~)27	(~)658	(~)9534	(~)73256

Grâce aux résultats des benchmarks nous pouvons voir que pour 1.000.000 d'élèves seul Student First peut associer les élèves avec les écoles en peu de temps mais le nombre d'élèves non affectés est plus important que pour APB. De plus, notre implémentation de Parcours Sup n'est pas optimale car plus de 73.000 élèves ne sont pas affectés.

Analyse Commerciale

I) Algorithme Admission Post Bac

La méthode employée sur APB se repose essentiellement sur l'algorithme de mariage stable de Gale Shapley. Ainsi, les étudiants propose à toutes les écoles de leur liste de voeux jusqu'à l'obtention d'une affectation (si c'est possible).

L'avantage de cette méthode, comme démontré dans le Benchmark, c'est qu'elle permet, sur une population de 750 000 qui équivaut au nombre de bacheliers de l'année universitaire 2016-2017, d'affecter 95% des candidats.

En revanche, l'inconvénient c'est que pour affecter le plus d'étudiants possibles certains vont devoir se passer de leur premiers choix et cet algorithme présente un temps de calcul relativement élevé par rapport aux autres algorithmes que l'on va aborder.

Notre équipe à bien réussi cette partie dans la mesure que nous avons obtenu les résultats attendu sur les différents jeu de données dans des temps raisonnables.

II) Algorithme Student First

Cette méthode hérite beaucoup de la méthode précédente dans le sens où les étudiants font des propositions aux écoles et se voient affectés selon leur choix. Cependant, si l'étudiant se trompe dans l'ordre des voeux, il peut se faire prendre sa place par étudiant moins classé que lui dans une école.

L'avantage de cette méthode c'est qu'elle respecte mieux les premiers choix d'étudiants. Elle est aussi plus rapide que APB car on ne va pas chercher à effectuer des permutations entre étudiants pour affecter le plus possible.

L'inconvénient est que cette méthode présente un pourcentage d'affectation de l'ordre de 92%, sur une population de 750 000 étudiants cela donne 22500 étudiants non affectés.
Comme pour APB, notre équipe a bien réussi cette partie et on a pu obtenir tous les bons résultats sur les jeu données fournis dans des temps convaincants (23s pour 1 000 000).

III) Algorithme ParcourSup

Ici nous avons une autre procédure d'affectation, chaque étudiant est doté d'un profil qui dicte ses choix, c'est aussi le cas pour les écoles elles accepteront des étudiants selon des stratégies différentes. En outre, les étudiants ne présentent plus des voeux ordonnés mais un ensemble de voeux et ils pourront ensuite, lors d'une proposition, accepter, refuser ou attendre un meilleur choix.

Il est facile de voir que l'avantage de cette méthode c'est de permettre aux écoles de mieux s'adapter en fonction de leur capacité, moyens et préférences pour remplir des filières plus que d'autres par exemple (grâce à l'overbooking).

L'inconvénient c'est que, en fonction des profils, nous pouvons avoir plus d'étudiants non affectés et aussi des temps de calcul plus élevés car on doit parcourir toute la liste des voeux d'un étudiant même s'il obtient une proposition favorable dès son premier.

