Gabrielle DURIEU
Nicolas NOLDUS
Théo PAQUIER

Compte rendu
Projet de programmation
(lien git: ***nicolasnoldus/ensae-prog23***)

On note :

- V le nombre de nœuds
- E le nombre d'arêtes

$V > 0$ et $E > 0$

Question 2

L'algorithme :

Le dfs est une fonction récursive qui, à chaque nœud, visite tous ses voisins. La récursion a lieu lorsque que l'on parcourt tous les nœuds qui sont adjacents au nœud en question puisqu'on applique dfs à chacun de ces nœuds. Ainsi, pour chaque voisin, on refait une liste de voisin et on les rajoute dans "visites". On obtient pas le graphe dans son intégralité parce que l'on ne parcourt que les voisins successifs de chaque nœud.

On utilise donc dfs pour la fonction connected components afin d'avoir des "sous-graphes" connexes. La liste "visite" va garder en mémoire tous les nœuds qui ont déjà été visités et on met une condition de non-visite pour alléger les temps de calculs sans quoi à chaque fois, la fonction devrait parcourir le graphe en entier.

L'avantage de dfs est que c'est un moyen d'avoir tous les chemins possibles entre deux nœuds, d'autant plus lorsqu'on rajoute une condition de puissance.

L'inconvénient est que lorsque l'on traite de grands graphes, à partir de network 5 ou 6 par exemple, le temps de calcul devient trop important pour l'utiliser comme façon de trouver des chemins.

Complexité :

La complexité de connected_components_set dépend de celle de connected_components qui dépend elle-même de celle de dfs.

Dfs a une complexité $O(V+E)$ parce qu'il doit explorer au plus une fois chaque nœud et qu'il passe 2 fois par chaque arête (une fois pour chaque nœud à une extrémité de l'arête).

Dfs est appelé sur chaque sommet du graphe au plus donc la complexité de connected_components est $O(V*(V+E))$. Or dans le cas où il y a relativement peu d'arêtes par rapport aux nœuds (cas d'un graph creux), la complexité est $O(V+E)$.

Ainsi la complexité de connected_components_set dépend de celle de connected_components et de celle de la conversion en ensemble de chaque composante connexe. La complexité de la conversion en ensemble d'une liste est O(n) où n est la longueur de la liste. en notant C la longueur moyenne d'une composante et en gardant l'hypothèse que nous avons un graphe creux, la complexité de connected_components_set est O(V+E+n*C). Dans la plupart des cas, n est négligeable devant V+E donc on a connected_components_set de complexité **O(V+E).**

Question 3

L'algorithme :

Afin de construire la fonction get path with power, nous avons utilisé un bfs. C'est une fonction qui demande bien moins de calculs que dfs et par conséquent permet d'avoir des temps de calculs raisonnables. Étant donné que la majorité de nos fonctions qui suivent dépendent de bfs nous avons dû l'optimiser afin qu'il soit rapide et puisse servir à plusieurs propos. Ainsi, le paramètre "dest" est désuet mais nous devions le créer sans quoi les autres fonctions n'allaient pas être compatibles.

Le but de notre bfs est de nous rendre un dictionnaire qui à pour clé un nœud et en valeur celui par lequel il est arrivé. Ainsi, nous créons un dictionnaire, une liste queue importante pour le fonctionnement de l'algorithme et un set qui gardera en mémoire les sommets visités (on utilise un set plutôt qu'une liste car nous cherchons seulement à savoir si le sommet en question est dedans ou pas, son indice importe peu).

À chaque sommet nous allons regarder les sommets adjacents qui se sont pas dans "visited" pour éviter de faire des boucles et calculs inutiles. Nous les rajoutons dans la queue et le sommet de référence sera le dernier sommet ajouté. Au fur et à mesure on ajoute de moins en moins de sommets car ils sont visités et on se retrouve avec une queue vide, qui est la condition pour laquelle on arrête la boucle while.

get_path_with_power est donc un bfs avec une condition de puissance. Pour se faire, on supprime les arêtes donc la puissance est trop élevée et on regarde si notre destination se trouve dans un graphe connexe trouvé par bfs.

