ENSAE Paris

Compte-rendu du projet Python

Optimisation d'un réseau de livraison

Antoine Gilson, Soline Mignot

Pour le 6 avril 2023

Table des matières

Introduction

L'objectif de ce projet python, comme annoncé par son titre est d'optimiser un réseau de livraison.

Pour ce faire, considérons un réseau routier constitué de villes et de routes entre les villes. Le but de l'optimisation est de choisir pour chaque trajet entre deux villes, le camion le plus adapté pour le trajet, avec comme contrainte la puissance minimale requise pour emprunter la route. Ainsi, on maximisera les profits grâce à l'acquisition la plus optimale de la flotte de camions. Le travail d'optimisation se divise alors en deux étapes. Il faudra dans un premier temps déterminer pour chaque couple de villes si un camion d'une puissance donnée peut couvrir le trajet entre ces deux villes, ce qui revient à calculer la puissance minimale pour couvrir un trajet ; puis chercher à optimiser la flotte de camions qu'on achète en fonction des trajets à couvrir.

1 Utilisation des scripts et tests unitaires

Le projet rendu contient
— 2 fichiers .py : rendufinal.py et main.py.
— Un document README
— Un dossier input

Le fichier main.py est celui à exécuter pour lancer les programmes, le fichier rendufinal.py contient toutes les fonctions du projet.

Le dossier input regroupe tous les fichiers network, routes et trucks.

Les tests unitaires utilisés au cours du développement des codes sont tous situés dans le fichier rendufinal.py, à la fin de chacune des séances.

Le seul module utilisé est time.

2 Calcul de la puissance minimale pour couvrir un trajet

Dans cette partie, on s'intéresse à la création de la fonction qui renvoie la puissance minimale pour un trajet. La séance 1 proposait une première approche naive, qui a ensuite été optimisée pendant les séances 2 et 3.

2.1 Première approche : la fonction *min_power*

2.1.1 Premiers contacts avec le projet

Tout d'abord, il nous semblait remarquable de rappeler que l'acclimatation aux divers logiciels utilisés pour le projet fut l'une des premières difficultés rencontrées. La notion de classe sur python, où encore les synchronisations entre GitHub et VSCode, bien que très utiles par la suite, ont été pendant la première séance de légers obstacles à l'avancée du projet. Notamment la prise en main des nouvelles notations avec "*data_path*" ou "*file_name*" qui diffèrent de python.

2.1.2 Implémentation de l'algorithme

On cherche, à partir des fonctions données par l'énoncé, à créer une fonction *min_ power* qui renvoie la puissance minimale pour couvrir un trajet.

Comme il est difficile de créer la fonction *min_ power* directement, on passe par des étapes intermédiaires comme suggeré par l'énoncé.

En effet, pour calculer la puissance minimale entre deux villes, il faut d'abord savoir si elles sont connectées, ce que l'on peut regarder avec la fonction *connected_ components_ set*, (qui n'est rien d'autre que la fonction *connected_ components* où on a enlevé les doublons). Ces deux fonctions se comprennent bien et se codent bien, mais toutefois il faut faire attention lors de la formalisation, où peuvent se produire des boucles infinies en cas de doublons. (c'est à dire quand on repasse du fils au père alors qu'on vient de passer du père au fils) Il faut alors faire attention à ces cas particuliers et empêcher leur réalisation dans le code.

Ensuite, avec *get_ path_ with_ power*, on vérifie si la route existe pour la puissance voulue, et on renvoie le chemin. Dans *get_ path_ with_ power*, on utilise l'algorithme de Dijkstra, en s'arrêtant de chercher des chemins lorsque la puissance du chemin dépasse la puissance spécifiée. On réduit ici un peu la complexité future de notre fonction *min_ power*.

Il vient alors notre première fonction *min_ power*. Pour la créer, nous avons utilisé en premier l'approche la plus naive car nous n'avions pas de contrainte de complexité : c'est à dire de regarder tous les trajets possibles et de simplement choisir celui avec la puissance la plus faible.

Pour la question 5, bonus, l'objectif est inverse : il faut trouver quelle est la distance minimale entre deux villes à puissance fixée. Pour réaliser Cette fonction, en attendant d'avoir des algorithmes plus performants, permet de compléter l'utilité de *min_ power*. Pour réaliser cette fonction, nous avons également fait la méthode naive : regarder toutes les distances et choisir la plus faible.

2.1.3 Complexité et temps d'éxécution

Cependant, en faisant tourner l'algorithme sur des gros graphes, on se rend compte que le temps d'éxecution est tellement long qu'on n'arrive pas à avoir des résultats. La fonction ne fonctionne que sur des petits graphes. De plus nos fonctions ne marchent que sur des graphes sans cercles (comme des graphes kruskalisés).