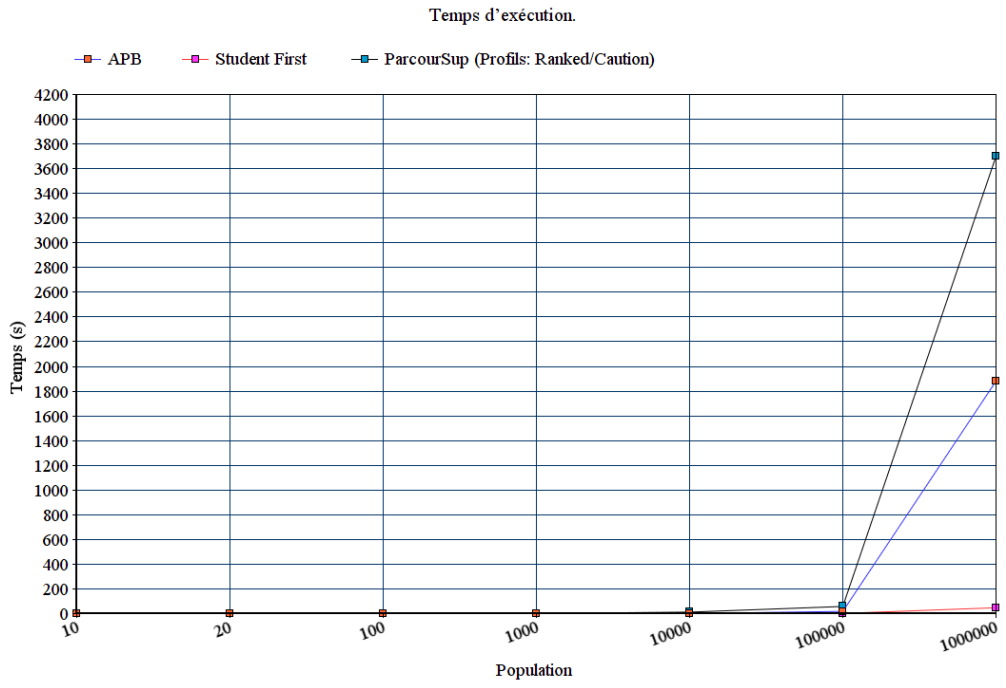
Parmi les profils que nous avons implémenté on cite:
-***Student- Caution*** : Profil par défaut, c'est le comportement le plus naturel et le plus répandu.
-***Student-Curious***: Profil qui nous permet de voir l'effet des décisions de l'étudiant sur la procédure (ralentir ou accélérer)
-***Student-Top5***: Profil qui se rapproche de APB et Student First dans le sens où l'étudiant présente toujours un ordre de préférence.
-***Student-Stubborn***: Profil relativement élitiste permet de tester une situation extrême.
-***School-Ranked***: Profil par défaut, c'est le comportement le plus répandu chez les écoles.
-***School-Open doors***: Permet de voir si , en augmentant le nombre de place libre, le gouvernement peut trouver une solution au nombre d'étudiants non affectés.
-***School-Picky***: Écoles élitistes, permet de tester des situations extrêmes.
-***School-Regional***: Écoles qui recrutent de leur région, puis le reste. Ce profil est assez répandu en France.

Faute de temps, dans les comparaison et métriques qui vont suivre la procédure ParcourSup n'a été testé que sur une population de 20 étudiants, mais nous avons pu toutefois en sortir des conclusions assez pertinentes.

IV) Temps d'exécution des différents algorithmes

I) Résultats en fonction des jeu de données fournis

En essayant de refléter une situation réelle, c'est à dire en utilisant les profils par défaut de ParcourSup , nous pouvons avoir ce résultat (au-delà de 20 étudiants les temps de ParcourSup sont une estimation).



Ces résultats viennent appuyer les arguments donnés lors de la présentation des algorithmes.
Les 3 algorithmes se démarquent à partir des 100 000 étudiants. Ainsi, Student First est le plus rapide car dès la première étape (1èr voeux) il va affecter environ 60% étudiants.

Ensuite vient APB qui ne s'arrête que lorsque l'on obtient le maximum d'étudiants affectés.
ParcourSup termine en dernière position car l'algorithme doit toujours parcourir tous les voeux de tous les étudiants.

Les temps d'exécution de ParcourSup change en fonction des profils mais le classement reste toujours le même.

En conclusion, en terme de temps d'exécution, ParcourSup qui est utilisé actuellement n'est pas le plus performant.

