Rapport projet de programmation

JEAN Aimé
TRAN-THUONG Tien-Thinh
Encadré par : LOISEAU Patrick

April 2023

1 Structure du code

Notre code s'articule autour d'une classe `Graph` qui possède les attributs suivants :

— `nodes` : une liste qui contient tous les noeuds de notre graphe (les objets que contiennent la liste peuvent être de n'importe quel type (int, float, str, ...)).

— `graph` : un dictionnaire qui a pour clés l'ensemble des noeuds du graphe, chaque clé contient une liste de tous les voisins du noeud conrespondant à la clé.

— `nb_nodes` : un entier qui indique le nombre de noeuds du graphe.

— `nb_edge` : un entier qui indique le nombre d'arrêtes du graphe.

— `tree` : cet attribut est à la base initialisé à `None` et est modifié plus tard lorsque nous appliquons la méthode `oriented_tree`. Lorsqu'il n'est pas à `None`, cet attribut est un dictionnaire qui a pour clé les noeuds du graphe et a chaque noeud, on retrouve un tuple contenant son père (son père est lui même si le noeud est la racine), sa hauteur et la puissance qu'il y a sur l'arrête qui relie le noeud à son père sur l'arbre couvrant minimal du graphe.

— `power_2puiss` : un dictionnaire qui a pour clé l'ensemble des noeuds du graphe et qui a chaque noeud associe une liste contenant tous ses ancêtres éloigné d'une hauteur égale à une puissance de 2 dans l'arbre couvrant de poids minimal du graphe. (la variable est à `None` si le process de calcul des ancêtre n'a pas encore été fait).

2 Q 7 : Graphviz

Voici notre réprésentation avec *graphviz* de nos graphes :

- Les noeuds vert=src, rouge=dest, bleu=intermédiaires
- Chaque arrête possède les informations $power, dist$

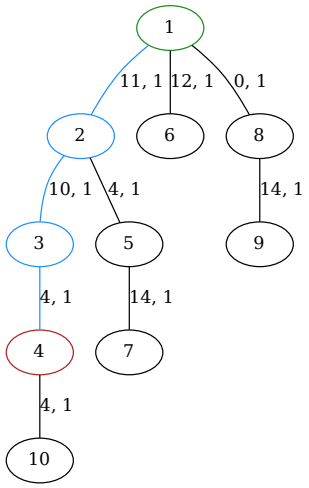


FIGURE 1 – La représentation graphique de network.00 : chemin de 1 à 4

3 Q 8,10 : Un problème de temps

Nous voyons dans la simulation ci-dessous que plus le network est grand, plus le temps d'exécution augmente. Jusqu'à être de l'ordre de la seconde pour les plus grand graphe.

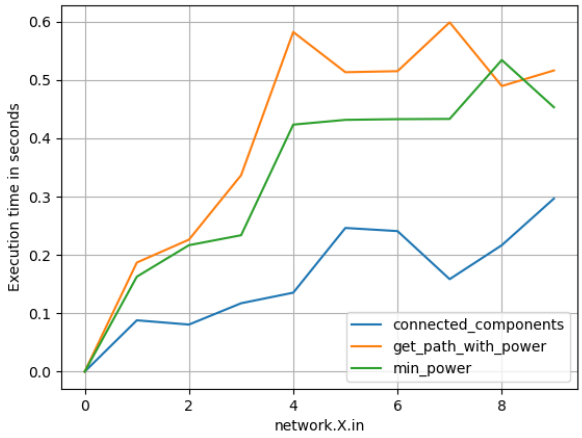
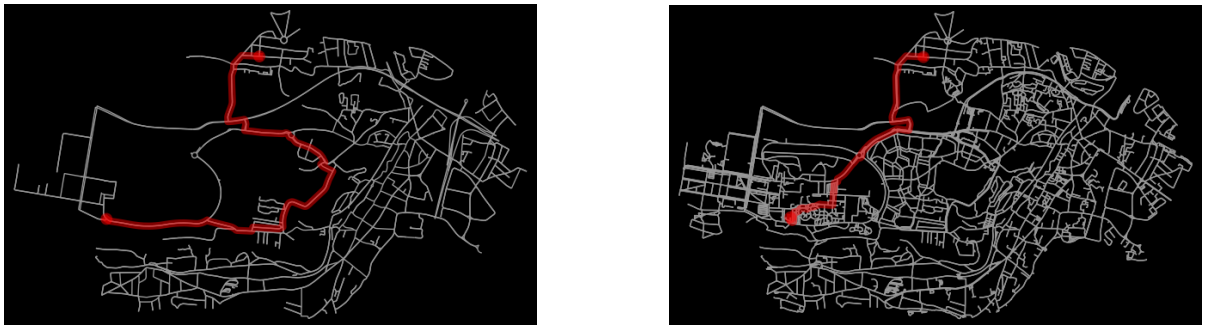


