ENSAE Paris

Compte-rendu du projet Python

Optimisation d'un réseau de livraison.

Malo Evain, Marama Simoneau

Pour le 6 Avril 2023

Table des matières

1 Calcul de la puissance minimale pour un trajet.

L'objectif de cette partie du projet était de déterminer la puissance minimale pour réaliser un trajet donné dans la mesure où ce trajet existe.

1.1 Première approche : min-power2.

La fonction min-power2 est une fonction naïve, elle consiste à chercher tous les chemins liant la source et la destination est choisit celui de puissance la plus faible. Dans le pire des cas, l'ensemble des arrêtes est $V \times V$ dans ce cas tout chemin est caractérisé par une unique permutation de V, ainsi il y a card(V) ! chemins et la complexité est en O(card(V) !).

Cette version fonctionne seulement pour des petits graphes. Dès que le graphe dépasse la centaine de noeud, la complexité rend le temps de calcul beaucoup trop long.

1.2 Deuxième approche : min-power.

Cette fonction utilise une méthode dichotomique, il s'agit d'abord de déterminer la liste triée des puissances présentes dans le graphe, il suffit donc de parcourir toutes les arêtes du graphe et stocker leur puissance dans une liste puis de trier cette liste. Par dichotomie on va choisir une puissance dans la liste et tester s'il existe un chemin dans le graphe par lequel on peut passer avec la puissance choisie. Si oui : on essaye une plus petite puissance, sinon on en prend une plus grande.

Dans cet algorithme il faut noter que pour trouver s'il existe un chemin on fait un parcours en profondeur. En effet, l'objetcif n'est pas de trouver le plus petit chemin, mais juste de trouver s'il existe un chemin satisfaisant la condition de puissance. En ce sens le parcours en profondeur nous évite de devoir parcourir tout l'arbre en moyenne. Cela nous permet donc de rendre l'algorithme plus rapide. La complexité de cette fonction est de $O(\log(V) \cdot (V^2))$, où V est le nombre de sommets dans le graphe.

Le premier bloc de code qui stocke toutes les puissances dans une liste a une complexité de $O(V^2)$ car il parcourt tous les sommets du graphe (i) et pour chaque sommet il parcourt toutes les arêtes (j).

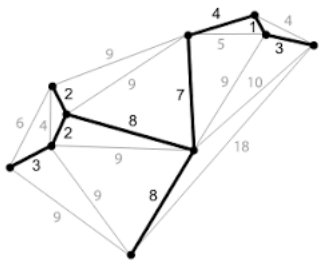
Ensuite, la liste de puissances est triée en $O(V^2 \cdot \log(V^2))$ en utilisant la méthode de tri Timsort de Python.

Le bloc principal utilise une approche de dichotomie pour tester si un chemin existe pour une puissance donnée. La boucle principale s'exécute en $O(\log(V))$ car elle divise à chaque itération la plage de puissances possibles par deux. Pour chaque puissance testée, la fonction exist-path est appelée, qui a une complexité de $O(V)$ car elle parcourt tous les sommets du graphe. Enfin, la recherche dans la liste des puissances explorées a une complexité de $O(V^2)$ car elle peut potentiellement parcourir toute la liste.

Dans le pire des cas, le nombre d'itérations de la boucle principale est de $\log(V)$, et pour chaque itération, exist-path est appelée avec une complexité de $O(V)$. Ainsi, la complexité totale de la fonction est de $O(\log V) \cdot (V^2))$. Avec cette approche on obtient en moyenne qu'il faudrait des centaines de jours pour obtenir la puissance minimale pour tous les trajets. Ce n'est donc pas une méthode viable. Toutefois elle reste meilleure que la première approche.

1.3 Troisième approche : algorithme de Kruskal.

La complexité de l'algorithme de Kruskal implémenté est de O(E $\log E$ + E.V) où E est le nombre d'arêtes dans le graphe et V est le nombre de noeuds.



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 log E). Ensuite, la fonction initialise une instance de UnionFind pour gérer les composantes connexes. Les opérations makeSet et union ont toutes une complexité en temps quasi-constante, find a quant à elle une complexité en O(V).

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.V).

En combinant ces trois étapes, on obtient une complexité totale de O(E log E + E.V) pour l'algorithme de Kruskal.

Maintenant que nous avons transformé notre graphe en un arbre couvrant de poids minimal, trouver la puissance minimale pour un trajet est assez simple. En effet, il nous suffit de trouver le chemin entre le noeud "src" et "dest", puis de regarder quelle est la plus grande puissance sur ce trajet. On obtient alors la puissance minimale requise pour ce trajet.