II) Estimations de résultats sur d'autres jeu de données

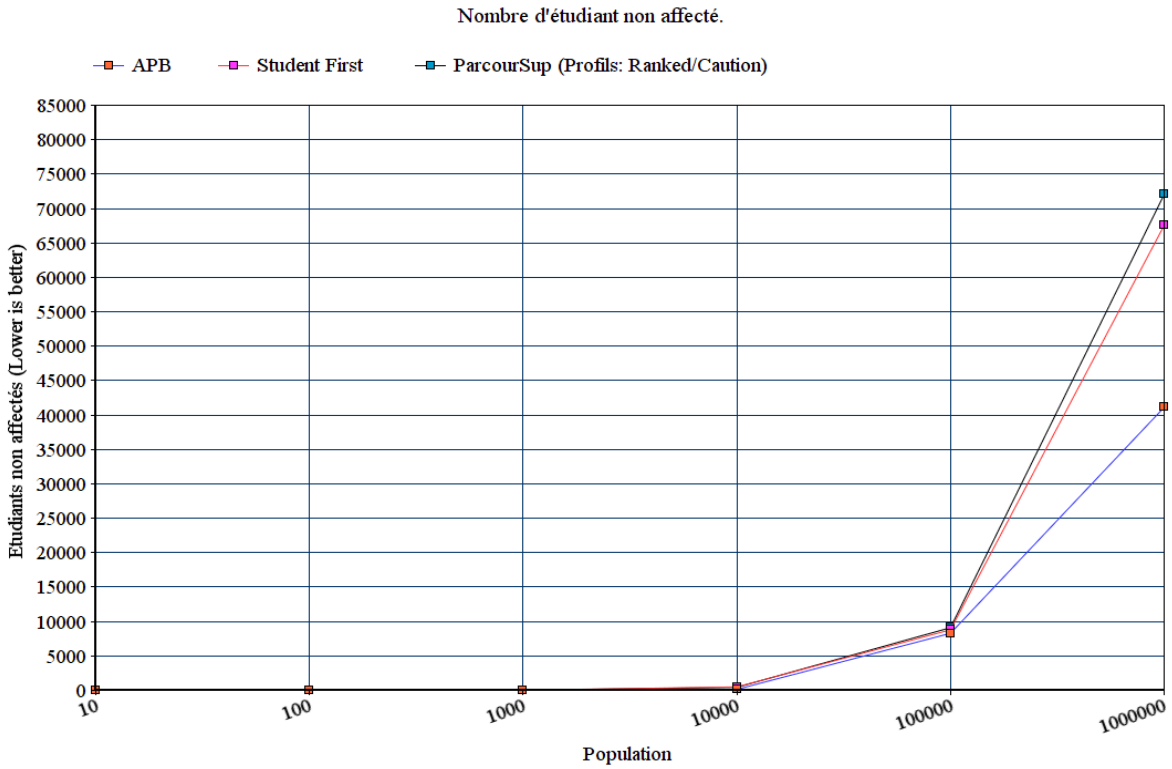
***APB** : Dans la mesure où nous n'avons pas un nombre d'écoles qui est égal au nombre d'étudiants, nous ne pouvons pas utiliser les estimations de l'algorithme de Gale Shapley.
Cependant, nous remarquons un comportement qui se répète sur les jeux de données fourni.
En effet,
si n le nombre de population
m le temps d'exécution
y défini tel que $10^y = n$. $\Rightarrow y*\ln(10) = \ln(n) \Rightarrow y = \ln(n) / \ln(10)$.
x le rapport des temps d'exécution des deux précédentes population
alors le temps d'exécution m de la population (n*10) est égal $m = x * (y-1)$
Donc, en se basant sur nos données du benchmark, on peut estimer le temps d'exécution de APB pour :
-10 millions à : $(m/1881) = 6 * (1881/18) \Rightarrow m = 1\,179\,387$ s
-100 millions à : $(m/1\,179\,387) = 7*(1\,179\,387/1881) \Rightarrow m = 5\,176\,329\,453$ s.