FIGURE 2 – Calcul de temps sur différent graphe



(a) Avec sa Ferrari

(b) Pedibus cum jambis

FIGURE 3 – Le chemin le plus court de chez Aimé jusqu'à l'école

4 Q 9 : OSMnx (c'est la carte)

Nous avons rencontré des difficultés à télécharger la carte de toute la France, nous nous sommes donc concentré sur la ville qui nous est chère : Palaiseau. Et nous avons choisi de chercher le chemin le plus court de chez Aimé jusqu'à l'école

Nous avons transformé les caractéristiques suivantes à notre modèles

- Chaque route est délimitée par deux carrefours
- La distance de la route est celle à parcourir d'un carrefour à l'autre
- La puissance d'une route est la vitesse maximale de cette portion

5 Q 11 : Démonstration

A démontrer

La puissance minimale pour couvrir un trajet dans le graphe G est égale à la puissance minimale pour couvrir ce trajet dans l'arbre A_{min}

Notation

- A est un arbre couvrant de G
- A_{min} est l'arbre couvrant minimal de G
- u, v, i_k des nœuds de G
- $t_G(u, v)$ un trajet dans G de u à v

- $t_A(u,v)$ l'unique trajet (car A est un arbre) dans A de u à v
- On pourra écrire de manière équivalente $t_G(u,v)$ et $i_1^G - i_2^G - ... - i_{n-1}^G - i_n^G$ avec $u = i_1^G$ *et* $v = i_n^G$
- $p(t_G(u,v))$ la puissance minimale du trajet $t_G(u,v)$
- $p(A)$ est la somme des puissances de l'arbre A

Soit G un graphe, soit A_{min} un arbre couvrant minimal de G, soit u, v des nœuds de G

Par l'absurde, supposons qu'il existe un trajet $t_G(u,v)$ tel que $p(t_G(u,v)) < p(t_{A_{min}}(u,v))$

Notons : $t_{A_{min}}(u,v) = i_1^{A_{min}} - ... - i_n^{A_{min}}$. Alors $\exists\ k \in [1, n-1]\ tq\ p(i_k^{A_{min}} - i_{k+1}^{A_{min}}) = p(t_{A_{min}}(u,v))$

En rompant l'arête $i_k^{A_{min}} - i_{k+1}^{A_{min}}$ dans l'arbre A_{min}, on obtient 2 arbres. Le nœud u est dans un arbre et v est dans l'autre car l'unique chemin les reliant dans A_{min} n'existe plus. Notons A_u l'arbre contenant u et A_v l'arbre contenant v.

Notons : $t_G(u,v) = i_1^G - ... - i_m^G$ avec $u = i_1^G \in A_u$ et $v = i_m^G \in A_v$. Donc $\exists\ l \in [1,m], i_l^G \in A_u\ et\ i_{l+1}^G \in A_v$

Remarque :

$$\begin{aligned} p(i_l^G - i_{l+1}^G) &\leq p(t_G(u,v)) \\ &< p(t_{A_{min}}(u,v)) \text{ par hypothèse} \\ &= p(i_k^{A_{min}} - i_{k+1}^{A_{min}}) \end{aligned}$$

En ajoutant l'arête $i_l^G - i_{l+1}^G$, on relie les arbres A_u et A_v pour former un arbre couvrant A' de G. De sorte que :

$$\begin{aligned} p(A') &= p(A_{min}) + p(i_l^G - i_{l+1}^G) - p(i_k^{A_{min}} - i_{k+1}^{A_{min}}) \\ &< p(A_{min}) \end{aligned}$$

Ce qui est absurde car par hypothèse A_{min} est l'arbre couvrant minimal. On a donc démontré le l'affirmation.

