ENSAE Paris

Projet de programmation
Optimisation d'un réseau de livraison

Auteurs :
Jérôme ALLIER
Maxime FACON
Axel PINCON

Responsable du cours :
Patrick LOISEAU
Référent :
Quentin JACQUET

`https://github.com/axp1cn/python_project_afp.git`

6 avril 2023

Table des matières

Chapitre 1

Présentation du projet

1.1 Projet et problématique

Le projet de programmation était un projet sur 2 mois : de début février à début avril. L'objectif final était d'optimiser un réseau de livraison. Nous étions une équipe de trois pour mener à bien le projet. Le projet était guidé pour nous amener étape par étape à faire progresser le code. Nos objectifs en tant qu'étudiants de première année étaient :

— Apprendre à structurer du code pour un projet de moyenne envergure ;
— Apprendre à chercher en ligne les informations, les documentations, et les librairies utiles ;
— Apprendre à analyser un problème et à le traduire en terme algorithmique ;
— Apprendre à analyser la complexité d'un algorithme ;
— Apprendre à manipuler quelques algorithmes et structures de données classiques ;
— Apprendre à debugger, tester, et optimiser du code ;
— Se familiariser avec les bonnes pratiques de code et avec un environnement d'édition ;
— Apprendre à coder en équipe, en utilisant GitHub.

Le projet était constitué de 20 questions plus ou moins indépendantes, nous amenant à l'optimisation du réseau de livraison. L'objectif des premières questions était de nous familiariser avec la notion de classe, en complétant la création d'une classe Graph pour modéliser une carte de la zone sur laquelle notre transporteur opère. Ce graphe est constitué de nœuds et d'arrêtes modélisant les chemins ayant chacun une puissance minimale qu'un camion doit avoir pour passer et une distance. Nous cherchions ensuite à obtenir nos premiers résultats sur ces graphes : obtenir un chemin, obtenir le plus court chemin, obtenir le chemin de puissance minimale. Pour cela nous avons utilisé des méthodes récursives : parcours en profondeur, algorithme de Djikistra. Ensuite, pour optimiser nos temps de calculs, nous avons, à l'aide de l'algorithme de Kruskal, pu extraire l'arbre couvrant de poids minimal, nous permettant d'obtenir les chemins de puissance minimale de façon beaucoup plus rapide. Les dernières questions portaient sur l'utilisation d'un pré-process pour améliorer la complexité de recherche de chemins de puissance minimale, et enfin d'ajouter du réalisme au modèle en ajoutant un coût de carburant proportionnel à la distance ainsi qu'une probabilité qu'une arrête devienne impraticable.

1.2 Organisation du travail en groupe

Nous avons eu six séances de TP de trois heures durant lesquelles nous pouvions avancer notre travail et profiter de la présence de notre encadrant pour lui poser nos questions sur des librairies Python, l'utilisation de GitHub ou encore l'optimisation d'un algorithme. Entre ces séances de TP nous étions libres de nous organiser comme nous préférions. Nous avons débuté en équipe de deux : Axel et Jérôme. Puis lors de la deuxième séance de TP, Maxime nous a rejoint avec l'accord du responsable et de notre référant car son binôme ne s'était pas manifesté. Nous avons donc passé la seconde séance de TP en partie à mettre à jour Maxime sur notre travail, et vice-versa, nous mettre à jour sur le travail de Maxime. Nous avons mis en commun nos premières questions puis nous nous sommes organisés pour la réalisation du projet à trois.

Sur l'ensemble des deux mois de projet, nous avions un fonctionnement assez fluide, chacun réfléchissait aux questions de son côté en informant les autres de son avancement, puis quand quelqu'un avait une solution à proposer il la soumettait sur le GitHub commun. Les autres pouvaient ensuite corriger ou optimiser si besoin et surtout comprendre la solution proposée. Parfois, nous nous sommes répartis les tâches, pour les questions 6, 7 et 12 par exemple. L'objectif était surtout que chacun d'entre nous réfléchisse sur le maximum de questions possibles puis comprenne chacun des algorithmes pour que le projet nous soit à tous utile. Nous avons chacun des qualités différentes et nous en avons profité pour optimiser notre temps de travail. Jérôme a aimé réfléchir à

la structure de l'algorithme qui apporte la solution et Axel s'est plu à concrétiser les idées en code et à adapter l'algorithme à la classe et aux données utilisées comme pour la question 18. Maxime a commenté certaines parties, et proposé des solutions à plusieurs questions. Nous avancions tous les trois ensembles, et les qualités organisationnelles et rédactionnelles de Maxime ont été essentielles à la bonne réalisation du projet.