Cependant, nous avons remarqué que la façon dont on parcourt l'arbre change de façon notable le temps de calcul pour l'ensemble des trajets.

1.3.1 Parcours en profondeur

La première approche pour trouver le chemin entre deux noeuds dans l'abre a été de faire un parcours en profondeur. La complexité (dans le pire cas) est O(V). Ainsi dans le cas de graphe avec des centaines de milliers de noeuds le temps de calcul est encore trop long.

route	Temps de calcul pour un trajet (s)	Temps pour tous les trajets (h)
1	4.415×10^{-6}	1.7176×10^{-7}
2	0.0675	1.875
3	0.0785	10.9049
4	0.1005	13.958
5	0.182	5.0616
6	0.254	35.312
7	0.2467	34.2667
8	0.2971	41.2761
9	0.222	30.948

1.3.2 Amélioration recherche de chemin dans un arbre

Toutefois, arriver à calculer l'ensemble des puissances minimales en 10h est encore trop long. C'est pourquoi nous avons amélioré l'algorithme de recherche de chemin dans l'arbre couvrant.

Nous nous sommes basés sur le fait que la structure de données soit un arbre, ce qui permet de caractériser entièrement un noeud par ses fils, son père (unique), sa profondeur et le poids sur les arêtes. Pour avoir ces informations nous avons fait un DFS qui a donc une complexité en $O(V)$.

Ensuite pour trouver le chemin entre 2 noeuds nous avons appliqué l'algorithme suivant :

Algorithm 1 Trouver le chemin entre deux nœuds

soient deux nœuds n_1 et n_2 on cherche un chemin entre n_1 et n_2

while n_1 et n_2 ne sont pas à la même profondeur **do**
Remonter le nœud le plus bas ;
Ajouter chaque nœud remonté et la puissance de l'arête à chemin ;

while n_1 et n_2 ne sont pas identiques **do**
Remonter les deux nœuds ;
Ajouter chaque nœud remonté et la puissance de l'arête à chemin ; **return** chemin ;

Les résultats obtenus sont assez impressionnants car on arrive à calculer toutes les puissances minimales pour tous les trajets pour toutes les routes en moins de 5 min.

route	Temps de calcul pour tous les trajets (s)
1	0.0003
2	1.34
3	19.76
4	19.10
5	6.20
6	24.69
7	24.01
8	20.91
9	18.37

2 Optimisation du choix des trajets.

Cette partie du projet consiste à choisir les trajets pour obtenir le meilleur profit sous une contrainte budgétaire. Il s'agit d'un problème du sac à dos où les objets sont les trajets, leurs poids sont le prix du camion le moins cher pouvant réaliser le trajet et les valeurs sont les profits correspondant au trajet en question.

2.1 Recherche du meilleur camion pour chaque trajet.

Dans un premier temps, on supprime les camions pour lesquels il existe un camion moins cher et plus puissant. C'est la fonction comparer-puissance-prix qui permet de le faire en O(c^2) avec c le nombre de type de camion. Ensuite, la détermination du meilleur camion se fait en O(n log c) où n est le nombre de trajet, en effet on effectue pour chaque trajet une recherche dichotomique pour trouver le camion le moins cher parmi ceux capables de faire le trajet.

2.2 Résolution d'un problème du sac à dos.

2.2.1 Solution pseudo-naïve : algorithme glouton.

Cette première approche consiste à trier tous les trajets en fonction de leur profit divisé par leur coût. Ensuite on ajoute les trajets jusqu'à ce que le budget soit atteint ou que l'ensemble des trajets a été parcouru. Le tri de cette liste se fait en O(n log n), donc la complexité solution-pseudo-naive est en O(n log n).

2.2.2 Algorithme génétique.

L'algorithme génétique est un algorithme qui se prête assez bien au problème du sac à dos pour donner une solution qui n'est pas optimale mais assez proche. Il se base sur le principe de selection naturelle.

Il se déroule en 3 étapes.

— En partant d'une population de solutions, on va garder juste la moitié (celles qui sont les plus performantes).

— Parmis cette population, on va choisir au hasard des "parents" qu'on va faire reproduire pour avoir des enfants qui sont censés être meilleurs que leur parents.

— Enfin on applique une mutataion aux enfants (probabilité p de prendre un objet ou non de manière aléatoire dans la solution). Cela évite d'être bloqué sur un maximum local. On ajoute les enfants à la nouvelle population et on répète cela de nombreuses fois.