C'est une méthode de calcul rapide qui permet d'avoir de bon résultats. Néanmoins, un petit problème qui se règle en quelques lignes de codes dans les utilisations futures, est que le bfs ne fait pas la distinction entre un nœud qui est l'ancêtre d'un nœud ou de plusieurs. Dès lors il peut apparaître des boucles si l'on décide de "remonter" le chemin puisqu'il reviendra souvent à un nœud qui a plusieurs "descendants".

Nous avons tout de même gardé cette option car elle permet de voir rapidement si un code est faux ou comporte une boucle en raison de sa rapidité.

Complexité :

La complexité de get_path_with_power dépend de celle de bfs.
La complexité temporelle de bfs est O(V + E). En effet, la boucle for a une complexité temporelle O(V) car il y a au plus V ancêtres et la boucle while a une complexité O(E) car il y a au plus E arêtes.
get_path_with_power y fait appel une fois à bfs donc sa complexité temporelle est O(V + E) également.

La complexité spatiale est **O(V)** pour dfs et get_path_with_power car on utilise la liste ancêtre et une liste parcours qui ont toutes les deux une longueur maximale V.

Question 6

L'algorithme :

La fonction min_power est une fonction de recherche binaire. On considère d'abord un graphe connexe contenant tous les sommets reliés. Pour ce faire nous utilisons donc notre bfs qui a été optimisé pour les besoins de la cause. Nous créons une liste actu qui changera à chaque itération.

Nous procédons ensuite à la recherche binaire, nous ne prenons pas toutes les puissances présentes sur le graphe mais simplement la plus grande, la plus petite une fois que l'on aura fait plusieurs itérations nous nous retrouverons avec la puissance minimale pour parcourir le trajet qui est nécessairement entre la puissance max et 0.

À chaque itération on regarde si notre destination est dans le graphe connexe obtenu par bfs (d'où l'importance de son optimalité) avec comme puissance la moyenne entre le min et le max. Si la destination y est, on pose que la puissance max est maintenant la moyenne, et on reprend la nouvelle moyenne de ces deux valeurs. Si la destination n'y est pas, on définit que le min prend la valeur de la moyenne et on continue en reprenant la nouvelle moyenne et en réitérant ce procédé.

Nous avons eu quelques soucis de boucles infinies dans notre code car il y avait des problèmes avec la condition de "fin==debut". C'est pourquoi nous avons opté pour une option où l'on sort de la boucle dès que l'écart entre le min et le max est de 1, et on prend la valeur max pour être sûr qu'il est possible de faire le chemin avec la puissance que l'on a.

En le testant sur network 2 entre 1 et 12, nous avons obtenu : ([1,2,4,12], 52761) en moins de trois secondes.

Sur le network 10, on trouve la réponse entre le sommet 1 et le 153789 en 17,53 secondes, avec un chemin assez long et un power min de : 5346873.

Au début nous faisions ce min power avec un dfs mais lorsqu'on l'a testé sur le network 5, nous avons eu un message d'erreur comme quoi le nombre maximal d'itérations avait été atteint. C'est pourquoi nous avons opté pour le bfs qui ici peut remplir la même fonction que le dfs avec moins de calculs.

Complexité :

La complexité temporelle de min_power est **O((V+E)*log(P))** avec P la puissance maximale entre les nœuds parcourus.
La boucle while s'exécute log(P) fois au plus et min-power fait appel à la fin à get_path_with_power de complexité O(V+E), d'où la complexité totale de l'algorithme.

Question 10

Algorithme

On crée une fonction afin de pouvoir mesurer le temps d'exécution de notre algorithme de calcul de puissances minimales. Pour ce faire, on prend un fichier route en parallèle d'un fichier

network pour pouvoir appliquer le min power. Dans l'état actuel de notre code, nous n'avons pas considérablement optimisé le temps de calcul en cela que nous continuons de faire des bfs, mais sur des graphes auxquels nous avons appliqué Kruskal (la résolution en passant par le dictionnaire des ancêtres est compliquée en cela que nous obtenons systématiquement une boucle en passant par le noeud "originel"). En effet, nous n'avons pas de nœud qui soit son propre parent. Dès lors lorsque l'on souhaite remonter en effectuant graph[a]=a, nous parvenons à un moment où il y aura un aller-retour permanent entre deux nœuds.

