ENSAE Paris

Rapport

Projet de programmation - Optimisation d'un réseau de livraison

Élèves :
Iokanaan Belfis
Augustin Cablant

Enseignant :
Patrick Loiseau

19 juin 2023

Table des matières

1 Description du problème : optimisation d'un réseau de livraison

On considère un réseau routier constitué de villes et de routes entre les villes. L'objectif du projet va être de construire un réseau de livraison pouvant couvrir un ensemble de trajets entre deux villes avec des camions. La difficulté est que chaque route a une limite inférieure de puissance, c'est-à-dire à dire qu'un camion ne peut emprunter cette route que si sa puissance est supérieure ou égale à la limite inférieure de puissance de la route. Il faudra donc déterminer pour chaque couple de villes si un camion d'une puissance donnée peut trouver un chemin possible entre ces deux villes ; puis chercher à optimiser la flotte de camions qu'on achète en fonction des trajets à couvrir. On représente le réseau routier par un graphe non orienté $G = (V, E)$. L'ensemble des sommets V correspond à l'ensemble de villes. L'ensemble des arêtes E correspond à l'ensemble des routes existantes entre deux villes. Chaque arête $e \in E$ est associée à une valeur (ou poids) $\overline{p}_e 0$ qu'on appelle puissance minimale de l'arête e, qui correspond à la puissance minimale d'un camion pour qu'il puisse passer sur la route e. Chaque arête $e \in E$ est également associée à une deuxième valeur de $d_e > 0$ correspondant à la distance entre les deux villes sommets sur l'arête e.
On considère un ensemble T de trajets, où chaque trajet t est un couple de deux villes distinctes, i.e., $t = (v, v')$ où $v \in V$, $v' \in V$, $v \neq v'$. L'ensemble T représente l'ensemble des couples de villes (v, v') pour lesquelles on souhaite avoir un camion qui puisse faire le transport entre v et v'. Notez que le graphe n'est pas orienté et on ne distingue pas la direction du trajet. À chaque trajet t est également associé un profit (ou utilité) $u_t 0$, qui sera acquis par la compagnie de livraison si le trajet t est couvert.
Enfin, le transport est fait par des camions. Chaque camion (noté K) a une puissance p et coûte un prix c. Le transport sur un trajet $t = (v, v') \in T$ sera possible si et seulement si on peut trouver dans le graphe G un chemin allant de vv' sur lequel la puissance minimale de chaque arête est inférieure où égale à p (i.e., le camion a une puissance suffisante pour passer partout sur le chemin). On dit alors que le trajet t peut être couvert par le camion K considéré.

Le projet contient deux parties :
Partie 1 : calcul de la puissance minimale pour un trajet. Dans cette partie, qui couvre environ les trois premières séances, on s'intéresse au problème de calculer la puissance minimale nécessaire d'un camion pour un trajet $t \in T$ (ainsi que le chemin associé).

Partie 2 : optimisation de l'acquisition de camions. Dans cette deuxième partie, on cherche à calculer quels camions acheter pour optimiser le profit des trajets couverts.

Notations :
On notera $V = |V|$ le nombre de sommets du graphe G (aussi souvent appelé n) ; $E = |E|$ le nombre d'arêtes du graphe G (aussi souvent appelé m) ; et $T = |T|$ le nombre de trajets que l'entreprise souhaite couvrir.
Lien vers notre répertoire Github : https ://github.com/ibelfis/rendu$_f inal$

2 Séance 1 : 10/02

Question 1. Compléter la définition de la classe Graph en implémentant en particulier la méthode "add edge" et écrivez une fonction "graph from file" qui permet de charger un fichier.

Il s'agit simplement de convertir le type *str* obtenu après ouverture du fichier en type *Graph* en suivant la notice de lecture du fichier.

Question 2. Écrire une méthode "connected components set" qui trouve les composantes connectées du graphe $\mathcal{G}$ et analyser la complexité de l'algorithme en fonction de V et E.

On implémente classiquement un parcours en profondeur pour discriminer les composantes connexes d'un graphe. Le principe est de partir d'un sommet, et d'explorer tous ses voisins récursivement, en marquant au fur et à mesure les sommets. Codé de manière récursive, il est donc de complexité $O(V + E)$: chaque arc et chaque noeud est parcouru une fois.

Question 3. Écrire une fonction "get path with power" qui prend en entrée une puissance de camion p et un trajet t et qui décide si un camion de puissance p peut couvrir le trajet t (et retourne, si c'est possible, un chemin admissible pour le trajet t).