Ce temps d'execution aussi long était prévisible car la complexité de notre fonction *min_ power* est $O(V^3 * log(V))$ ce qui est très important. On aurait pu s'attendre à une complexité de $O(V!)$, car on essaie tous les trajets, mais les légères optimisations apportées à *get_ path_ with_ power* permettent d'améliorer un peu la complexité.

Avec la fonction de la question 10, *question1_ séance2*, on calcule le temps d'execution total pour tous les trajets avec notre fonction *min_ power*. Mais on a pas de résultat car le temps d'éxecution est trop long comme dit précedemment.

Il vient alors évidemment la nécéssité d'optimiser cette fonction, pour qu'elle fonctionne plus rapidement et pour tous les graphes.

2.2 Approche optimisée : la fonction *min_ power_ bis*

L'objectif de cette sous partie est de présenter notre nouvelle fonction *min_ power* (*min_ power_ bis*), avec une complexité plus faible que *min_ power*.

2.2.1 Implémentation de l'algorithme

Pour cette partie, on change complétement d'approche après "l'échec" de la partie précédente. On utilise une fonction auxiliaire basée sur l'algorithme de Kruskal. En premier lieu, notre fonction kruskal était inutilisable car complexité beaucoup trop importante. Le souci était que nous n'utilisions pas les dictionnaires, ces derniers permettant d'avoir une complexité linéaire. Alors, nous avons incorporé les dictionnaires dans notre fonction, et notre nouvelle fonction Kruskal a une complexité plus faible de $O(Elog(E))$.

En effet, la fonction kruskal commence par parcourir tous les noeuds dans le graphe et construit une liste triée des arêtes. Cette opération a une complexité de $O(V+E\,|logE)$.

Ensuite, comme la fonction *find* a une complexité en temps quasi-constante, n'ajoute pas de complexité supplémentaire.

Enfin, la fonction parcourt la liste triée des arêtes et vérifie si l'ajout de l'arête crée un cycle ou non. Si ce n'est pas le cas, elle ajoute l'arête à l'arbre couvrant et fusionne les composantes connexes des deux noeuds de l'arête. Cette boucle a une complexité de $O(E \log V)$. En combinant ces trois étapes, on obtient une complexité totale de $O(E \log E)$ pour l'algorithme de Kruskal.

On introduit également une fonction *parentalité* pour optimiser notre nouvelle fonction. L'objectif de la fonction *parentalité* est de créer un dictionnaire qui nous permettra, pour construire les trajets, de ne plus qu'a avoir à piocher dans les noeuds du dictionnaire. On élimine ainsi les trajets inutiles.

Il vient la création de la fonction *min_ power_ bis*. Cette fonction ne marche qu'avec des arbres kruskalisés. Mais, ici, comme la fonction est optimisée pour les arbres kruskalisés, on a beaucoup moins d'arêtes, et on a bien des résultats pour tous les graphes peu importe leur taille. Pour la réalisation de la fonction, il n'était pas facile de démarrer et d'avoir une fonction opérationnelle tant que nous n'avons pas eu l'idée de passer par la récursivité et les dictionnaires. Une fois les idées assimilées, le code n'est pas spécialement compliqué, mais l'astuce réside dans l'idée.

Par ailleurs, nous avons fait une variante nommée *min_ power_ trois* qui ne retourne que la puissance et pas le trajet. Cela pourrait servir pour des fonctions ultérieures.

2.2.2 Complexité et temps d'éxécution

Intéressons nous à la complexité de notre nouvelle fonction, a-t-on réussi à la réduire ? C'était l'enjeu principal de la nouvelle fonction.

Tout d'abord, la complexité de la fonction *parentalité* est $O(V + E)$ car elle parcourt tous les sommets et toutes les arêtes du graphe pour construire le dictionnaire *pere_ dict*.

La complexité de la fonction *trajet* dépend de la profondeur maximale de l'arbre de la parentalité, qui est de $O(V)$ dans le pire des cas (lorsque le graphe est une chaîne). Chaque appel récursif diminue la profondeur de 1, donc le nombre d'appels récursifs est également de $O(V)$.

La complexité totale de la fonction *min_ power_ bis* est donc de $O(V+E+V^2)$, ce qui simplifie en $O(V^2)$ dans le pire des cas lorsque le graphe est une chaîne. On a bien baissé la complexité avec la nouvelle modélisation.

Mais surtout les temps d'exécution sont beaucoup plus courts, comme on peut le constater avec *question15_ séance2*, cette fois ci appliqué à *min_ power_ bis*. On trouve des résultats, ce qui n'était pas le cas avec *min_ power*, et ce pour tous les graphes.

Ainsi, on on a trouvé une fonction qui marche en un temps rapide pour toutes les routes, même les plus gourmandes en calcul. (on a au plus *question15_ séance2(9)* = 190 secondes).

On peut directement passer à la seconde partie : acquisition des camions.

3 Optimisation de l'acquisition de la flotte de camions

Maintenant que nous avons récupéré les puissances minimales pour effectuer chacun des trajets, l'objectif est maintenant d'optimiser l'acquisition des camions, tout en restant dans la contrainte budgétaire.