Nous avons bien évidemment rencontré des difficultés certaines fois pour comprendre l'algorithme ou la solution proposée par l'un de nous. Nous nous sommes regroupés en dehors des heures de TP pour clarifier tout cela ensemble. Nous ne commentions pas toujours l'algorithme au moment de la rédaction, et ce fut un bon exercice de commenter le code proposé par un autre pour le comprendre parfaitement. La compréhension de l'utilisation de GitHub pour pouvoir travailler simultanément sur le code fut la priorité lors de la première séance. Grâce à cela nous n'avons presque pas rencontré de difficultés de collaboration en ligne. Nous nous mettions au courant des commits de chacun d'entre nous pour prévenir les autres de mettre à jour leur code avec les dernières avancées.

Nous avons su faire preuve de flexibilité et d'ouverture en mettant à jour notre travail et en réorganisant nos méthodes de travail pour inclure Maxime dans le projet. Cela témoigne d'une capacité à travailler en équipe et à faire preuve d'adaptabilité face aux imprévus. De plus, notre organisation et notre communication régulière sur GitHub ont permis une collaboration efficace malgré le fait que nous travaillions essentiellement chacun de notre côté. Enfin, le fait de nous réunir pour clarifier les parties difficiles et de commenter le code pour le comprendre parfaitement démontre un engagement commun pour la réussite du projet.

Chapitre 2

Notre code

Notre code est divisé en trois dossiers différents : le fichier main.py, le fichier graph.py, les fichiers tests.

La majorité de notre code se trouve dans graph.py qui contient la classe Graph et ses méthodes, la fonction graph from file, et l'algorithme de kruskal ainsi que sa structure de données Union-Find.

Dans le fichier main.py, nous testions nos algorithmes avec des fichiers network, nous pouvions y mesurer les temps de calculs et vérifier les résultats à l'aide des représentations graphiques. Nous y avons donc mis les codes répondant aux questions de mesure de temps de calculs, ainsi que la réponse à la question 11, justifiant que la puissance minimale pour couvrir un trajet t dans un graphe est égale à la puissance minimale pour couvrir ce trajet dans l'arbre couvrant de puissance minimale du graphe .

2.1 Le graphe et ses chemins

Dans cette partie nous expliquerons nos réponses aux questions 1 à 9. En expliquant notre classe Graph et les premières méthodes que nous avons rédigées.

2.1.1 La classe Graph

Dès le début de notre code, nous définissons une classe appelée Graph qui représente un objet graphe et contient des méthodes utiles pour travailler avec des graphes. La classe a des variables d'instance qui stockent les noeuds, le nombre de noeuds, le nombre d'arêtes et le graphe sous forme de dictionnaire qui contient la liste d'adjacence. L'utilité de cette classe est de faciliter la manipulation de graphes en regroupant les opérations courantes dans une seule classe, ce qui permet de mieux organiser le code et d'éviter la répétition de code. Nous avons ajouté des attributs pour les besoins des questions 14, 16 et 20.

La méthode add edge permet d'ajouter une arête au graphe, cela nous sert à construire nos graphes depuis les fichiers network. Nous avons écrit ensuite une fonction graph from file qui permet de convertir les données des fichiers network.i.in sous la structure de données que nous avons défini dans Graph, elles seront ensuite utilisables pour tester nos algorithmes. Ainsi la première question nous permettait de prendre en main la classe Graph et de préparer les données à être traitées comme des graphes par nos méthodes.