6 Démonstration Complexité Union-Find Path Compression

Démontrons que la complexité amortie de l'Union-Find Path Compression est en O(1).
Pour commencer calculons la complexité d'exécuter `m` fois le Find Path Compression sur un arbre à `n` noeuds. Comme notre objectif est d'avoir un arbre de hauteur 2, considérons que `find` appliqué à la racine ou à un enfant de la racine est en complexité constante. Pour les autres noeuds, la complexité du `find` est égal à *sa hauteur* − 2, car il suffit de remonter à un fils de la racine. Au début la borne sup de la hauteur de l'arbre est n.
Ainsi, lorsque le find s'exécute k fois, alors il y a k compressions, et la borne sup de la hauteur de l'arbre diminue de k. (Attention, on parle bien de borne sup de la hauteur dans le cas de l'initialisation avec l'Union-Find).
Si nous exécutons m fois le find, avec m>n, et que nous notons $\forall i \in [1,m], k_i$ le nombre de fois que s'éxecute find à la i ème requête, alors la complexité amortie se note $\frac{1}{m} \times \sum_{i=1}^{m} k_i$. Comme la hauteur de l'arbre est au minimum de 2, alors $\sum_{i=1}^{m} k_i < n - 2$. D'où une complexité $O(1)$

Pour le cas de l'Union-Find Path Compression, au lieu de regarder la borne sup de la hauteur de l'arbre, il suffit de regarder la somme de la borne sup des arbres. Etant donné que cette somme reste inchangée par Union, la démonstration d'avant reste vrai.

Nous avons pu trouvé des articles de recherches qui ont trouvé des démonstrations plus rigoureuses (Wu et Otoo 2005).

7 Problème du sac à dos

7.1 Pourquoi nous n'avons pas choisit d'utiliser la programmation dynamique pour résoudre le problème

Une manière assez naturelle de résoudre le problème du sac à dos est d'utiliser la programmation dynamique. En utilisant la programmation dynamique, on cherche à calculer la quantité $C(b,i)$ qui correspond à l'utilité maximale que l'on peut atteindre avec un budget b et les i premiers trajets. Celle ci peut facilement s'exprimer et est donnée par :

$$C(b,i) = \max(C(b - \text{coût du i-ème trajet}, i-1), +\text{utilité du i-ème trajet}), C(b, i-1))$$

Avec b toujours inférieur à la contrainte budgétaire totale. Si on note B le budget total et N le nombre de trajets que l'on considère, on doit finalement remplir une matrice de taille $N \times B$. Le calcul de chaque coefficient se fait en temps constant, on trouve donc finalement une complexité $O(N \times B)$. Ceci n'est pas raisonnable au vu de la taille de notre budget et du nombre de trajet possibles (surtout au vu de la taille du budget). Nous avons pensé à une solution qui était d'incrémenter le budget par des valeurs plus grandes que 1 (de 100k $ en 100k $ par exemple) mais il aurait fallut adapter l'algorithme classique et la formule de récurrence et le fait que quelques camions coûtaient un prix nettement inférieur (environ 10k $) compliquait encore plus la chose. Nous avions d'autres idées d'algorithmes paraissant assez efficaces dès le début et avons donc décidé de délaisser cette méthode qui serait très inefficace si nous ne l'adaptions pas.

Nous allons dans la suite vous présenter les différentes méthodes que nous avons choisis d'utiliser pour résoudre le problème qu'il nous était posé.

7.2 Approche naïve

L'approche la plus naïve est de parcourir tous les sous-ensembles non vide de routes, pour voir si la contrainte de budget est correcte et pour en calculer l'utilité. On cherche alors l'utilité maximale.
Cette solution est sûre de fonctionner et trouver la meilleure allocation, mais elle est en complexité $O(2^{nb\_routes})$.

