Compte rendu du projet de programmation

Perann Nedjar, Adrien Letellier

March 2023

1 Première Partie

Dans cette séance, on modifie la classe Graph fournie dans le code. Dans la première question, on implémente une fonction permettant de lire les fichiers "network" fournis et de les transcrire en dictionnaires correspondants à un graphe. Dans ces dictionnaires, les clés correspondent à des noeuds et les valeurs associées aux noeuds correspondent aux noeuds connectés au noeud clé, avec des informations sur la liaison: $\{node_k : (node_i, power, distance)\}$. On implémente également une fonction permettant d'ajouter des arêtes et des sommets aux graphes.

Par la suite, on implémente une fonction get_connected_component servant à trouver les composantes connexes d'un graphe. Cette fonction est basée sur un algorithme de recherche profonde ("Depth first search" en anglais).

Complexité : L'algorithme Depth first search (DFS) a une complexité en $O(V + E)$, où V désigne le nombre de sommets du graphe et E désigne le nombre d'arêtes. En effet, cet algorithme explore chaque noeud et chaque arête exactement une fois (grâce au dicitionnaire des noeuds visités, on ne passe par un noeud que si il n'a pas été visité avant). Etant donné que l'on effectue un l'algorithme DFS sur chaques noeuds du graphe, on obtient une complexité en $O(V(V + E))$. Dans le cas où le graphe est dense (E est de l'ordre de V^2), on obtient une complexité en $O(V^3)$.

Maintenant que l'on connaît les composantes connexes de notre graphe, on programme une fonction get_path_with_power permettant de savoir si un camion de puissance donnée p peut parcourir un trajet précis. Pour ce faire, on réalise un DFS à partir du sommet de départ jusqu'à atteindre l'arrivé, puis on reconstruit ensuite le chemin parcouru si le point d'arrivé a été atteint par le DFS.

Complexité: La fonction DFS_chemin a une complexité en $O(V+E)$ (même raisonnement qu'à la question précédente). La vérification de la connexité des sommets a une complexité en $O(V)$ car on parcourt tous les sommets. La reconstruction du chemin à partir du dictionnaire parent a elle aussi une complexité en $O(V)$ car elle parcourt le dictionnaire parent à partir du sommet d'arrivée jusqu'au sommet de départ pour reconstruire le chemin. Finalement la com-

plexité de l'algorithme est en $O(V + E)$.

On cherche maintenant à determiner le plus court chemin pour un trajet donné et une puissance donné. On va pour se faire utiliser l'alogrithme de Dijkstra dans une fonction dijkstra_shortest_path_with_power.

On va désormais chercher à determiner la puissance minimale permettant de couvirir un trajet donné. Pour ce faire, nous implémentons une fonction min_power qui prend en argument un trajet et renvoie la puissance minimale qu'un camion doit avoir pour le couvrir. On va procéder à une recherche binaire. On liste d'abord les puissances des arrêtes du graphe par ordre croissant puis on reélaise une dichotomie entre la puissance maximale et la puissance minimale, en vérifiant avec get_path_with_power si un trajet existe. On réduit ainsi l'intervalle entre p_{min} et p_{max} jusqu'à tomber sur la puissance minimale.

Complexité: Lors de la première boucle for, on parcourt toutes les arêtes du graphe, ce qui implique une complexité en $O(E)$. La conversion de la liste des puissances en un ensemble avec la fonction set() a une complexité de $O(E)$. Le tri de la liste avec la méthode sort() a une complexité de $O(ElogE)$ (la fonction sort() de python réalise un tri fusion, dont on connait la complexité). La boucle while de recherche dichotomique a une complexité de $O(logE)$. En effet, à chaque itération, elle divise la taille de la plage de recherche par deux.
Ainsi, étant donné que la fonction get path with power a une complexité en $O(V + E)$, la complexité de l'algorithme est $O((V + E)log(E))$.
Dans le cas d'un graphe dense, on obtient une complexité en $O(V^2log(V^2))$.

2 Deuxième partie

En testant nos algorithmes de la section précédente sur les fichiers routes, on obtient les temps suivants (en minutes):

routes 1	routes 2	routes 3	routes 4	routes 5
2.69	9760	22972	38951	53088

routes 6	routes 7	routes 8	routes 9	routes 10
31968	44822	39986	9543	38777

On réalise que ces temps ne sont pas raisonnables. On va utiliser la métthode de l'arbre de poids minimal pour obtenir un résultat plus efficacement. Pour cela, on programme une fonction Kruskal appliquant l'algorithme du même nom à un graphe, nous fournissant l'arbre de poids minimal couvrant ce graphe.

Complexité: La fonction Kruskal_int réalise l'algorithme de kruskal avec une strucutre de type union-find. La première boucle for parcourt tous le noeuds et réalise $O(1)$ opérations élémentaires, la complexité de cette partie de l'algorithme est donc en $O(V)$. La seconde boucle for qui sert à la création

de la liste d'arête triée a une complexité en $O(E)$ puisqu'elle parcourt un nombre de valeur du même ordre que l'ordre de grandeur que l'ensemble des arêtes, la fonction de tri utilisée juste après a une complexité en $O(Elog(E))$. La complexité de cette partie de l'algorithme est donc en $O(Elog(E))$. Dans la boucle suivante on parcourt la liste des arêtes en appelant soit la fonction find dont la complexité est donnée par la fonction d'Ackerman inverse, (on peut considérer cette complexité comme constante, car la fonction d'Ackerman est inférieure à toute fonction logarithmique) soit la fonction union dont la complexité est en $O(1)$. La complexité de cette partie est donc en $O(Elog(V))$. Enfin la fin de l'algorithme consiste en une application de la fonction kruskal_int et en la création d'un graphe initial en parcourant tous les sommets et les arêtes, donc en $O(V + E)$. Cette fonction a donc une complexité finale en $O(Elog(E))$.