2.1.2 Les algorithmes de recherche de chemins

La première étape fut d'écrire une méthode permettant d'obtenir les composantes connexes d'un graphe. L'algorithme de composantes connexes pour un graphe non orienté consiste à parcourir le graphe et à identifier toutes les composantes connexes en utilisant une recherche en profondeur (DFS) récursive à partir de chaque nœud non visité. Chaque composante est stockée dans une liste.
La complexité temporelle de l'algorithme est O(V+E) où V est le nombre de nœuds et E est le nombre d'arêtes dans le graphe, ce qui signifie que la complexité peut atteindre $O(V^2)$ dans le pire des cas pour un graphe dense.

Cette complexité est justifiée par le fait que pour chaque noeud du graph, l'algorithme visite ses voisins une seule fois. Si le noeud a k voisins, alors cela prend O(k) temps pour le traiter. Dans le pire des cas, chaque arête est visitée une seule fois, car une arête relie deux noeuds différents. Par conséquent, la somme de tous les degrés des noeuds est 2E. La boucle for parcourt donc toutes les arêtes du graphe, ce qui prend O(E) temps.

La méthode connected components parcourt tous les noeuds du graph une seule fois à travers la boucle for extérieure. Par conséquent, cela prend O(V) temps. Dans la boucle for extérieure, la fonction dfs est appelée

pour chaque noeud non visité. La complexité de la fonction dfs est de O(V + E), comme nous l'avons vu précédemment. Comme chaque noeud n'est visité qu'une seule fois, la complexité totale de l'appel à la fonction dfs est de O(V + E).

Ensuite nous avons pu écrire la méthode get path with power. Elle cherche un chemin entre deux points avec une contrainte de puissance donnée. La méthode utilise une approche récursive pour parcourir le graphe et vérifier si les nœuds visités satisfont la contrainte de puissance. Elle commence par le nœud source et explore ses voisins jusqu'à atteindre le nœud de destination ou épuiser toutes les possibilités de chemins. Si un chemin est trouvé, elle renvoie la liste des nœuds dans le chemin, sinon elle renvoie None. La complexité temporelle de cet algorithme est O(E) dans le pire des cas.

Pour la question 5, nous avis écrit la méthode get shortest path with power. Le code donné implémente l'algorithme de Dijkstra pour trouver le chemin le plus court entre un nœud source et un nœud destination dans un graphe pondéré, en prenant en compte la contrainte de la puissance du camion. L'algorithme utilise une liste des nœuds visités, un dictionnaire des distances les plus courtes entre le nœud source et chaque nœud du graphe, ainsi qu'un dictionnaire précédent pour stocker le nœud précédent de chaque nœud dans le chemin le plus court. Le code parcourt ensuite les voisins non visités du nœud actuel, vérifie que la contrainte de puissance est vérifiée, et calcule la distance totale jusqu'à chaque voisin en ajoutant la distance entre le nœud actuel et le voisin à la distance actuelle. Si la distance totale est inférieure à la distance stockée dans le dictionnaire pour le voisin, le code met à jour la distance et le nœud précédent pour le voisin. L'algorithme se termine lorsque le nœud destination est atteint, ou lorsque tous les nœuds accessibles ont été visités. Enfin, le code renvoie la liste des nœuds composant le chemin le plus court entre le nœud source et le nœud destination.

La complexité en temps de l'algorithme est de O(Eln(V)). Dans le pire des cas (graphe connecté), la complexité est de $O(V^2 \ln(V))$.

Enfin, pour la question 6, nous écrivons notre première minpower qui utilise l'algorithme de recherche binaire pour trouver la puissance minimale requise pour atteindre la destination à partir de la source tout en empruntant le chemin le plus court possible. La fonction commence par chercher la puissance maximale dans le graphe en parcourant chaque nœud et chaque arête, puis elle initialise les bornes de la recherche binaire en affectant 0 à la borne inférieure et la puissance maximale trouvée précédemment à la borne supérieure. Ensuite, la recherche binaire est effectuée jusqu'à ce que les bornes se croisent, en utilisant la fonction get shortest path with power pour trouver le plus court chemin avec la puissance requise pour chaque itération. Si un chemin est trouvé, la borne supérieure est réduite et le chemin est stocké comme le meilleur chemin trouvé jusqu'à présent. Si aucun chemin n'est trouvé, la borne inférieure est augmentée. Finalement, la fonction renvoie une paire composée de la puissance minimale requise ainsi que le chemin le plus court pour atteindre la destination à partir de la source.