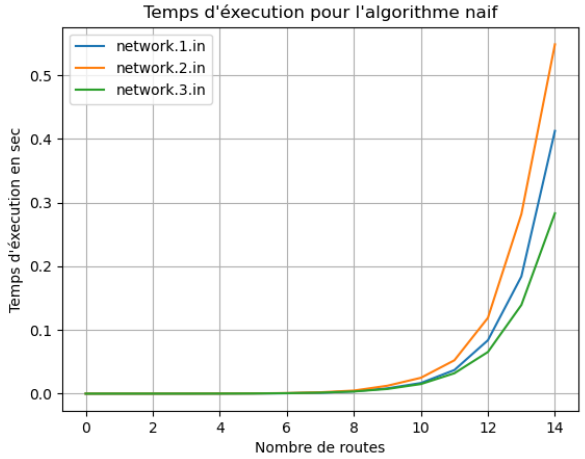


FIGURE 4 – Simulation de l'algorithme naïf sur des petits graphes

7.3 Approche grâce à l'algorithme glouton

Nous avons ensuite décidé d'utiliser un algorithme glouton afin de résoudre notre problème, l'avantage de ce type d'algorithme est que sa complexité est en complexité $O(\texttt{nb\_routes} \times \log(\texttt{nb\_routes}))$ et bien qu'il ne donne pas la solution optimale, il permet de l'approcher relativement bien en un temps très raisonnable. Le principe est simple, on calcule le prix du camion le moins cher qui peut desservir un trajet et on ordonne tous les trajets suivant la valeur $\frac{\text{utilité du trajet}}{\text{prix du camion}}$. On ajoute donc les routes par ordre décroissant d'utilité jusqu'à saturer la contrainte budgétaire (il est possible que la contrainte soit saturée à partir d'un certains moment, mais on continue de parcourir la liste en essayant d'ajouter des trajets qui ont un moins bons ratio utilité/prix mais qui coûtent potentiellement moins cher et peuvent donc être rajoutés à la solution du problème). Dans la prochaine sous-partie, nous allons prouver la complexité de l'algorithme glouton et montrer que l'ont peut obtenir une borne théorique d'utilité à la solution optimale.

7.4 Démonstration Glouton

Ce qui est bien avec l'algorithme est qu'il est en $O(n \times log(n))$. Mais il est possible de trouver des configurations où l'algorithme glouton ne nous retourne pas la meilleur allocation, par exemple :

Nom de la route	$route_1$	$route_2$	$route_3$
Utilité	15	8	8
Prix	3	2	2
Ratio	5	4	4

Si dans l'exemple ci-dessus nous avons un budget de 4, l'algorithme glouton ne prend que $route_1$ et nous donne donc une utilitée de 15. Ce qui n'est pas l'allocation optimale, car on peut prendre $route_1$ et $route_2$ pour ce même budget et avoir une utilité de $8 + 8 = 16$.

En réalité, quand on se trouve dans un cas similaire à ci-dessus, c'est à dire qu'il n'y a que 2 camions différents (seulement 2 prix différents), on peut trouver une condition suffisante pour que l'algorithme glouton fonctionne. En effet, il suffit de retirer le dernier camion de prix maximal pour voir si le remplacer avec 2 camions de l'autre prix (quittent à prendre ceux avec les ratios les plus élevés) satisfait toujours le budget ou non. On prend alors l'allocation avec l'utilité la plus grande et on réitère jusqu'à ce que le fait de remplacer n'est plus possible ou n'est plus optimal. Cela est vrai car il ne sert à rien de retirer un camion de prix minimal, on ne pourrait alors le remplacer que par 1 autre camion dont le ratio est inférieur grâce au classement de glouton. Cette vérification se fait en temps linéaire sur le nombre de routes utilisant les camions de prix maximal.

Voici un autre exemple de ce cas avec un budget de 12 :

Nom de la route	$route_1$	$route_2$	$route_3$	$route_4$	$route_5$	$route_6$
Utilité	50	50	27	27	27	27
Prix	5	5	3	3	3	3
Ratio	10	10	9	9	9	9

7.5 Optimisation de la solution gloutonne grâce à une démarche de type "Monte - Carlo"

Avant de commencer, nous signalons que nous avons délibérément choisit de faire un code qui ne marche pas au cas où le budget est assez grand pour pouvoir desservir tous les trajets et qui ne marche pas si le nombre de routes est trop petit (marche dès lors qu'il reste tout le temps 20 trajets non alloués quelque soit la solution proposée qui respecte la contrainte budgétaire), et ce afin d'alléger le code, car l'algorithme glouton gère déjà très bien ces cas.

