Rapport projet de programmation

Mathieu Janin, Aurore Pont

Le 06/04/2023.

Encadrant : Lucas Baudin

Table des matières

I - Première séance

Le but de cette première séance a été pour nous de comprendre la structure de graphe utilisée et d'implémenter les premières fonctions liées à la classe Graph. Nous avons également trouvé une solution naïve du problème de chemin minimal dans un graphe.

i) Implémentation de add_edge et graph_from_file

La fonction *add_edge* sert à ajouter une arête dans un graphe. Elle est en O(1) car il s'agit uniquement d'opérations élémentaires.

La fonction *graph_from_file* sert à construire un graphe à partir d'un fichier, en utilisant la fonction *add_edge*. Elle est en O(m) où m est le nombre d'arêtes du graphe, car on exécute *add_edge* pour chacune de ses arêtes.

ii) Composantes connexes : fonction *connected_components*

La fonction *connected_components* renvoie les composantes connexes du graphe donné en argument. Elle est récursive et en O(n+m) où n est le nombre de nœuds et m le nombre d'arêtes.

iii) Possibilité de trajet en considérant la puissance du camion

La fonction *get_path_with_power* donne l'itinéraire d'un trajet avec une puissance maximale donnée et renvoie None si un tel trajet est impossible. La fonction *get_path_with_power* que nous avons écrit n'est pas optimisée. Elle calcule, à l'aide d'une fonction récursive *fonc* le trajet nécessitant le moins de puissance et compare cette puissance minimale à celle du camion. Elle est en $O(m^n)$, où n est le nombre de nœuds et m le nombre d'arêtes. Elle n'est donc pas utilisable pour les gros graphes, ce serait trop long.

iv) Bonus : Réécriture de *get_path_with_power* en prenant en compte la distance

Le principe de la fonction *get_path_with_powerbonus* est qu'elle renvoie, s'il existe, le trajet ayant la distance minimale parmi les trajets ayant une puissance inférieure à celle donnée en argument. Tout comme *get_path_with_power*, cette fonction est en $O(m^n)$, où n est le nombre de nœuds et m le nombre d'arêtes.

Nous avons également implémenté des tests sur les petits graphes afin de vérifier que notre algorithme était bien valide.

v) Calcul de la puissance minimale requise pour un trajet

La fonction *min_power* utilise le même principe que *get_path_with_power*, mais renvoie la puissance minimale et le chemin correspondant dans tous les cas. Elle a donc une complexité en $O(m^n)$, comme *get_path_with_power* (en gardant les mêmes notations pour m et n).

vi) Tests des algorithmes

Nous avons testé tous nos algorithmes sur des petits graphes (les network.0x.in) et avons constaté qu'ils marchaient et renvoyaient les bons résultats.

II - Deuxième séance

Lors de cette deuxième séance nous avons constaté que notre algorithme de la séance 1 calculant la puissance minimale d'un trajet n'était pas optimisé. L'enjeu de la séance a alors été de chercher un algorithme optimisé pour ce calcul.

i) Estimation du temps de calcul de la puissance minimale

Afin d'estimer le temps de calcul de la puissance minimale pour l'ensemble des routes sur chacun des fichiers routes.x.in, nous avons implémenté la fonction *estimation_duree*, qui ne calcule la durée d'exécution de la fonction *min_power* uniquement pour les 5 premières routes, ce qui permet ensuite d'en faire une moyenne pour estimer le temps de calcul pour l'ensemble des routes du fichier.

Cependant, nous avons été dans l'incapacité de mesurer le temps d'exécution de la fonction *min_power* sur les fichiers routes.x.in car elle était trop lente, même si on ne l'utilisait que sur un seul trajet. On en a donc déduit qu'elle n'était pas optimisée. L'objectif de cette séance 2 a alors été d'optimiser cette fonction.

ii) Implémentation de l'algorithme de Kruskal

Afin d'optimiser notre fonction *min_power*, l'énoncé indiquait qu'il était utile de transformer nos graphes en arbres couvrants minimaux. Pour cela, nous avons implémenté l'algorithme de Kruskal, qui consiste à trier les arêtes par puissance croissante, à créer un graphe ne contenant que les nœuds et à ajouter une à une les arêtes si les nœuds concernés n'étaient pas déjà reliés.

Pour cela, nous avons donc dû implémenter par ailleurs la fonction *sont_relies* qui prend en argument un graphe et deux nœuds et renvoie True s'ils sont reliés et False sinon. Cette fonction est en O(m) où m est le nombre d'arêtes du graphe.

Ainsi, la fonction *kruskal* est en $O(n^2 + m^2)$ car la première partie de la fonction est en $O(n^2)$ tandis que la seconde partie est en $O(m^2)$.

Puis nous avons implémenter des tests afin de vérifier que la fonction était bien correcte et avons confirmé sa validité.

iii) Puissance minimale requise pour un arbre couvrant minimal

Grâce à l'algorithme de *Kruskal*, nous avions donc accès à des arbres couvrants minimaux. Il était alors plus simple d'implémenter une fonction *min_power* sur ce type de graphes. Nous avons alors créé la fonction *power_min_arbre_couvrant* adaptée, ne prenant en argument que des arbres (et non des graphes avec des boucles).

