ENSAE Paris

Projet informatique ENSAE 1A

Adam El Bernoussi, Mathieu Lepage

Avril 2023

Le but de notre projet informatique est de modéliser et d'optimiser un réseau de livraison. On veut construire un réseau de livraison qui permet de couvrir un ensemble de trajets entre deux villes à l'aide de camions. Chaque route demandant une puissance minimum, un camion ne peut emprunter cette route que s'il est doté d'une puissance supérieure à celle de la route.

On modélisera le réseau routier par un graphe non orienté où l'ensemble des sommets correspond à des villes et l'ensemble des arêtes à l'ensemble des routes existantes entre deux villes. A chaque arête, on lui associe un poids qui modélise la puissance minimale nécessaire à un camion pour traverser la route. Chaque arête est associée à une deuxième valeur, qui détermine la distance entre deux villes.

On notera t un trajet, qui est représenté par un couple de deux villes distinctes. Chaque trajet t est également associé à un profit que gagne l'entreprise si le trajet est couvert.

Description du projet

Partie 1 : Calcul de la puissance minimale pour un trajet

Dans cette partie, nous allons déterminer la puissance minimale nécessaire à un camion pour parcourir un trajet donné. Nous utiliserons pour cela l'algorithme de Dijkstra, qui permet de trouver le chemin le plus court entre deux sommets d'un graphe pondéré.

Partie 2 : Optimisation de l'acquisition de camions

Dans cette partie, nous chercherons à optimiser l'acquisition de camions pour couvrir tous les trajets avec un coût minimum.

Partie 1 : Calcul de la puissance minimale pour un trajet

D'abord, nous souhaitons représenter le graphe sous la forme d'un dictionnaire, avec comme clé une ville et comme arguments la liste des triplets de ses voisins, la puissance minimale pour aller vers un voisin et la distance entre les deux points.

Nous implémentons une première méthode *add_ edge* qui ajoute une arête au graphe. D'abord, nous vérifions si *node 1* est dans le graphe. La complexité de cette opération est en O(n) dans le pire des cas. Ensuite, on ajoute *node 1* au graphe s'il n'y est pas.La complexité de cette opération est en O(1). On fait de même avec *node 2*. Donc cet algorithme a une complexité en temps en **O(n)** dans le pire des cas

Ensuite, on écrit une fonction *gaph_ from_ file*, en dehors de la classe Graph, qui permet de charger un fichier. D'abord, on lit la première ligne pour obtenir 'n' et 'm' (O(1)). Puis, on crée un objet graphe (O(n)). Enfin, on lit les m lignes suivantes et on ajoute les arêtes au graphe (O(n)xcomplexité(*add_ edge*). Donc cet algorithme est en **O(n+n*m)** dans le pire des cas.

Maintenant, nous allons écrire une deuxième méthode *connected_ components_ set* qui trouve les composantes connectées du graphe G. Nous allons d'abord implémenter une méthode *connected_ components* qui retourne une liste de liste. Pour ce faire, on effectue un parcours en profondeur du graphe en utilisant une fonction récursive. La complexité de cet algorithme est en O(V+E). La méthode frozenset a elle une complexité en O(C) où C est le nombre de composantes connexes. Dans le pire des cas, C=V. Donc, dans le pire des cas, la complexité de la méthode frozenset est en O(V). Au total, la complexité dans le pire des cas de *connected_ components_ set* est donc celle de *connected_ components* soit en **O(V+E)**

Dans un troisième temps, écrivons une méthode *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. La complexité de cette méthode dépend du parcours en profondeur effectué. Dans le pire des cas, aucun chemin ne relie deux points, donc nous devons parcourir toutes les arêtes et tous les noeuds du graphe pour montrer qu'il n'existe aucun chemin reliant les deux points. La complexité en temps dans le pire des cas est donc en **O(V+E)**