Nous savons que l'algorithme glouton renvoie des solutions approchées de la solution optimale, cependant nous savons qu'il est parfois possible d'améliorer encore ce résultat. Nous utilisons donc une méthode de type Monte Carlo afin d'améliorer la solution trouvée. Le principe est simple, on part de la solution que renvoie glouton et on va choisir des camions à enlever au hasard. Ensuite on va choisir d'autres camions à ajouter à la solution jusqu'à saturer la contrainte budgétaire à nouveau, on garde cette nouvelle solution uniquement si l'utilité générée par celle ci est supérieure à celle générée par l'ancienne solution. Le nombre de camion à enlever est choisit aléatoirement suivant une loi uniforme comprise entre 0 et 20 ce choix est critiquable et il existe différentes heuristiques pour éviter le problème suivant : Il est possible que la solution dont nous disposons soit localement optimale et que si l'on n'enlève pas assez de camions, on ne puisse jamais trouver une meilleure solution car on ne pourra pas améliorer l'utilité de la solution sans enlever plusieurs camions en même temps de notre solution.

Exemple 7.1. Considérons un budget de 10, et trois routes (1), (2) et (3) ayant respectivement des couples (utilité, prix) de : (10, 5), (10, 5), (35, 6). Il est évident que la solution optimale pour maximiser l'utilité est d'allouer tout son budget pour desservir le trajet (3) bien qu'il reste un excédant d'argent à la fin. Considérons la solution qui consiste à desservir les routes (1) et (2), il est alors clair que si on désaloue uniquement une des deux routes on ne pourra jamais desservir le trajet (3) sans dépasser le budget et donc trouver la solution optimale de ce problème.

On arrête d'exécuter ce processus au bout d'un certains temps choisit préalablement et on renvoie la dernière solution trouvée.

N.B. A noter que dans notre code nous avons aussi appliqué cet algorithme avec une solution de départ générée en ajoutant les trajets dans l'ordre où ils venaient jusqu'à saturer la contrainte budgétaire. Cependant cette méthode partant de solutions loin d'être optimales ne générait pas d'aussi bon résultats que le glouton même après avoir tourné 3 à 4 fois plus longtemps que l'algorithme glouton. Ce qui nous à poussé à devoir partir d'une solution générée par l'algorithme glouton.

7.6 Pistes d'améliorations de notre algorithme d'optimisation aléatoire

Nous avons eu différentes idées d'heuristiques pour améliorer notre processus aléatoire cependant nous n'avons pas eu le temps de les mettre en pratique pour les essayer ni de réfléchir à si elles pouvaient vraiment permettre en théorie de converger plus rapidement vers la solution optimale. Voici quelques une de ces pistes :

- Classer la liste des routes non allouées de manière décroissante en fonction de la quantité $\frac{\text{utilité}}{\text{prix}}$ et choisir l'indice des camions que l'on ré alloue suivant une loi de probabilité décroissante. (loi de Paretto ou de poisson par exemple même si dans les fait ces loi tendent très rapidement vers une probabilité de 0 pour les grands nombre et qu'on aurait plutôt utilisé une loi qui avait une décroissance linéaire). On aurait ainsi tendance à essayer plus souvent d'allouer des routes avec un meilleur rapport d'utilité, car il semble cohérent que les routes avec un mauvais rapport d'utilité ne rentreront sûrement pas dans la solution finale. Cette heuristique augmenterais cependant la complexité de l'opération de désallocation d'un trajet (auparavant on insérait la route désallouée à la fin ce qui se fait en temps constant, désormais il faudrait insérer la route désallouée au bon endroit pour que la liste reste triée, ce qui se fait en $O(n)$ (n étant la longueur de la liste). Cependant on peut minimiser cette complexité en disant que nous utilisons la méthode `pop()` pour enlever les trajets alloué et les passer dans la liste des trajets non allouées. Or nous utilisons cette méthode sans que notre liste ait une structure de pile donc la complexité de `pop()` est $O(n)$.
- Il existe également des heuristiques permettant d'éviter de rester bloquer autour d'une solution localement optimale comme nous l'avons évoqué plus haut, par exemple l'euristique Simulated annealing ou l'algorithme Metropolis-Hasting.

Bibliographie

[WO05] Kesheng Wu et Ekow Otoo. "A simpler proof of the average case complexity of union-find with path compression". In : (2005).