Nous avons implémentés 2 algorithmes génétiques :

— le premier part d'une solution aléatoire.

— le second part de la solution trouvée par l'algorithme pseudo naif.

Le premier cas fonctionne mais met un grand nombre d'itérations pour converger. Ce qui n'est pas efficace dans le cas des fichiers routes 6,7,8,9. Il vaut donc mieux partir d'une solution approchée.

La complexité de l'algorithme génétique dépend de plusieurs facteurs, notamment la taille de la population, le nombre d'itérations et la complexité de la fonction d'évaluation.

Dans l'implémentation fournie, la taille de la population est de 100, le taux de mutation est de 0.1 et le nombre d'itérations est de 50. Par conséquent, la complexité de l'algorithme génétique est en O(100 * 50 * m), où m est la complexité de la fonction d'évaluation, or la fonction d'évaluation

consiste à faire une boucle for sur une liste dont la longueur est n le nombre de trajet donc sa complexité est O(n).

La fonction d'évaluation compare chaque individu de la population avec chaque trajet et calcule la somme des profits des trajets qui peuvent être couverts par l'individu. Par conséquent, la complexité de la fonction d'évaluation est en O(taille-population * n).

En conclusion, la complexité de l'algorithme génétique est en O(5000 * n). En pratique, n est de l'ordre de 100000 donc log(n)«5000, la solution pseudo-naïve est donc plus rapide. De plus à moins de faire un grand nombre d'itérations, l'algorithme ne dépasse pas la solution trouvée par l'algorithme pseudo naif. Or, faire de nombreuses itérations pour des solutions de la taille de centaines de milliers est beaucoup trop gourmand en temps.

2.2.3 Algorithme "aléatoire"

L'algorithme "solution aleatoire" génère une solution aléatoire en utilisant un algorithme pseudo-naïf, puis en ajoutant ou retirant des éléments de manière aléatoire. La méthode commence par appeler la fonction "solution pseudo naive 2" pour générer une solution initiale proche de la solution optimale. Ensuite, elle crée une copie de cette solution initiale, et procède à une série de modifications aléatoires jusqu'à ce qu'elle trouve une solution qui maximise la valeur totale du sac.

On choisit arbitrairement de faire varier le dernier quart des objets mais ca aurait pu être 1/10 ou la moitié. On va alors remplacer certains éléments de cette partie par des éléments qui n'avaient pas un ratio assez intéressant pour être gardés lors de l'algorithme pseudo naif.

La complexité de cet algorithme dépend des complexités des fonctions appelées à l'intérieur de la fonction "solution aleatoire".

La fonction "utilite solution" est appelée une fois pour chaque solution générée par l'algorithme. Elle a une complexité temporelle de $O(n)$, où n est le nombre d'éléments dans la solution.

La boucle principale de la fonction "solution aleatoire" a une complexité de $O(iterations * n)$, où iterations est le nombre maximum d'itérations dans la boucle principale et n est la taille de la solution.

La complexité temporelle de la méthode random.randint(a, b) est de O(1), car elle ne dépend pas de la taille de la liste. La méthode list.remove(x) a une complexité de $O(V)$, où n est la taille de la liste.

Dans l'ensemble, la complexité temporelle de cet algorithme dépend de la taille de la solution, ainsi que du nombre d'itérations. On peut donc la représenter comme suit :
$O(iterations * V^2)$

Cet algorithme trouve de meilleures solution que les 2 autres algorithmes.

2.3 Resultats

2.3.1 Trucks1

route	solution pseudo naive	solution aleatoire	algo genetique
1	675 730	675 730	675 730
2	499 692 265	499 692 265	499 692 265
3	485 781 985	485 782 458	485 781 985
4	1 875 108 015	1 875 110 603	1 875 108 015
5	500 937 561	500 937 561	500 937 561
6	376 392 722	376 394 326	376 392 722
7	376 738 234	376 739 806	376 738 234
8	378 284 075	378 284 075	378 284 075
9	1 794 411 667	1 794 415 261	1 794 411 667

2.3.2 Trucks2

route	solution pseudo naive	solution aleatoire	algo genetique
1	675 730	675 730	675 730
2	499 692 265	499 692 265	499 692 265
3	1 057 279 918	1 057 280 376	1 057 279 918
4	2 499 020 204	2 499 020 204	2 499 020 204
5	500 937 561	500 937 561	500 937 561
6	836 335 720	836 337 752	836 335 720
7	838 553 509	838 553 509	838 553 509
8	838 632 004	838 633 581	838 632 004
9	2 352 199 957	2 352 199 957	2 352 199 957