3.1 La gestion de l'aspect ouvert du problème

L'un des aspects stimulants mais également frustrants de cette partie est son aspect beaucoup plus ouvert. Parfois, on a des idées, mais en codant on se rend compte des cas où notre idée ne marche pas, et on ne peut pas toujours disjoindre les cas. De même, on ne sait pas toujours quels outils utiliser, entre listes, dictionnaires, listes de liste... Cela dit on peut toujours s'inspirer des questions précédentes et outils précédents, ce que nous avons fait.

La séance 4 est de loin la plus dure de par toutes les possibilités.

3.2 Travail préliminaire

Le travail préliminaire consiste à récuperer tous les camions avec la fonction *camions*, qui a chaque camion associe sa puissance et son coût.

De même, on va récuperer toutes les routes qui vont être parcourues par les camions, ainsi que leurs puissances minimales avec la fonction *puissances_ minimales_ routes*

Pas de difficulté majeure pour coder ces fonctions, mis à part ne pas se tromper dans les indices des classes graph, les *data_ path* et *file_ name*

3.3 Méthodes de résolution

Enfin on va chercher à résoudre le problème : optimiser l'achat des camions

3.3.1 Méthode brute

La première idée qui vient est comme souvent la plus naive : une méthode brute. L'objectif est de récuperer toutes les routes, tous les camions et on trie ces derniers par ordre croissant de prix. On implemente également une fonction *truck_ choice* prend en argument une route (une liste contenant deux villes) et une liste de camions sert à choisir le camion le plus approprié pour parcourir une route donnée, en fonction de la puissance minimale requise pour effectuer le trajet et des puissances des camions disponibles.

Puis ensuite, on calcule tous les coûts, tous les gains, et on cherche la meilleure combinaison, tout en s'assurant que les couts soient inférieurs au budget qui nous est alloué.

Mais cette fonction a une complexité beaucoup trop importante est n'est pas viable dans le cas de gros graphes. En effet, si n est le nombre de routes et m le nombre de camions la fonction teste toutes les combinaisons possibles de routes, ce qui donne une complexité de $O(2^n)$. Pour chaque combinaison, la fonction teste chaque camion possible pour chaque route, ce qui donne une complexité de $O(m)$: On a alors une complexité exponentielle $O((2^n) * m)$

On cherche alors une meilleure solution.

3.3.2 Méthode glouton

De manière générale, un algorithme glouton résout un problème en prenant la meilleure décision locale à chaque étape, dans l'espoir que cette approche locale conduira à une solution globale optimale.

Dans le cas présent, on cherche à approximer une solution à la complexité moins importante mais qui reste pertinente, et on sait que les algorithmes gloutons sont souvent utilisés pour résoudre des problèmes type problème du sac à dos.

Ici, le concept glouton consiste a classer nos chemins par efficacité (le rapport de l'utilité sur le coût) croissante, et pour chaque chemin on y associe le camion ayant la puissance minimale requise pour le parcourir et ayant le coût le plus faible parmi ceux qui ont cette puissance. On s'arrête quand on a plus de budget. On aura ainsi choisi à chaque fois à la meilleure solution à l'échelle locale dans l'espoir d'obtenir la meilleure approximation à l'échelle globale.

Enfin, comme la fonction doit nous renvoyer la flotte de camions, on définit *glouton* de sorte qu'elle renvoie le gain total obtenu et le dictionnaire *camion_et_trajet* contenant pour chaque route l'indice du camion choisi.

La complexité de cet algorithme n'est pas exigée et assez difficile à calculer mais elle est nécessairement plus faible que celle de la méthode brute au vu des temps d'exécutions.

La solution trouvée est satisfaisante tant au niveau de la rapidité que de la précision.

Conclusion

Grâce à ce projet, nous avons pu découvrir un tout nouvel environnement informatique, avec des outils pionniers du monde de l'informatique comme Github ou VSCode. Travailler du python en autonomie, avec les exigences attendues, notamment sur la structure du code, nous a fait beaucoup progresser, et cela s'est ressenti même au long des semaines, venant se conclure avec la confrontation face à la question 18. A chaud, la capacité que nous pensons le plus avoir développé est l'aptitude à tester et à debugger des codes, à travers la manière dont était construit le sujet, nous poussant à revenir sans cesse sur nos précedents algorithmes pour en déceler les faiblesses.

Concernant notre travail, un problème est l'absence de réponse à la question 10 à cause de la complexité trop importante de la première fonction *power_min*. Toutefois, sans avoir traité les questions bonus, nous avons réalisé deux fonctions à complexité raisonnable qui répondent pour tous les graphes aux deux problématiques soulevées par le sujet : la puissance minimale pour couvrir un trajet et l'optimisation de l'acquisition de la flotte de camions.

Lien du github :
https : *//github.com/solinemignot/OMI*1*C*9*_rendu* − *Gilson_mignot/tree/main*