***Student First :** Cet algorithme se rapproche beaucoup du précédent et en gardant les mêmes variables et en étudiant le rapport des temps d'exécution en fonction de la population on peut déduire ce résultat: $m=x*(y-2)$
Donc, en se basant sur nos données du benchmark, on peut estimer le temps d'exécution de APB pour :
-10 millions à : $(m/47.5) = 5*(47.5/1.829) \Rightarrow m = 6167$ s
-100 millions à : $(m/6167)=6*(6167/47.5) \Rightarrow m= 4\,563\,826$ s

***ParcourSup :** Les données qu'on a pu extraire compte tenu des délais qui nous sont imposés ne nous ont pas permis d'extrapoler une estimation concrète mais d'après la comparaison qu'on a vue précédemment nous pouvons s'attendre à des temps encore plus élevés que ceux de APB.

V) Affectation des étudiants:

Toujours en utilisant les profils Ranked et Caution pour ParcourSup, nous nous retrouvons avec des résultats qui ne jouent pas en faveur de ParcourSup.



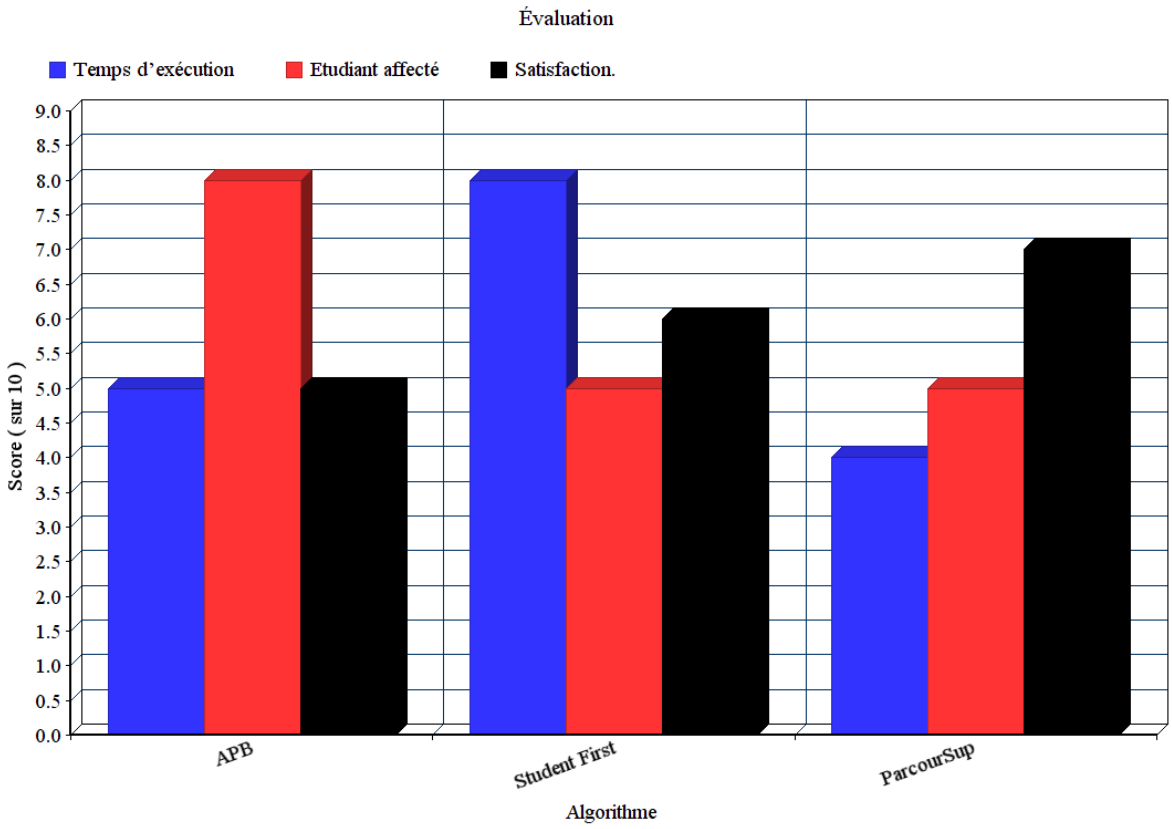
Ainsi, sur nos résultats, ParcourSup se rapproche de Student First d'un point de vue nombre d'affectation et reste assez loin de APB, ce qui n'est pas très intéressant vu les temps d'exécution aperçus précédemment.