Pour ce qui est des tests et des estimations, sur le fichier routes 1, qui est très court, on met 0.01981806755065918 secondes pour tracer l'ensemble des routes. Néanmoins, dès que l'on teste sur le deuxième fichier routes, on met 30 secondes pour déterminer 16 trajets. Étant donné que l'on cherche à déterminer 100 000 trajets, on peut estimer que le temps de calcul sera de 52 heures (2 jours et 4 heures) pour déterminer l'ensemble des trajets. On peut, à l'évidence, estimer que cela n'est pas la méthode la plus optimale.

Il s'agira donc de trouver une fonction min_power utilisant le dictionnaire d'ancêtres pour réduire considérablement la complexité.

Question 11

On peut prouver qu'un arbre de n noeuds a n-1 arêtes en raisonnant par récurrence.
Il n'y a qu'une arête entre deux nœuds.
Puisqu'un arbre n'a pas de boucle, à chaque fois que l'on ajoute un nœud, on n'ajoute qu'une seule arête.
En conséquence, on a toujours n-1 arêtes dans un arbre de n noeuds.

Question 12: Algorithme de Kruskal

Nous suivons ici l'approche générale inspirée par exemple par le manuel de Dasgupta et al. L'idée générale de l'algorithme est très simple: on commence par trier nos arêtes par ordre croissant de puissances, puis on les ajoute une par une dans l'ordre tant qu'elles ne produisent pas de cycle dans notre nouveau graphe. On obtient ainsi un arbre couvrant de poids minimal par itération. La subtilité repose dans la méthode de détection de cycles, qui consiste dans les méthodes union et find. Il y a beaucoup de façons de les coder en pratique, et nous avons choisi la forme de classe pour différentes raisons, notamment parce qu'elle évite un certain nombre d'erreurs, d'indice en particulier, et qu'elle permet de mieux contrôler les différents aspects de la fonction. Au moment d'ajouter une nouvelle arête, on regarde s'il y a déjà un cycle: pour cela, la fonction find(x) donne les parents de x, en l'occurrence les noeuds connectés à x, qu'on considère comme un ensemble avec la fonction union; si ajouter l'arête ne forme pas de cycle, elle entre dans notre arbre en constitution et on prend en compte avec union qu'il existe désormais une arête entre nos deux noeuds, formellement x et y appartiennent au même ensemble. Pour éviter d'avoir un amas d'ensembles différents de tailles variables, on prend systématiquement le plus gros et on lui ajoute le nouveau nœud, d'où le filtrage par "rang" des sous-arbres.

Question 14

Pour notre nouvelle mouture de la fonction min_power, il y a différentes possibilités, la plus simple consistant à simplement appliquer notre min_power initiale à notre graphe “nettoyé” qui est désormais un arbre, et donc dans lequel des méthodes DFS/BFS seront moins coûteuses en temps.

Question 16. LCA

L’idée est en fait d'optimiser le temps de calcul en prenant deux nœuds que l’on veut se faire rejoindre et d’utiliser d’abord un bfs pour avoir le dictionnaire ancêtre et à partir de ce dictionnaire trouver un chemin entre les deux nœuds qui serait rapidement calculé.

L’idée est donc de prendre dans le dictionnaire des ancêtres le nœud d’où l’on vient pour remonter jusqu’à ce que les chemins partant des deux nœuds se croisent. Néanmoins, il pourrait y avoir un problème comme évoqué précédemment car des boucles pourraient se faire en cela qu’on pourrait revenir plusieurs fois au même nœud qui aurait plusieurs descendants.

C’est pourquoi on crée un set pour chaque chemin qui garderait en mémoire les nœuds déjà visités et qui permettra d’éviter de faire des boucles et donc de prendre le chemin le plus court.

Les dernières conditions du code ont été écrites afin de ne pas omettre le sommet de “rencontre” entre les deux chemins.

Néanmoins notre LCA ne peut aboutir en cela que notre bibliothèque d’ancêtres conduit nécessairement à une boucle en absence de nœud “originel”. Nous avons essayé de pallier ce problème en ajoutant “artificiellement” que l’ancêtre du nœud 1 était lui-même. Nous n’avons cependant pas la garantie que ce sera toujours le premier nœud qui sera l’origine. Dès lors, la solution alternative qui aurait été de raisonner en terme de rang (par rang il s’entend éloignement au nœud d’origine) est impossible.