On commence par vérifier qu'il existe bien un chemin entre les deux points en vérifiant que le graphe est bien connexe. On implémente ensuite l'algorithme de Djikstra afin de trouver le chemin de puissance minimale. Cet algorithme nous permet de calculer le plus court chemin à partir de notre source *src* vers notre destination *dest* dans un graphe orienté pondéré par des réels positifs. Nous terminons en vérifiant que le chemin de puissance minimale peut être couvert avec la puissance initiale que nous avons rentré. Codé ainsi, il est donc de complexité $O((E+V)\log(V))$ où V représente le nombre de sommets et E le nombre d'arêtes du graphe.

Question 4. Modifier la fonction de lecture des fichiers "graph from file" pour lire la distance d'une arête qui est optionnelle.

Même type de manipulations que pour la question 1.

Question 5. Question 5 (bonus). Modifier la fonction de la question 3 pour qu'elle retourne, lorsque le trajet peut être couvert par le camion, le plus court (au sens de la distance d) chemin admissible pour le trajet t.

On adapte l'algorithme de Djikstra utilisé en question 3 pour pondérer les arêtes par la distance plutôt que par la puissance.

Question 6. Écrire une fonction "min power" qui calcule, pour un trajet t donné, la puissance minimale d'un camion pouvant couvrir ce trajet. La fonction devra retourner

le chemin, et la puissance minimale.

Nous raisonnons en deux étapes :

- **Etape 1 :** Nous appliquons *get_path_with_power*, qui nous retourne le chemin optimal dans le graphe pour le trajet t dans le cas où il existe (ie. dans la même composante connexe et de puissance inférieure à p).
- **Etape 2 :** Nous parcourons ce chemin pour sommer les puissances de chaque arête parcourue, on obtient ainsi la puissance minimale du chemin.

Question 7(bonus). Implémenter une représentation graphique du graphe, du trajet, et du chemin trouvé. On pourra utiliser une bibliothèque pour cela, par exemple graphviz, vous pouvez l'installer sur le SSP Cloud avec le script install graphviz.sh, à exécuter dans un terminal.

Question traitée, mais les erreurs au lancement de *graphviz* nous empêchent d'obtenir le résultat. D'après l'analyse lancée lors d'une tentative de réinstallation du mondule, l'installateur *conda* a relevé des erreurs de compatibilité de modules, dont pourraient provenir le problème. Or nous avons travaillé sur deux machines aux environnements similaires, et n'avons donc pas eu de moyen de tester le code, cependant laissé dans le projet.

Question 8. Tester les algorithmes sur les graphes fournis. Implémenter de nouveaux tests des fonctions développées.

On s'appuie pour la rédaction des nouveaux tests sur les modèles proposés dans l'énoncé. On configure les tests en raisonnant « à la main » sur les plus petits réseaux proposés.

3 Séance 2 : 24/02

Question 10. Estimer le temps nécessaire pour calculer (à l'aide du code développé dans la séance 1) la puissance minimale (et le chemin associé) sur l'ensemble des trajets pour chacun des fichiers routes.x.in donnés.

Avec le code de la séance 1, la durée d'obtention de l'ensemble des trajets est déraisonnable. Dès 10 000 entrées comme pour *route.1.in* on arrive à un ordre de grandeur de plusieurs heures de calcul.

On voit bien que ce temps n'est pas raisonnable. On va maintenant chercher à l'améliorer significativement. Une clef pour améliorer est l'observation suivante : soit $\mathcal{A}$ un arbre couvrant de poids minimal du graphe $\mathcal{G}$. Alors, la puissance minimale pour couvrir un trajet t dans le graphe $\mathcal{G}$ est égale à la puissance minimale pour couvrir ce trajet t dans le graphe $\mathcal{A}$. On rappelle qu'un arbre est un graphe acyclique connecté (qui a la forme classique sous forme d'ancêtres et descendants). Un arbre couvrant de poids minimal (minimum spanning tree en anglais) est un arbre construit avec des arêtes du graphe initiale (en n'en retenant que certaines) qui couvre tous les sommets du graphe et dont la somme des poids des arêtes (les $\overline{p}$ pour nous) est minimale parmi tous les arbres couvrants. Il a exactement $V1$ arêtes (comme tous les arbres ayant V sommets).

Question 11(bonus). Prouver l'affirmation ci-dessus.

Lemme : Soit A un arbre couvrant de poids minimal du graphe G. Alors la puissance minimale pour couvrir un trajet t dans le graphe G est égale à la puissance minimale pour couvrir ce trajet t dans le graphe A.

