Compte-rendu Projet de Programmation Avancée

- Tom CARRERE -
- Antoine BROSSAS -

SE3
ANNÉE UNIVERSITAIRE 2020/2021

Introduction et vue d’ensemble

Pour le rendu de ce projet en programmation avancée, il nous était demandé de réaliser un programme complet qui permet de lire des fichiers CSV de taille conséquente contenant plusieurs milliers de vols avec diverses informations ainsi que les compagnies aériennes et aéroports associés à ces vols, de coder de nombreuses requêtes afin que l’utilisateur puisse y lire les informations qui l’intéresse à propos de certains vols.
Tout cela à l’aide d’une interface utilisateur ergonomique pour pouvoir utiliser les requêtes plus facilement.

Pour ce faire, dans notre répertoire GIT, nous disposons d’un makefile pour la compilation automatique, d’un bin pour l'exécutable, d’un répertoire data pour stocker les CSV, d’un dossier source src qui contient les programmes .c, d’un fichier include qui contient les .h dans lesquelles sont stockées les structures de données et prototypes des fonctions, d’un lib pour les .o sans oublier un README.md pour la description brève du programme.

Structures et chargement des données

Nos structures de données sont initialisées dans data.h, nous avons décidé de définir 3 structures de données principales qui sont flight (pour les vols), airport (pour les aéroports) et airline (pour les compagnies aériennes).
Ces 3 structures sont des structures de listes chaînées simples contenant dans chaque “case” respectivement la même structure que dans les fichiers CSV flights, airports et airlines.

Nous avons choisi d’utiliser des structures de liste chaînées simples pour pouvoir y ajouter facilement des données contrairement aux listes contiguës dont on ne peut pas augmenter la taille, d’où notre choix pour les listes chaînées afin de pouvoir ajouter simplement des vols sur plusieurs années voir différents pays donc différents aéroports et compagnies aériennes.

Pour ce qui est du chargement des données, cela se passe dans data.c, à l’aide de programmes de lecture de fichier que l’on a séparés en 3 catégories: flights, airports et airlines.

Ces programmes lisent les CSV ligne par ligne et y ajoutent les données dans nos listes, le programme sépare les différentes données à l'aide des virgules dans les CSV grâce à la fonction **strsep**. Mais parfois il arrive qu'il y ait des données manquantes comme dans flights.csv par exemple ou si un vol est annulé il n'y aura pas de durée de trajet ou bien d'heure d'arrivée, dans ce cas précis on mettra arbitrairement un "-1" dans ces cases et continuera de lire. Ce qui peut poser souci pour l'interprétation de certaines requêtes pour l'utilisateur.

Pour ces programmes là on parcourt chaque fichier CSV entièrement ligne par ligne, on est donc sur une complexité en O(n).

Requêtes

Nos requêtes sont stockées dans requetes.c, il y a aussi de nombreux sous-programmes utiles à l'exécution de ces requêtes. Passons maintenant en revue les différentes requêtes :

1) **show-airports <airline_id>**

   Pour réaliser cette requête on a créé plusieurs sous-fonction: **info_airport** (renvoie toutes les informations des aéroports contenus dans la liste passée en argument) , **extract_airport** (extrait un aéroport identifié par son code IATA) et un **ajout_tete** d'airports.
   La fonction va initialiser une liste_airports temporaire vide et passer en revue la base de données flights pour chercher les vols concernés par la compagnie rentrée en argument et s'ils ne sont pas déjà dans notre liste on ajoute les aéroports concernés à notre liste. Une fois fait, on affiche les résultats à l'aide de la fonction **info_airports.**.
   On a une complexité en O(n*o) avec n le nombre d'éléments de l_flights et o le nombre d'éléments de l_airports car à chaque fois que l'on parcourt les flights, on doit parcourir les airports avec la sous-fonction extract_airport. On notera qu'il y a beaucoup moins d'éléments dans l_airports que dans l_flights donc on est plus proche d'une complexité en O(n).