De plus, nous n’avons pas eu le temps de faire une solution alternative qui aurait été de laisser remonter les deux trajets jusqu’au moment où l’on obtiendrait un cycle pour parvenir au pseudo nœud originel et retirer les chemins en commun si les deux nœuds étaient du même “côté” du nœud et les concaténer s’ils étaient de côtés opposés.

Notre LCA permet donc de trouver parfois le chemin correct mais dès que le nœud pseudo-originel est impliqué de près ou de loin, les résultats sont faussés.

Question 18. Maximisation de la somme des profits

Remarque préliminaire générale sur l’interprétation du problème: il y a une marge d’interprétation, dans la mesure où il y a 1° un écart entre les unités et 2° où la notion de profit à maximiser n’est pas complètement définie. Expliquons-nous en prenant l’exemple du network.1 (avec ses fichiers associés), qui est le principal fichier autour duquel la conception et les tests d’algorithmes s’orientent. Sur ce graphe léger, les puissances minimales sont de l’ordre de la dizaine ou de quelques dizaines, alors que le camion le moins puissant a une puissance de 500 000. En maintenant les unités inchangées, on ne choisit donc jamais que le premier camion, et par ailleurs le rapport utilité/gain est disproportionné de même: le gain fluctue entre quelques centaines et quelques milliers (rarement une dizaine de millier ou plus), alors que le camion le moins cher coûte 50 000 et que les prix augmentent vite.

Pour avoir une approche intéressante du problème, nous avons choisi 1° de réévaluer les unités, de sorte que les puissances soient multipliées par 100 000 et les profits par 1000. De la sorte, l'algorithme de sélection de camions a bien un rôle. Mais aussi, ceci permet d'avoir 2° une approche "économique" du profit intéressante, parce qu'alors l'utilité devient une recette et on peut calculer un véritable profit comme soustractions recettes - coûts, qui donne un sens à nos valeurs, et qui permet aussi un algorithme plus robuste dans le sens où il est confronté à des valeurs très différentes, parfois (mais rarement) négatives. A défaut, on peut aussi considérer dans nos approches naïves un rapport recette/coût. Ceci ne change que du détail et pas le fonctionnement général de l'algorithme.
Un autre intérêt de cette réévaluation, on en reparlera brièvement, est qu'elle permet d'alléger certaines fonctions (notamment celles qui utilisent de la programmation dynamique) car on peut alors changer l'échelle pour réduire le budget d'un facteur 1000 par exemple: ceci, lorsqu'on crée un tableau (comme nous avons optimisé, c'est souvent un vecteur à une seule ligne, mais tout de même!), permet d'alléger significativement l'algorithme en faisant passer un vecteur de 25*10**9 colonnes à un vecteur beaucoup plus réduit (de seulement quelques 25 000 colonnes).
Cela dit, le terme de *profit* est utilisé systématiquement dans l'algorithme au sens de soustraction ou de rapport, à la différence de la simple *utilité* mentionnée dans le document routes.x.in qui est en général nommée *amount*.

Première approche globale: algorithme de sac à dos et programmation dynamique.

Nous avons directement tenté l'approche inspirée du problème du sac à dos (*knapsack*). Son intérêt est qu'elle permet d'obtenir un maximum global. L'idée de base est assez simple: on regarde, sous contrainte (ici de budget) toutes les différentes combinaisons d'objets réalisables sous notre contrainte. Ainsi on regardera systématiquement si ajouter un chemin augmente le profit ou non, et à force d'essayer toutes les combinaisons, on obtiendra l'optimum global.
La version que nous proposons ici repose essentiellement sur l'idée de programmation dynamique la plus poussée que nous avons pu faire en temps réduit: c'est-à-dire qu'on crée un vecteur à une seule ligne qui enregistre les différentes possibilités pour éviter de recalculer plusieurs fois les mêmes sous-problèmes, enregistre le profit et sert de mémoire temporaire. On n'aurait besoin que d'une seule ligne (et non pas d'une matrice qui prend beaucoup plus d'espace intermédiaire) si l'on n'avait à résoudre que le problème du profit, mais comme il nous faut aussi enregistrer l'allocation des camions sur les routes, une matrice doit être introduite en plus. Il y aurait encore beaucoup d'optimisation à faire, mais l'algorithme donne une première idée, et calcule relativement rapidement les optima de petits graphes.
Notons que cette approche est déjà inspirée par une approche naïve, avec notre filtrage du profit, etc.