Une simulation a donné de bons résultats d'affectation c'est de basculer sur une majorité de profil Open Doors pour les écoles et garder un profil Caution pour les étudiants.
Le souci c'est que ce n'est pas une situation réelle dans la mesure où la plupart des écoles ont des capacités limitées.

VI) Conclusion:

L'avantage principal de ParcourSup c'est qu'il permet aux écoles de gérer son recrutement de façon unique et qui correspond aux besoins et capacités de ces écoles. Aussi, vu que les voeux ne sont plus hiérarchisés, l'étudiant aura plus de temps et plus d'aisance à choisir entre les filières.
Cependant, sur les temps d'exécution et risque de saturation (observé à ParcourSup en mai 2018) on a intérêt à basculer sur Student First qui présente aussi une meilleure satisfaction.

Enfin, APB n'est pas à jeter aussi car il présente un bon ratio temps d'exécution/étudiant affecté.
En définitive, la supériorité d'une méthode par rapport à une autre repose essentiellement sur les profils et les objectifs visés.



Gestion du projet

I) Répartition des tâches:

La répartition des tâches s'est basée sur un terrain d'entente en considérant les compétences et envies de chacun. Ainsi, certains se sont occupés des parseurs, des métriques et de l'exécution et d'autres se sont concentrés sur les algorithmes et les benchmarks.

Concernant l'utilisation de github nous avons décidé concernant de le système de branche d'en créer une nouvelle pour chaque issue, ainsi nous pouvions travailler chacun sur chaque classe sans problèmes et avoir le moins de conflits lors des merge.

II) Evaluation des tâches:

Tâche	Note	Commentaire
Task management	**4/5**	Tout le monde connaissait, depuis le départ, ses responsabilités .
Tests	**3.5/5**	Par manque de temps, certains tests ne sont pas très poussés.

Object-oriented quality	**4/5**	Nous avons accordé une grande importance à ce que le code respecte l'orientation objet et le principe de responsabilité unique ainsi que de l'ouverture.
Releasing process	**4.5/5**	Nous avons pu faire toutes les releases dans les temps jusque là et nous avons pu suivre notre amélioration.
Documentation	**3/5**	Certaines classes ne sont pas très bien documentées.

III) Evaluation des efforts fournis:

Que ce soit sur le code, les tests, l'optimisation, la réflexion, tout le groupe a donné le meilleur de lui-même durant cette semaine pour avoir le meilleur rendu possible. Ainsi, nous divisons les 400 points en 100/100/100/100.

Conclusion

Cette dernière semaine n'était pas simple, la contrainte temporelle couplée avec l'ampleur du projet nous a permis d'avoir un avant-goût de ce que l'on appelle "Crunch Period" qu'on peut voir sur certain grand projet.

Ainsi, nous avons du bien départager les tâches et ne jamais se relâcher afin de pouvoir avoir un bon rendu.

En outre, nous avons eu recours à des connaissances de divers enseignements pour venir à bout de certains attendus comme l'approche algorithmique, le calcul statistique pour les comparaisons, les techniques de génération de graphe et de l'analyse des résultats ainsi que la capacité de pouvoir tout rassembler, d'une façon claire et succincte, dans un rapport à rendre. Grâce aux cours de Java nous avons pu rendre le code plus immuable. L'utilisation de Maven dans ce projet nous a montré la nécessité d'utiliser un système de gestions de dépendances. De même, nous avons ainsi pu découvrir l'outil Travis nous donnant beaucoup d'informations complémentaires et permettant une intégration continue plus efficace.

Enfin, au cours de cette semaine, nous avons appris l'importance de la communication et de la gestion du temps. Et d'un point de vue technique, nous avons vu la différence entre un code efficace et un autre d'une complexité élevé et leurs effets sur un problème réel qui touche une grande population.