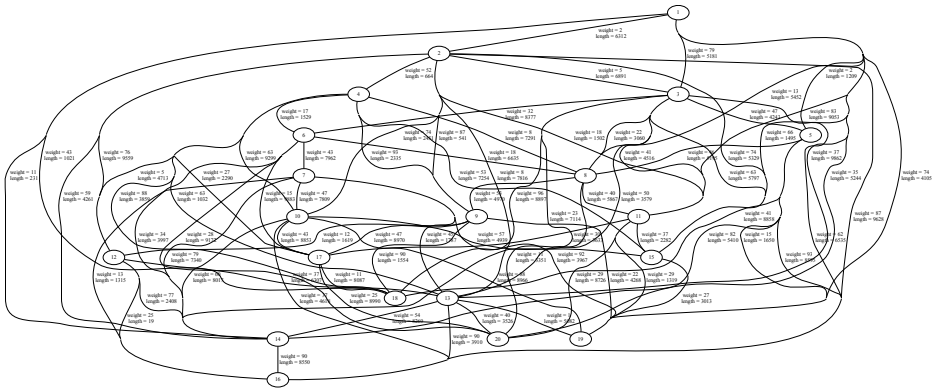

FIGURE 1 – Visualisation du graphe network.1.in à l'aide du module graphviz

Maintenant, nous allons implémenter une méthode *min_power* qui calcule, pour un trajet t donné, la puissance minimale d'un camion pouvant couvrir ce trajet. Cette méthode retourne la puissance minimale et le chemin suivi. Pour ce faire, nous effectuons une recherche dichotomique de la puissance minimale. Déjà, la compléxité en temps de cet algorithme dépend en partie de celle de *get_path_with_power*. De plus, la recherche binaire est en O(ln(B-A)) où A est le poids minimum des arêtes du graphe et B le poids maximum. Donc la compléxité en temps dans le pire des cas de *min_power* est en **O(ln(B-A)*(V+E))**

Puis, nous implémentons une méthode de visualisation de graphe *view*. Nous importons *graphviz*. Cette méthode prend en entrée deux noeuds du graphe et retourne une visualisation du plus court chemin pour aller d'un de ces points à l'autre. La compléxité de cette méthode dépend principalement de celle de *min_power*. De plus, pour visualiser le graphe, il faut créer les autres noeuds et arêtes, ce qui coute, dans le pire des cas, une compléxité temporelle en O(V) pour les noeuds et O(E) pour les arrêtes. Puis, la boucle imbriquée implique une compléxité en O(V*E). Au total, la complexité de *view* est donc en **O(V*E+V+E+ln(B-A)*(V+E))**

Voici des exemples de visualisations :

On utilise maintenant *time.perf_counter* pour déterminer le temps néces-

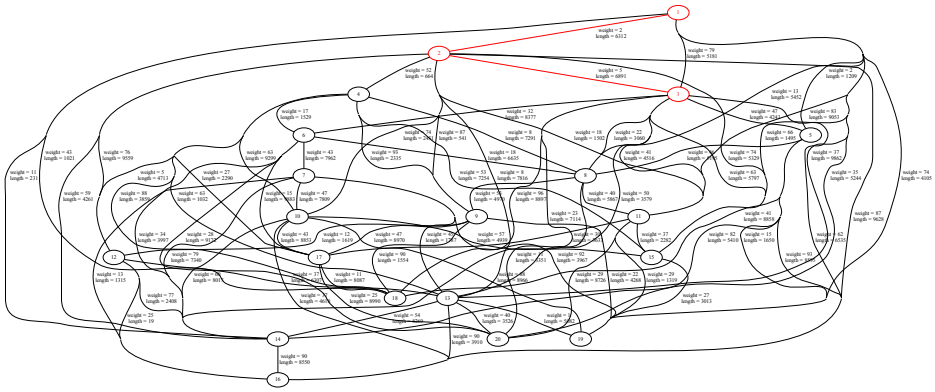

FIGURE 2 – Visualisation du graphe network.1.in à l'aide du module graphviz (on visualise en rouge le chemin le moins coutant pour aller de 1 à 3)

saire pour calculer la puissance minimale sur l'ensemble des trajets. Nous le faisons à l'aide de la fonction *time_array* présente dans la branche *main* de notre fichier git. Nous estimons à plusieurs dizaines de milliers de secondes ce temps.