La complexité de la fonction est de O(log(max(puissance)*Elog(V)). Cette complexité est due à la recherche binaire et à l'appel de la fonction get shortest path with power pour chaque itération.

2.1.3 La représentation graphique et les tests

Dans la question précédente, nous avons calculé, pour un trajet t donné (soit un couple de nœuds), la puissance minimale d'un camion pouvant couvrir ce trajet ainsi que le chemin de puissance minimale associé. A présent, nous voulons implémenter une représentation graphique du graphe et du chemin trouvé. Pour ce faire, nous avons utilisé la bibliothèque graphviz (conseillée par l'équipe pédagogique). Cette bibliothèque est très simple d'utilisation et est parfaitement adaptée à la tâche demandée. La class graphviz.Graph() nous permet de créer un objet dont les méthodes objet.edge() et objet.node() permettent de générer les arêtes et les nœuds de notre graphe, qui va pouvoir être visualisé grâce à la méthode finale objet.view(). Nous avons parcouru la documentation du module un bon moment et avons joué avec les différents paramètres graphiques tels que la couleur des nœuds, la couleur des arêtes, leur forme, leur légende (en dessous de chaque arête nous affichons la puissance et la distance associées). Finalement, nous avons fait le choix de représenter les nœuds ainsi que les arêtes du graphe avec des couleurs sobres (bleu, noir) puis nous avons mis en évidence le chemin de puissance minimale entre les deux points entrés en paramètres de notre méthode graphicrepresentation() en mettant les nœuds et arêtes parcourus en rouge et en intégrant la légende « Shortest path » en dessous de chaque arête. En ce qui concerne les tests créés lors de notre projet, nous nous sommes en grande majorité inspirés de ceux qui avaient été créés antérieurement par l'équipe pédagogique. En effet, nous avons conservé la même structure à la différence près que nous avons modifié les fonctions/méthodes à tester et modifié les résultats attendus.

Pour améliorer la complexité temporelle et de ce fait les temps de calculs, nous pouvons plutôt que de travailler sur les graphes entiers, travailler sur un arbre couvrant. Plus particulièrement l'arbre couvrant de poids minimal. Comme nous le démontrons dans la question 11, entre deux noeuds, le chemin de poids minimal sera le même dans le graphe et dans l'arbre couvrant de poids minimal.

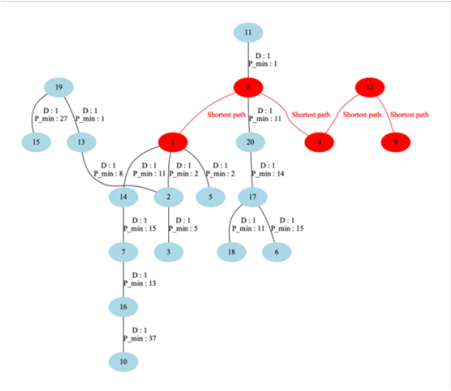

FIGURE 2.1 – Représentation graphique d'un chemin de puissance minimale sur l'arbre couvrant de poids minimal de network1

2.2 Arbre couvrant minimal

Pour trouver l'arbre couvrant de poids minimal, nous utilisons l'algorithme de Kruskal et sa structure de données Union-Find. L'algorithme fonctionne en triant les arêtes du graphe par ordre croissant de poids et en les ajoutant successivement à l'arbre jusqu'à ce que toutes les arêtes soient incluses ou que l'arbre devienne cyclique. L'algorithme de Kruskal utilise Union-Find pour déterminer si l'ajout d'une arête à l'arbre formera un cycle. L'Union-Find maintient un ensemble de sous-ensembles qui peuvent être fusionnés en un seul ensemble et permet de vérifier rapidement si deux sommets appartiennent au même ensemble. Lorsque Kruskal ajoute une arête, il vérifie si les deux sommets qu'elle relie appartiennent déjà au même ensemble. Si tel est le cas, l'ajout de cette arête créerait un cycle et l'arête est ignorée. Sinon, l'arête est ajoutée à l'arbre et les deux ensembles sont fusionnés en un seul.