Notations

1. Soit G un graphe connexe à k noeuds. On notera $G.n = \{n_j\}_{j\leq k}$ l'ensemble des noeuds de G, et $G.e = \{(n_i, n_j)_{i,j\leq k}$ l'ensemble des arêtes de G. On définit sur les graphes l'union : $A \cup B$ est tel que $A.n \cup B.n$ et $A.e = B.e$.
2. Soit A un arbre couvrant de poids minimal de G, que l'on considère comme un objet de type graphe, avec des notations similaires $A.n$ et $A.e$.
3. Soient n_i, n_j deux noeuds d'un graphe F, alors on note $C_F^{i,j}$ un trajet de puissance minimale de n_i vers n_j dans F, que l'on code sous la forme d'un graphe, avec des notations similaires $C_F^{i,j}.n$ et $C_F^{i,j}.e$.
4. Soit $P_t : t = (n_i, n_j) \to P_a(t)$ qui à un trajet associe sa puissance minimale dans G, et $P_g : G \to \sum_{(n_i,n_j)\in G.e} P_a((n_i, n_j))$ qui à un graphe associe la somme des puissances minimales des trajets dans G. Cette fonction donne en particulier le poids d'un trajet d'après (3), en voyant le trajet comme un sous-graphe, en particulier un graphe.

Preuve Raisonnons par l'absurde.

On se donne $i, j \leq k$, tels que $P_g(C_G^{i,j}) < P_g(C_A^{i,j})$, *ie.* il existe dans G un trajet de puissance minimale inférieure que la puissance minimale du trajet optimal de A.

Posons $A^* = \bigcup_{\alpha,\beta \leq k} C^{\alpha,\beta}_{A \cup C_G^{i,j}}$.

Alors A^* est un arbre couvrant (1) qui contredit la minimalité de A (2) : $P_g(A^*) < P_g(A)$.

1. **Arbre couvrant :**
   - *(Couvrant)* Immédiat : tous les noeuds de G sont dans A^* par construction.
   - *(Arbre)* On sait qu'un arbre est un cas particulier de graphe acyclique connexe. Ici la connexité de A^* est évidente par construction.
     Reste à montrer l'acyclicité de A^* : si par l'absurde il existe un cycle dans A^*, et sachant l'acyclycité de A comme arbre, cela signifie que le cycle est composé d'arêtes de A et d'arêtes de $C_G^{i,j}$.
     Par définition d'un cycle, il existe donc deux noeuds n_γ et n_δ reliés par au moins deux chemins dans A^*, dont au moins un emprunte une arête de $C_G^{i,j}.e - A.e$, ce qui signifie qu'il est une étape dans le chemin de n_i vers n_j.
     Or si $C_{A^*}^{\gamma,\delta}$ ne passe pas par une arête de $C_G^{i,j}.e - A.e$, on a une contradiction avec la minimalité du trajet de n_i vers n_j, puisqu'un autre trajet de puissance inférieure convient.
     Donc par optimalité de tous les trajets dans la construction de A^* on a que n_γ et n_δ sont reliés par un unique trajet, ce qui contredit l'hypothèse de cyclicité.
2. **Minimalité :** On a $P_g(A^*) - P_g(C_{A^*}^{i,j}) = P_g(A) - P_g(C_A^{i,j})$. Or $P_g(C_{A^*}^{i,j}) = P_g(C_G^{i,j})$, et $P_g(C_G^{i,j}) < P_g(C_A^{i,j})$ par hypothèse.
   De là, $P_g(A^*) = P_g(A) + P_g(C_G^{i,j}) - P_g(C_A^{i,j})$ implique $P_g(A^*) < P_g(A)$, d'où la minimalité.

Question 12. Ecrire une fonction kruskal qui prend en entrée un graphe au format de la classe Graph (e.g., g) et qui retourne un autre élément de cette classe (e.g., g mst) correspondant à un arbre couvrant de poids minimal de g. Analyser la complexité de l'algorithme implémenté.

On implémente l'algorithme de Kruskal, dont la complexité est quasi-linéaire, à condition d'implémenter correctement une structure d'union-find.
On implémente donc une structure d'union-find en arbre équilibré, qui permet d'accéder au nom de la classe (opération $find$) en coût $O(logV)$, et de joindre deux classes ($union$) avec un coût dominé par un $find$.
De là finalement il vient que la complexité de l'algorithme de Kruskal s'obtient en sommant celle du tri des arrêtes (classiquement en $O(E * logE)$ avec un quicksort ou un trifusion par exemple, la complexité implémentée en Python), et celle de la boucle de parcours des arrêtes, qui réalise $find$ puis $union$, ce qui donne un $O(E * logV)$.
Bilan : on obtient finalement un algorithme en $O(E * logE) + O(E * logV)$ donc certainement en $O(E * logE)$ considérant que $E >> V$, l'objectif de Kruskal étant précisément d'obtenir $E = V - 1$.