2) **show-airlines <port_id>** :

C’est la même chose que pour la première requête, on crée des sous-fonctions **info_airlines** et **extract_airlines** ici ainsi que l’ajout tête airlines.
La fonction procède de la même manière mais ici elle cherche les vols qui ont pour aéroport de départ celui passé en argument, et si c’est le cas de la même façon elle ajoute les airlines à la liste initialisée précédemment.
On a une complexité en O(n*o) avec n le nombre d’éléments de l_flights et o le nombre d’éléments de l_airlines pour les mêmes raisons. Pareil, il y a beaucoup moins d’éléments dans l_airlines (15) que dans l_flights (58592) donc on est plus sur une complexité en O(n).

3) **show-flights <port_id> <date> [<time>] [limit=<xx>]**

Pour réaliser cette requête on a créé plusieurs sous-fonction: **info flight** (renvoie toutes les informations des vols contenus dans la liste passée en argument avec ici un maximum de vols à afficher) et un **ajout_tete** flights.
Le programme commence par créer une liste de flight vide temporaire puis cherche les vols qui partent de port_id à la même date que d en parcourant la base de donnée des vols puis comme pour les exemples précédents, on vérifie si ce n’est pas dans notre liste. Complexité en O(n) car on parcourt seulement l_flights.

4) **most-delayed-flights**

Pour cette fonction, on utilisera un tableau de 5 vols qui stockera les vols les plus en retard à l’arrivée.
D’abord on crée 3 sous-fonctions: **init_tab_flights_arr_delay** (initialise un vol avec arr_delay à nul), **min_tab_flight** (retourne l'indice du vol ayant le plus petit retard parmi le tableau de flights passé en argument) et **elt_more_tab_flights** (si le retard du vol flight est supérieur au minimum des retards des vols de tab_flights alors il prend sa place).
Le programme initialise les éléments du tableau puis parcourt tous les éléments de flights et on récupère les 5 vols qui ont subi les plus longs retards à l'arrivée dans le tableau tab_flights il affiche ensuite tous les vols que le tableau contient.
On a une complexité en O(n).

5) **most-delayed-flights** :

Pour cette fonction on crée 3 sous-fonctions : **init_tab_airlines_delay** (initialise une struct Airline_delay{Flights[5], int mean_delay} avec mean_delay à nul), **mean_delay_airline** (renvoie la moyenne des retards à l'arrivée des vols associés à une compagnie mais ne prend pas les vols annulés ou déviés en compte pour le calcul de la moyenne) et **min_tab_airlines_delay** (retourne l'indice de la compagnie ayant la plus petite moyenne de retard parmi celles du tableau passé en argument).
La fonction initialise d'abord le tableau contenant 5 compagnies avec leur moyenne de retard puis parcourt l_airlines et on remplace les valeurs du tableau à chaque fois qu'on trouve une moyenne plus grande que le minimum de celles contenues dans le tableau et si la moyenne des retards associées à la compagnie tmp (variable temporaire) est plus grande que le min des moyennes des retards des compagnies stockées alors tmp prend sa place, puis on affiche le tableau.
Complexité en O(n*o) mais O(n) comme o<<n.

6) **delayed-airline <airline_id>**

Pour cette fonction on aura besoin de la sous-fonction **mean_delay_airline.**
Le code commence d'abord par trouver la compagnie associée au code iata que l'utilisateur va entrer. Le programme renvoie false si on ne trouve pas de compagnie et s'il trouve une correspondance après avoir parcouru airlines il renvoie le retard moyen.
Complexité en O(n*o) mais O(n) comme o<<n.

7) **most-delayed-airlines-at-airport <airport_id>**

Pour cette fonction on aura besoin de la sous-fonction **mean_delay_airline_at_airport** (renvoie la moyenne des retards à l'arrivée d'une compagnie passée en argument et identifiée par son code iata mais ne prend pas les vols annulés ou déviés en compte pour le calcul de la moyenne).
Le programme tout d'abord initialise un tableau contenant 3 compagnies avec leur moyenne de retard à l'arrivée de l'aéroport. Il parcourt désormais airlines et si la moyenne des retards associées à la compagnie tmp est plus grande que le min des moyennes des retards des compagnies stockées alors tmp prend sa place. Enfin, il affiche le tableau qui contient les airlines ayant le plus de retard en moyenne à l'arrivée.
Complexité en O(n*o) donc O(n) comme o<<n.