Deuxième approche: la version "naïve" (greedy)

Notre deuxième approche brille par sa simplicité: il s'agit simplement de filtrer les chemins par leur profit (au sens défini plus haut, mais on aurait aussi pu prendre une fraction, par exemple pour aller dans le sens d'un knapsack fractionnel), puis de les ajouter jusqu'à saturer notre budget. La limite de cette approche est qu'elle repose sur l'efficacité de notre fonction pour obtenir une puissance minimale, et s'il fallait le faire pour de grands graphes, le temps ne serait pas réaliste; il n'en demeure pas moins que sur de petits graphes, la réponse sort quasi instantanément.

Encore une fois, il s'agit avant tout d'illustrer une approche: on combine ici une recherche locale et naïve. A part son manque d'efficacité potentiel (sauf à beaucoup optimiser nos fonctions sous-jacentes), son autre problème majeur est de conduire à un optimum local: il est possible qu'il reste un peu de budget, et, même à admettre la conception du profit que nous avons prise comme point de départ, le reste de budget aurait pu être mieux alloué si l'on échangeait un ou plusieurs chemins contre une saturation totale du budget avec des chemins moins profitables sur le plan du ratio, mais globalement plus profitables dans le cas de saturation.
Il y a donc de nombreuses perspectives d'amélioration: programmation dynamique, combinaison avec d'autres approches, mais nous présentons cet algorithme pour sa simplicité et pour sa (relative) efficacité.

Question 20: Probabilité de casse (ébauche)

Nous n'avons pas eu le temps de faire plus qu'esquisser des réponses à cette question, mais elle est intéressante notamment dans la mesure où elle permet de dépasser certaines limites précédemment rencontrées. A part le simple calcul d'espérance que recouvre la première partie, notre inspiration principale est le modèle de "recuit simulé" (*simulated annealing*)

Première partie: calcul d'espérance

Faute de temps, notre fonction est loin d'être optimale, mais il s'agit essentiellement d'un calcul d'espérance, où il faut seulement retrouver la longueur du chemin et la multiplier par la probabilité de casse (en admettant qu'on paie quand même, mais qu'on ne récupère rien si une arête quelconque casse), ainsi que multiplier la distance par le prix de l'essence. Il n'y a donc pas de difficulté algorithmique particulière, dans la mesure où l'optimisation à proprement parler est déjà faite.

Deuxième partie: battre le fer tant qu'il est chaud

Nous avons essayé une seule approche, parmi la multitude que nous aurions pu adopter, celle de recuit simulé: il s'agit essentiellement, dans l'idée de base, de changer aléatoirement un élément et de comparer avec notre configuration précédente, en admettant des erreurs ponctuelles "tant que le fer est chaud", c'est-à-dire pour un certain paramètre, pour éviter à la fois de rester bloqué dans un optimum local, mais aussi des boucles infinies. Une première ébauche d'algorithme que nous proposons consiste à maximiser le profit avec probabilité de casse sous la contrainte initiale de minimisation de la puissance (et donc du coût du camion associé). On part alors de notre liste de puissances minimales, et on échange de façon aléatoire (ou pas: on pourrait combiner avec une approche naïve, la difficulté principale cependant est de permettre d'échanger un chemin pour plusieurs, difficulté que nous n'avons pas encore surmontée, etc.), de sorte à comparer. De cette façon, on optimise, le coût de camion étant bloqué au minimum d'emblée, sur la distance, c'est-à-dire en intégrant la probabilité de casse.
Un deuxième algorithme augmente le caractère aléatoire, en changeant cette fois aléatoirement, pour une liste de chemins donnée par leurs extrémités, la façon de les parcourir: cette fois, le coût du camion n'est plus une contrainte initiale, et on intègre plus fortement l'importance de la probabilité.

Il ne s'agit que d'ébauches, à peine testées, et loin d'être optimales, mais qui permettent de premières idées. L'idée évidente suivante serait de passer à un vrai recuit simulé ou tous les changements

seraient aléatoires, mais il faudrait alors beaucoup l'optimiser pour qu'il donne le résultat en temps réaliste. Bref, le problème est loin d'être résolu.