Question 13. Implémenter des tests de la fonction ci-dessus sur les petits graphes fournis.

On s'appuie pour la rédaction des nouveaux tests sur les modèles proposés dans l'énoncé. On configure les tests en raisonnant « à la main » sur les plus petits réseaux proposés.

Question 14. Ecrire une nouvelle fonction basée sur l'arbre couvrant de poids minimal qui calcule, pour un trajet t donné, la puissance minimale d'un camion pouvant couvrir ce trajet (et le chemin associé).

Comme suggéré par l'énoncé (question 11), on applique l'algorithme de Kruskal au graphe une unique fois, avant de calculer les chemins minimums (par l'application de la fonction de la question 6) dans l'arbre mimimum de poids couvrant obtenu.
Le coût de calcul de Kruskal en $O(E * logE)$ d'après la question 12 est ainsi amorti par l'économie réalisée en appliquant Djisktra (classiquement de complexité $O((V+E)*logV)$, devient avec $E \approx V$ simplement $O(V * logV)$).

Question 15. Analyser la complexité de la solution précédente théoriquement, et reprendre en pratiques les estimations de temps de la question 10 pour cette nouvelle fonction. Comparer.

Comme vu à la question 14, la complexité est de l'ordre de $O(V \log(V))$. Comme le temps est trop long, on se référera à la question 17 pour avoir l'estimation du temps de la création des fichiers routes.out.

4 Séance 3 : 10/03

Question 16. Ecrire un programme qui calcule, pour un trajet t donné, la puissance minimale d'un camion pouvant couvrir ce trajet, tel que le temps de pré-processing soit $O(VlogV)$ et ensuite le temps de réponse pour chaque trajet soit $O(logV)$. Justifier cette complexité.

L'algorithme de Kruskal implémenté à la séance précédente donne bien un temps de pré-processing en $O(V * logV)$.
L'idée est ensuite de se servir de la structure d'arbre donnée par Kruskal, en se basant sur le fait que le chemin optimal dans le graphe initial est l'unique chemin existant dans l'arbre couvrant de poids minimal. En raisonnant de la même manière que pour l'implémentation de la classe $Union - Find$, c'est à dire en regardant la profondeur en fonction d'un noeud racine fixé arbitrairement.

Question 17. Tester l'algorithme ci-dessus sur les exemples fournis et calculer le temps d'exécution.

Comme demandé, nous avons créé les fichiers routes.out et estimé le temps nécessaire pour effectuer la tâche. Nous avons aussi ajouté le nombre de chemins calculés afin de mesurer l'ampleur des calculs.

Fichier route n° :	1	2	3	4	5	6	7	8	9
Temps de calcul (s)	0.001	5.2	115.3	120.5	28.5	146.3	174	140.6	114.5
Temps de calcul (min)	0	0.1	1.9	2	0.475	2.9	2.4	2.3	1.9
Nombre de chemins calculés	140	100k	500k	500k	100k	500k	500k	500k	500k

```
(base) iokanaanbelfis-simon@MacBook-Pro-Iokanaan GitHub % /Users/iokanaanbelfis-simon/opt/anaconda3/bin/python /Users/iokanaanbelfis-simon/Documents/GitHub/projetpython2023/ens
ae-prog23-main/delivery_network/routeout.py
Fichier route 1 créé en 0.0007738919999999982
None
Fichier route 2 créé en 5.153488363000001
None
Fichier route 3 créé en 115.29882942100001
None
Fichier route 4 créé en 120.49441295399998
None
Fichier route 5 créé en 28.47387119800004
None
Fichier route 6 créé en 146.28071862500002
None
Fichier route 7 créé en 173.91036564800004
None
Fichier route 8 créé en 140.59460575799994
None
Fichier route 9 créé en 114.48718882999992
None
(base) iokanaanbelfis-simon@MacBook-Pro-Iokanaan GitHub %
```

5 Séance 4 : 16/03, 30/03, 13/04

Question 18. Écrire un programme qui retourne une collection de camions à acheter ainsi que leurs affectations sur des trajets, pour maximiser la somme des profits obtenus sur les trajets couverts. Analyser la solution et tester le programme sur les différents fichiers network.x.in / routes.x.in et avec, pour chacun, les différents catalogues trucks.x.in.