8) **changed-flights <date>** :

   Pour cette fonction on aura besoin de la sous-fonction **is_changed_flight_at_date** (renvoie 1 si le vol flight a été dévié ou annulé à la date d et 0 sinon).
   La fonction crée d'abord une liste temporaire qui va contenir tous les vols qu'on voudra afficher puis parcourt toute la liste des vols et quand on trouve un vol annulé ou dévié on l'ajoute à la liste des vols à afficher et enfin affiche les vols contenus dans la liste temporaire.
   Complexité en O(n).

9) **avg-flight-duration <port_id> <port_id>**:

   Pour cette fonction on aura besoin de 2 nouvelles sous-fonctions: **mean_airtime** (retourne la moyenne des airtime des vols dans la liste passée en argument) et **is_flight_between_airports** (renvoie 1 si le vol flight est entre les aéroports passés en argument).
   La fonction va d'abord créer une liste temporaire qui va contenir tous les vols entre airport1 et airport2 entré par l'utilisateur puis parcourir tous les vols pour trouver ceux entre les 2 aéroports passés en argument et on les ajoute à la liste temporaire et enfin calculer et afficher la moyenne des temps de vol.
   Complexité en $O(n^2)$.

10) **find-itinerary <port_id> <port_id> <date> [<time>] [limit=<xx>]**:

   Nous avons effectué cette requête de façon simplifiée. Elle affichera seulement un vol direct entre les 2 aéroports passés en argument à la date d.
   On crée d'abord une fonction intermédiaire **is_direct_flight_date** qui renvoie 1 si le vol rentré est entre les 2 aéroports passés en argument à la date voulue.
   On parcourt ensuite tous les vols et si la fonction intermédiaire renvoie 1 on arrête de parcourir et on affiche ce vol. Si elle n'a jamais renvoyé 1, on prévient l'utilisateur qu'aucun vol n'a été trouvé.

11) **find-multicity-itinerary <port_id_depart> <port_id_dest1> <date> [<time>] <port_id_dest2> <date> [<time>] ... <port_id_destN> <date> [<time>]:**

   Nous n'avons pas eu le temps d'effectuer cette requête.

Interface Utilisateur

La partie interface utilisateur du programme est stockée dans **main.c** dans le répertoire src. Le programme contenu dans le main va lire l'entrée standard strdin et donc les requêtes lignes par lignes jusqu'à-ce qu'il rencontre la commande **quit** auquel cas il sortira de l'interface utilisateur. Il va traiter chaque requête avec leurs arguments mais les arguments optionnels ne fonctionnent malheureusement pas. Nous avons d'ailleurs ajouté une commande **help** qui affiche les différentes requêtes disponibles et comment les utiliser.

Conclusion

En conclusion, nous avons choisi de travailler avec des liste chaînées pour leur confort d'utilisation et leur modularité mais il aurait été possible de réduire la complexité de certains programmes en utilisant d'autres types de structure comme par exemple les arbres binaires de recherche qui aurait pu descendre la complexité de certains programmes en O(log(n)) ou des tables de hachage.

Mais malheureusement nous n'avons pas réussi à aller au bout des requêtes demandées car nous avons notamment eu notre lot de problème, par exemple Tom ne pouvait pas git push sur sa machine Windows (il a honte) et devait ainsi passer par le site et ce n'était pas très ergonomique, il envoyait donc ses codes sur discord pour qu'Antoine puisse les push ce qui n'est pas super efficace..

Pour ce qui est de la critique de notre programme et des éventuels améliorations que l'on pourrait lui apporter et de ses limitations :

- Il ne prévient pas l'utilisateur en cas d'erreur de saisie d'une requête
- La requête 10 est simplifiée et la requête 11 n'est pas codée
- Les arguments optionnels ne marchent pas (nous pensons que c'est dû à l'utilisation de la fonction strtok qui d'ailleurs n'est pas conseillée au vu des différentes critiques visibles sur le net)
- Le traitement des valeurs manquantes peut être problématique pour l'interprétation que peut faire l'utilisateur