Nous allons maintenant répondre à la question 11 : Soit un graphe G et A un arbre couvrant de poids minimal du graphe G. On considère T un trajet de G. Déjà,il est clair que Puissance minimale pour couvrir T dans G <= Puissance minimale pour couvrir T dans A. De plus, distinguons deux cas :

Cas 1 : Il existe une arête de T qui n'est pas dans A. On appelle C le sous graphe de G formé de l'ensemble des arêtes de T qui sont dans A, et de l'ensemble des arêtes de T qui ne sont pas dans A. Soit B un arbre couvrant de poids minimal du graphe C Alors B couvre tous les sommets de T (Car B contient toutes les arrêtes de T). Notons D le sous graphe de G composé de l'ensemble des arrêtes de T qui sont dans A. Puisque A est un arbre couvrant de poids minimal pour G, il est aussi un arbre couvrant de poids minimal pour D. Donc somme des poids des arrêtes dans A <= somme des poids des arrêtes dans B. De plus, B contenant toutes les arrêtes de T qui sont dans A, Puissance minimale pour couvrir T dans A <= Puissance minimale pour couvrir T dans B Par le théorème de sous-optimalité de l'arbre de poids minimal, Puissance minimale pour couvrir T dans A = Puissance minimale pour couvrir T dans B Or il est clair que la puissance minimale

pour couvrir T dans B est égale à la puissance minimale pour couvrir T dans G D'où Puissance minimale pour couvrir T dans A = Puissance minimale pour couvrir T dans G **D'où le résultat**

Cas 2 : T se trouve totalement dans A. Alors toutes les arrêtes de T qui sont dans A ont un poids nul et sont donc nécessaire pour couvrir T. Le résultat est alors clair.

Nous implémentons maintenant une fonction *kruskal* qui prend en entrée un graphe g et qui renvoie un arbre couvrant de poids minimal de g. Pour ce faire, nous écrivons deux fonction *find*, qui détermine les parents d'un noeud x, et *union* qui, pour deux noeud x et y de même profondeur, renvoie le parent commmun de ces deux noeuds. Ainsi, kruskal implique d'abord un tri des arêtes, dont la complexité temporelle est, dans le pire des cas, en O(E*log(E))

Ensuite, il faut construire l'arbre couvrant. La boucle parcourt toutes les arêtes et, pour chaque arête, fait les opérations d'union et de find. Dans le pire des cas, chacune des deux fonctions ont une complexité en O(V). La compléxité totale de cette étape est donc en 0(V*E). Au total, la complexité de *kruskal* est donc en **O(log(E)*E+V*E)**

On implémente désormais la fonction *min_power_for_path* qui prend en entrée un arbre couvrant et qui retourne la puissance minimale de cet arbre. D'abord, on initialise le dictionnaire, ce qui demande une compléxité en O(V) dans le pire des cas. Ensuite, on fait un parcours en profondeur de l'arbre, ce qui a une complexité en O(V+E). Puis, on trouve le chemin entre la source et la destination, ce qui donne une compléxité temporelle dans le pire des cas en O(V). Enfin, on calcule la puissance minimale, qui a une complexité en O(V). Au final, *min_power_for_path* a comme compléxité temporelle dans le pire des cas **O(V+E)**

Partie 2 : Optimisation de l'acquisition de camions

Nous allons implémenter une fonction de type greedy qui permet de retourner une collection de camions à acheter ainsi que leurs affectations sur des trajets, pour maximiser la somme des profits obtenus sur les trajets couverts. L'idée est d'assigner les camions aux trajets ayant le meilleur rapport profit/puissance requise. Nous implémentons d'abord la fonction *greedy_knapsack* qui renvoie une liste de tuples représentant les affectations des camions aux trajets. Chaque tuple contient deux éléments : le tuple représentant un camion et le tuple représentant un trajet ainsi que le profit total avec ces camions. Puis nous implémentons la fonction *assign_trucks_to_routes* qui retourne la sélection de camions la plus rentable et le profit correspondant.

Analysons la compléxité de cette implémentation : On crée d'abord le kruskal : compléxité en O(E*log(E)). Puis, on calcule la puissance minimale pour chaque arête et pour chaque trajet : compléxité en O(R*Vlog(V)). Puis, nous utilisons la méthode greedy, qui est en O(T*R) avec T le nombre de camions et R le nombre de trajets. Au total, la compléxité en temps de cette implémentation est donc dans le pire des cas en **O(R*Vlog(V)+T*R+E*log(E))**

Conclusion

En conclusion, nous avons déterminé la puissance minimale nécessaire pour couvrir un trajet, puis nous avons optimisé le temps de calcul de cette puissance. Enfin, nous avons optimisé l'acquisition de camions en déterminant les camions qui optimisent le trajet, le trajet effectué et le profit de l'entreprise sur ce trajet.