On a tout d'abord besoin de fonctions renvoyant le camion optimal pour couvrir un chemin de puissance donnée. On procède en deux étapes.
Tout d'abord, on sélectionne tous les camions de puissance supérieure ou égale à la puissance requise (méthode dichotomique dans la liste des camions triée par puissance en coût quasi linéaire), puis on choisit parmi eux le camion de coût minimal en coût linéaire.
Le coût résultant est celui du tri du catalogue des camions, en $O(C * logC)$, si C est le nombre de camions du catalogue.

- **Idée 1 (algorithme glouton) :**
  Principe : il s'agit de réaliser les chemins par ordre d'utilité décroissante jursqu'à épuiser le budget.
  Complexité : tri des chemins en $O(T * logT)$, si T est le nombre de trajets repertoriés, et réalisation d'un chemin en calculant le camion optimal pour le réaliser, en $O(C * logC)$. On aura donc au pire un coût en $O(T * C * logC)$.
  Limite : nous ne disposons d'aucune garantie d'optimalité du résultat.
  Intérêt : la complexité est raisonnable.

- **Idée 2 (force brute) :**
  Principe : on teste toutes les combinaisons possibles, et on renvoie la meilleure en terme d'utilité cumulée.
  Implémentation : on doit parcourir toutes les possibilités. Un raisonnement combinatoire montre qu'il y en a 2^T avec les notations précédentes (en effet, on choisit T fois de prendre (0) ou de ne pas prendre (1) un trajet). De là on peut identifier une combinaison à son codage binaire (exemple : 111 sera la combinaison correspondant à la réalisation de tous les trajets, si la liste des trajets est de longueur 3). On parcours ensuite toutes les combinaisons facilement sachant que la fonction *bin* de Python renvoie le codage en binaire d'un nombre, en réalisant une bijection entre $[|0, 2^T|]$ et $\{(i_1 i_2 ... i_T)_{i_j \in \{0,1\}}\}$. Il s'agit ensuite de calculer l'utilité cumulée de chaque combinaison, comme somme des utilités de chaque trajet réalisé, et de calculer le coût de chaque combinaison, comme somme des coûts des trajets réalisés. On garde la solution qui réalise le maximum d'utilité sous la contrainte d'un coût inférieur à notre budget. Complexité : $O(2^T)$, la pire imaginable pour ce type de problème (c'est la solution naïve).
  Intérêt : on calcule une solution exacte.
  Limite : la complexité rend la méthode irréaliste sur un graphe de taille intéressante.

Question 19. (bonus). Implémenter une visualisation de l'allocation sur des petits exemples (par exemple network.0.in et network.1.in).
Idem question 7.

Question 20(bonus). Reprendre la question 18 dans un cas plus réaliste suivant : on suppose maintenant que
(i) Chaque arête a une probabilité ϵ de se "casser" (i.e., la route est bloquée), indépendamment des autres arêtes. On acquiert le profit correspondant à un trajet seulement si aucune arête sur le chemin entre les deux villes n'est cassée.
(ii) Le passage d'un camion sur un trajet t donne un coût (de carburant) proportionnel à la distance totale du chemin couvrant ce trajet (i.e., la somme des distances sur les arêtes de ce chemin).
On cherche à maximiser le profit en espérance du *(i)* moins le coût de carburant du *(ii)*. On pourra prendre un coût de carburant de 0.01 par unité de distance et $\epsilon = 0.001$ pour les tests.

Non traitée.

CONCLUSION

Nous sommes parvenus à couvrir l'ensemble des questions non optionnelles et avons pu réfléchir à d'autres qui l'étaient. Nos résultats sont plutôt concluants, puisque nous sommes parvenus à obtenir des résultats (en temps fini) à l'issue de la séance 4, alors qu'à la fin de la séance 1 nous avions peu d'espoir. Nous avons ainsi pu nous confronter à la difficulté de la programmation : écrire des algorithmes optimaux et les combiner entre eux, tout en s'assurant qu'ils tournent en temps fini.
Ce projet nous a permis de voir ou revoir certains algorithmes très connus et très utilisés en programmation, ce qui sera un véritable avantage pour nos années futures à l'ENSAE.

6 Références

Références

[1] Les documents de la communauté Python.
Lien : https ://peps.python.org/pep-0008/

[2] Donald E. Knuth Kenneth Reitz Tanya Schlusser, The Hitchhiker's Guide to Python !, https ://docs.python-guide.org/writing-great-python-code.

[3] Wikipédia pour divers algorithmes : Djikstra, Problème du sac à dos, Glouton.

[4] Les pages de Xavier Dupré sur son site :
http ://www.xavierdupre.fr/app/ensae_teaching_cs/helpsphinx3/index.html