La complexité de l'algorithme de Kruskal est dominée par le tri des arêtes O(E log E). Dans le pire des cas, la recherche d'un élément dans l'Union-Find peut prendre jusqu'à O(log V), où V est le nombre de sommets dans le graphe. L'opération d'union a également une complexité de O(log V).

Dans l'ensemble, la complexité de l'algorithme de Kruskal est de O(E log E + E log V), ce qui est équivalent à O(E log VE). Cette complexité est considérée comme très efficace pour les graphes denses. Cependant, pour les graphes creux, c'est-à-dire les graphes avec peu d'arêtes, d'autres algorithmes comme l'algorithme de Prim peuvent être plus efficaces.

Nous devons maintenant implémenter une nouvelle méthode, minpower4. Après trois tentatives : minpower1, minpower2 et minpower3, dont les complexités n'étaient pas satisfaisantes (on utilisait mal le fait qu'on est maintenant dans un arbre couvrant de pois minimal). La méthode finalement implémentée, minpower4, exploite seulement le fait que nous sommes dans un arbre couvrant de poids minimal, nous avons juste à expliciter le chemin dans cet arbre entre les deux noeuds (puisqu'il est unique et de poids minimal). Pour cela nous utilisons dfs14, un parcours du graphe en profondeur renvoyant par rapport à un noeud considéré comme racine, la liste des parents de chaque noeud et la profondeur de chaque noeud. Ce pre-process que nous aurons à faire tourner qu'une seule fois par graphe nous simplifie les temps de calcul quand nous aurons de nombreux chemins à trouver dans un même graphe. minpower4 a une complexité en O(h) avec h la hauteur de l'arbre, soit O(V) dans le pire des cas (arbre linéaire). Le dfs14 a aussi une complexité en O(V).

Pour la question 16, nous avons effectué quelques tentatives sans succès... La question n'était pas pourtant beaucoup plus difficile, le pré-process cette fois se contentait, pour chaque noeud, de donner la liste des noeuds ayant une différence de profondeur par rapport à lui en puissance de 2. Le pré-process passe en Vlog(V), mais le minpower passerait en log(V). Cependant, étant donné que la profondeur d'arbre la plus grande, dans les données à disposition, était 40, le gain de temps n'était pas significatif.

Nous avons maintenant un algorithme rapide que nous pouvons utiliser pour optimiser le réseau de transport routier.

2.3 Optimisation du réseau de livraison

Nous voilà dans la dernière partie de ce projet. L'objectif désormais est de maximiser le profit d'une compagnie de transport, ayant un budget fixe, qui achète des camions parmi une sélection de modèles qui seront affectés à des trajets respectifs. Cette problématique peut être associée au problème du sac à dos , duquel nous nous sommes inspirés pour écrire notre programme. En premier lieu, nous avons pensé à ranger les routes du graphe par ordre décroissant du profit (soit la différence entre le profit associé à la route empruntée et le coût du camion utilisé). Seulement, un échange avec notre chargé de TD, nous a appris qu'il était plus optimisé de ranger les routes par ordre décroissant du ratio du profit associé à la route empruntée sur le coût du camion utilisé. C'est donc ce que nous avons choisi de faire pour optimiser l'allocation du budget de l'entreprise de transport. Notre méthode knapsackgreedytrucks() prend en paramètres le budget de l'entreprise et les numéros des fichiers routes.i.in et trucks.i.in utilisés pour notre optimisation. On commence donc par parcourir les fichiers routes.i.in et trucks.i.in pour récupérer les informations nécessaires à l'optimisation (le profit associé à chaque trajet (couple de nœuds) dans le fichier routes.i.in et la puissance et le coût de chaque modèle de camion dans le fichier trucks.i.in). Puis, on calcule les ratios associés à chaque couple trajet-modèle de camion et on range ces couples par ordre décroissant des ratios. On crée aussi un compteur qui permet de suivre le nombre de camions achetés par modèle. Finalement on itère sur les couples classés jusqu'à avoir alloué l'ensemble du budget de l'entreprise entré en paramètre. Bien qu'il s'agit d'une méthode très simple théoriquement, le travail le plus demandant a été de récupérer les données nécessaires au rangement des couples dans les différents fichiers input et de les stocker dans des objets adaptés, pratiques et compréhensibles par le lecteur.