La fonction réalisée est donc en O(m) (m : nombre d'arêtes du graphe).

iv) Analyse de la complexité de ce nouveau *power_min_arbre_couvrant*

La fonction *duree_routes* que nous avons implémentée sert à calculer le temps d'exécution de notre fonction qui calcule la puissance minimale du trajet pour chaque routes des fichiers routes.x.in en utilisant les fonctions *kruskal* et *power_min_arbre_couvrant*.

On a alors obtenu de très bons résultats : pour le fichier routes.1.in, notre algorithme calculait l'ensemble des routes en 0,0026s, alors que notre algorithme de la séance 1 mettait un temps énorme pour nous renvoyer le temps trajet pour une seule route.

La méthode avec un arbre couvrant était donc bien plus optimale que la méthode naïve que nous avions implémentée lors de la première séance.

III – Séances 4 à 6

Le but de ces dernières séances est de répondre à la problématique finale :

- On a un budget donné
- Chaque route admet une utilité
- Chaque camion a un prix

L'enjeu est alors de répartir le budget pour acheter des camions et maximiser l'utilité des trajets.

L'idée est alors de faire un algorithme de type sac à dos.

i) Définition de la fonction *camions_from_file*

Tout d'abord on définit la fonction *camions_from_file* qui prend en argument un fichier et renvoie la liste des camions correspondants, afin de pouvoir ensuite exploiter le fichier.

Cette fonction est donc en O(k) où k est le nombre de camions du fichier.

ii) Définition de la fonction *recherche_dicho*

La fonction *recherche_dicho* sera ensuite utilisée dans la fonction *cout_des_routes*. Elle prend en argument une puissance et une liste de camions et renvoie le prix du camion capable d'avoir cette puissance.

(On suppose alors que le prix d'un camion est strictement croissant de sa puissance et les camions classés par puissance croissante.)

Pour cela, elle effectue une recherche dichotomique dans la liste des camions. La fonction *recherche_dicho* est alors en O(log(k)) où k est le nombre de modèles de camions possibles.

iii) Implémentation de la fonction *cout_des_routes*

La fonction *cout_des_routes* permet de rendre une liste exploitable pour l'algorithme du sac à dos.

Elle prend en argument un nombre x entre 1 et 10 et une liste de camions. Elle renvoie la liste des routes avec leur coût (calculé à partir la puissance requise par la route donnée par la fonction implémentée en séance 2 à laquelle on applique la fonction *recherche_dicho*) et le profit associé à la route.

La complexité de cette fonction est alors en (avec r le nombre de routes):

$O(m + n^2 + m^2 + r\times(m+\log(k))) = O(n^2 + m^2 + r\times(m+\log(k)))$

iv) Première implémentation de l'algorithme du sac à dos (méthode naïve)

La première version de l'algorithme du sac à dos est une fonction récursive de type bruteforce, qui prend en argument le budget total, la liste des trajets et la solution qui sera renvoyée, initialisée à une liste vide. L'algorithme vérifie que la liste étudiée n'est pas vide, puis compare, par un appel récursif sur la liste des trajets amputée du premier élément de la liste, les utilités associées aux situations suivantes :

- Le premier élément de la liste est retenu dans la solution, i.e un camion effectuera le trajet associé
- Le premier élément n'est pas retenu dans la solution

Puis, lorsque la solution est définie, l'algorithme renvoie la liste constituant la solution ainsi que l'utilité associée.

Cette fonction renvoie les bons résultats mais n'est pas optimale : il faut attendre plusieurs minutes pour traiter une liste d'une vingtaine de routes.

v) Seconde implémentation de l'algorithme du sac à dos (méthode optimisée)

Cette seconde version de l'algorithme du sac à dos utilise une approche dynamique, moins coûteuse car ne faisant pas appel à la récursivité. La liste des trajets est d'abord classée par ordre décroissant du coût. Puis, on fait le choix de normaliser le budget total par le coût minimal des trajets afin de réduire les temps de calculs, car lorsque l'on parcourra la liste range(1, budget_normalisé), on sera sûr de prendre en compte tous les trajets possibles car le pas « réel » sera égal au coût minimal. On crée ensuite un tableau prenant en ordonnée les trajets et en abscisse la liste range(1, budget_normalisé). On parcourt ce tableau afin d'attribuer à chaque case l'utilité maximale possible pour les paramètres suivants : le budget correspondant, en abscisse, et l'ensemble des trajets ayant un coût supérieur ou égal au trajet correspondant, en ordonnée. Ainsi, on obtient une matrice triangulaire inférieure et on est sûr que l'utilité pour chaque case est optimale pour les paramètres associés, car lors du remplissage du tableau on compare, comme avec bruteforce, les utilités correspondant aux situations dans lesquels le trajet considéré est pris en compte dans la solution ou non. Puis, on retrouve à partir du tableau la liste correspondant à la solution. On fait ensuite attention au fait que la solution ait un coût total inférieur au budget.

Cette fonction est en O(r × (budget//cout_min_chemin)) où r est le nombre de routes dans la liste (ie la longueur de la liste prise en argument).

On a alors exécuté cette fonction en l'appliquant au fichier routes.1.in et le résultat a été instantané, alors que le fichier contenait 100 routes. Cet algorithme est donc beaucoup mieux optimisé que *brute_force*.

Sources

Afin de réaliser le travail demandé, nous avons pu nous aider de wikipédia afin de mieux comprendre les concepts et algorithmes à utiliser. En outre, la chaîne Youtube Algomius nous a également été utile, et nous a permis de comprendre l'implémentation d'algorithmes classiques comme Sac à dos afin d'adapter ces algorithmes à la résolution du problème posé.