Après avoir testé notre fonction Kruskal, nous allons implémenter une fonction min_power_routes qui prend en argument une liste de routes et un point de départ, fixé par défaut au noeud 1. Cette fonction nous rend un fichier routes.out avec la puissance minimale nécéssaire pour couvrir chaque trajet. Dans cette fonction, on commence par effectuer une recherche en profondeur sur l'arbre de poids minimum recouvrant le graphe, afin d'obtenir une liste des parents de chaque sommets.Par la suite, on détermine le chemin entre le sommet d'arrivé et de départ en remontant les parents pour chaque route de la liste. Pour obtenir le chemin final, on fusionne les chemins obtenus entre le sommet de départ et chaque sommets. On calcule finalement la puissance minimale pour couvrir chaques chemins.

Complexité: La fonction get_kruskal a une complexité en $O(Elog(E))$ et la fonction de DFS a une complexité en $O(V + E)$. Notons N le nombre d'élément dans la list _routes. Dans cette boucle on éxécute d'abord un DFS (en $O(V+E)$) puis on fusionne les chemins avec une opération en $O(V)$. L'opération de calcul de la puissance minimale requise pour chaque chemin est en $O(E)$. Finalement, on obtient une complexité en $O(N(Elog(E) + V + E))$.

Procédons maintenant à l'estimation des temps d'executions nécéssaires à notre fonction (en secondes), à l'aide du scipt Estimations question 15.py:

routes 1	routes 2	routes 3	routes 4	routes 5
0.0014	6.8	374	645	439

routes 6	routes 7	routes 8	routes 9	routes 10
658	560	406	401	486

Nous obtenons des temps très inférieurs à ceux que nous avions précédemment. Par exemple, pour le fichier routes.10.in nous obtenons 486 secondes alors que nous obtenions environ 26 jours avec la première méthode : l'algorithme donc est environ 4600 fois plus rapide.

3 Troisème partie

Nous allons désormais considérer des fichiers trucks correspondant à des catalogues de camions disponibles. Nous allons tâcher de trouver le choix otpimal d'une combianaison de camion pour une route donnée afin de maximiser le profit (ici, l'utilité).

Nous començons par implémenter une méthode naïve otpimale avec la fonction algorithme_naif. Nous allons, pour un fichier routes donné, considérer toutes les combinaisons de trajets possibles, calculer les utilités respectives, prendre les combinaisons pouvant être effectuées dont le profit est maximal. Dans les faits, cette fonction est impossible à faire tourner sur la majorité des graphes.
Complexité: Si on note N le nombre de trajet et T le nombre de camions, la fonction algorithme naïf a une complexité exponentielle en $O(T * 2^N)$. En effet on explore toutes les combinaisons possible (2^N combinaisons) et on appelle à chaque fois la fonction is_list _trucks _buyable qui a une complexité en $O(T)$.

On comprend ainsi pourquoi cette méthode optimale est impossible à faire tourner sur des gros graphes.

Nous allons donc par la suite nous diriger vers une approche glouton ("greedy" en anglais) en assimilant notre problème au problème du sac à dos. Cette méthode de résolution n'est pas optimale mais permet d'obtenir un résultat dans un temps correct. Nous chercherons ici à selectionner les trajets de façons à maximiser le profit total sans dépasser le budget.

La fonction max_utility_from_ratio_glouton est la première fonction que nous avons construite pour faire un algorithme glouton. Elle s'appuie sur des fonctions intermédiaires, dont la plus importante est ratio_utility_cost, qui calcule les ratios entre l'utilité et le coût (c'est-à-dire le coût du camion le moins cher permettant de faire le trajet) pour chaque trajet d'un fichier routes, et renvoie une liste triée dans l'ordre décroissant de ces ratios. Le principe de la fonction max_utility_from_ratio_glouton est de prendre les éléments de la liste des ratios utilité-coût dans l'ordre, jusqu'à ce que le budget soit dépassé. Ainsi, on prend en priorité les trajets ayant le meilleur rapport utilité-coût.

Nous avons essayé d'améliorer cette fonction avec la fonction improved_glouton qui fait la même chose que la précédente mais au lieu de s'arrêter dès que le budget est dépassé et se contenter de la liste des trajets obtenue, regarde parmi les trajets restants si l'on peut en ajouter tout en restant dans le budget. Cependant, sur les tests que nous avons faits, avec le fichier routes.3, et les fichiers trucks.1 et trucks.2, nous n'observons aucune différence entre les deux fonctions car il n'y a pas de trajet peu coûteux restant que l'on peut effectuer en restant dans le budget.

Enfin, nous avons commencé la question 20 qui faisait partie des questions bonus. Nous avons réussi la 1ère partie de la question. Il a fallu pour cela créer des fichiers routes_esp.x.in, parallèles aux fichiers routes.x.in, dans lesquels on trouve à chaque ligne le départ, l'arrivée, et l'utilité du trajet. Une fois cette étape réalisée, il suffit de réutiliser l'algorithme knapsack de la question 18. Néanmoins, nous n'avons pas eu le temps d'aborder la deuxième partie de la question 20.