A l'image de la question 7, on veut désormais visualiser l'allocation en camions sur notre graphe. Comme pour la question 7, nous utilisons le module graphviz et nous commençons par représenter le graphe entier (nœuds et arêtes). Ensuite, pour visualiser l'allocation en camions, nous avons choisi de représenter les routes empruntées et d'ajouter en légende en dessous de chaque route le modèle de camion utilisé. On utilise donc notre méthode knapsackgreedytrucks() pour récupérer l'ensemble des trajets couverts associés au modèle de camion utilisé ainsi que notre méthode minpower4() pour récupérer le chemin de puissance minimale associé à chaque trajet couvert.

Nous voilà rendu à l'ultime épreuve ! Nous souhaitons maintenant donner plus de réalisme à notre modèle. Pour cela nous avons attribué à chaque arête une probabilité de se casser et avons introduit un cout de carburant proportionnel à la distance totale du chemin entre deux nœuds du graphe. Pour optimiser l'allocation, nous avons opté pour une méthode qui n'est pas optimale et nous sommes inspirés de ce que nous avons produit à la question 18 du début de séance. En effet nous avons conservé la même structure mais avons mis à jour les profits en espérance pour chaque trajet. Pour obtenir la distance totale du chemin de chaque trajet, nous avons tout d'abord créé une nouvelle instance pour la classe Graph() que nous avons appelé edgeswithpowdist et qui stocke, pour chaque couple de nœuds connectés, la puissance minimale requise pour voyager entre les deux ainsi que la distance les séparant. Nous avons ensuite intégré à notre méthode addedge() une actualisation de notre instance edgeswithpowdist (elle sera mise à jour lorsqu'elle sera appelée dans notre méthode graphfromfile()). Ainsi en itérant sur les nœuds constituant le chemin de chaque trajet, on a pu cumuler les distances entre ceux-ci et obtenir la distance totale du trajet en question. Finalement, on a mis à jour le profit associé à chaque trajet qui est à présent égal au profit initial associé au trajet (récupéré dans le fichier routes.i.in) auquel on retire le coût du carburant (prix carburant * distance totale). On peut maintenant faire tourner le code de la même manière que pour la question 18 sans oublier que le coût d'un trajet est désormais égal à la somme du coût du camion (prix du modèle utilisé) et du coût du carburant.

Chapitre 3

Conclusion

Ce projet nous a permis de nous initier à la résolution de problème de moyenne envergure en Python. Nous avons pris plaisir à découvrir le sujet et ses problématiques durant ces deux mois. Le projet était guidé pour nous amener à optimiser les premières solutions apportées puis répondre à la problématique finale.

Nous avons pu apporté une réponse satisfaisante au problème en des temps de calcul de l'ordre de la minute pour les gros fichiers. Nos algorithmes peuvent être encore un peu optimisés (en gardant la même complexité) pour réduire de quelques secondes les temps de calculs, mais par manque de temps nous ne pourrons pas. Le projet pourrait être continué en apportant de la complexité au modèle comme proposé dans la question 20 et se rapprocher de la réalité pour à terme optimiser un vrai réseau de livraison. A la question 18 et 20 nous aurions pu proposer d'autres algorithmes, il en existe différents types pour résoudre le problème du sac à dos. Nous nous sommes contenter d'un algorithme greedy qui apporte une bonne solution sans être forcément la meilleure solution, mais le fait en temps de calcul raisonnable.

Dans un dernier temps, nous tenions à remercier le responsable du cours, Mr Loiseau, notre référent, Mr Jacquet, ainsi que Mr Baudin pour leur accompagnement sans faille tout au long du projet. Plus particulièrement, nous les remercions pour leur pédagogie, pour la flexibilité dont ils ont su faire preuve (notamment lors de la refonte de notre groupe), mais aussi et surtout pour le temps passé durant les séances à nous aiguiller, à préciser ainsi qu'à pousser nos reflexions. Ce faisant, le projet semble bien avoir permis à chacun d'entre nous de progresser, indépendamment de son niveau initial.