Projet Programmation 2021

DUMOND Thomas – METZGER Benjamin

Table des matières

1)Introduction

Le but du projet est d'analyser une base de données provenant du site *dblp.org* que nous devons parser, stocker dans une structure adaptée et y appliquer différents algorithmes de graphes .Pour cela, nous avons réalisé un programme en C censé répondre au fonctionnement donné précédemment. Chaque sous fonction du programme est documentée dans un fichier .h qui renseigne sur son utilisation et ce qu'elle fait.

2)Présentation du programme

a) Fonctionnement global du programme

Le programme principal est le programme *dblp-parsing* situé dans le dossier *bin*. Pour utiliser le programme, il y a plusieurs options de lancement. Premièrement, il faut lancer le programme afin de parser une base de données de type html et stocker la structure du graphe dans un fichier binaire pour la traiter plus tard et surtout pour ne pas avoir à regénérer le graphe de multiples fois si l'on veut par exemple réexécuter le programme en recherchant différents auteurs car la partie parsage de la base de données et stockage dans la structure graphe est la partie qui prends le plus de temps dans l'exécution du programme (cela peut durer jusqu'à 15 min en utilisant le fichier *dblp.xml* en entrée, et dans ce cas on exécutera la commande :

```
./dblp-parsing -d dblp.xml -o binaire
```

afin de stocker le graphe du fichier *dblp.xml* dans le fichier binaire.). Une fois cette première opération réalisée, on peut alors réexécuter le programme afin de réaliser plusieurs opérations à partir de notre fichier binaire. Si on veut rechercher la distance entre 2 auteurs on exécutera la commande :

```
./dblp-parsing -i binaire -a auteur1 -b auteur2
```

Cette commande renverra la distance en nombre de titres séparant les 2 auteurs.

Si on veut savoir tous les titres des œuvres réalisées par un auteur, on exécutera la commande :

```
./dblp-parsing -i binaire -t auteur
```

Cette commande renverra la liste des tous les titres des ouvrages de l'auteur renseigné.

Si on veut connaitre les autres commandes disponibles, on exécutera la commande :

```
./dblp-parsing -h
```

Cette commande affichera à l'écran toutes les options de lancement disponibles et les arguments attendus.

b) Choix des structures

Toutes les structures utilisées sont placées dans le fichier `struct.h`.

i) Structure de données pour le parsage

Pour le parsage, on utilise la structure suivante :

```
typedef struct data_t {
    int nbAuteurs;
    char *auteurs;
    char *titre;
} data_t;
```

Cette structure est utilisée lors du parsage et initialisée à chaque fois qu'un nouvel article est détectée (dès que l'on détecte la balise ouvrant article dans le fichier xml a parser). Cette structure est composée de 3 éléments :

- Une variable `nbAuteurs` qui va compter le nombre d'auteurs qui ont écrit l'article.
- Une chaine de caractère `auteurs` contenant la liste de tous les auteurs ayant écrit l'article tous les uns à la suite des autres, séparés par le caractère « **;** ».
- Une chaine de caractère `titre` contenant le titre de l'article.

ii) Structure du graphe

Pour la partie graphe, on utilise la structure suivante :

```
typedef struct graphe_t {
    char **liste_auteurs;
    size_t nb_auteurs;
    char **liste_titres;
    size_t nb_titres;
    size_t **liste_sucesseurs;
    size_t *liste_nb_liens;
    size_t *hachage_auteurs[100000];
    size_t nb_auteurs_hache[100000];
} graphe_t;
```

Cette structure est utilisée pour stocker en mémoire le graphe et les données ajoutées au graphe sont celles contenues dans la structure `data_t` qui contient les éléments de l'article en cours de traitement. Cette structure est composée de 8 éléments :

- Un tableau de chaine de caractères contenant la liste de tous les auteurs contenus dans le fichier à parser (par exemple, on peut avoir `liste_auteurs[i]="auteur de l'index i"` et chaque auteur ne peut apparaitre qu'une seule fois dans la liste).
- Une variable `nb_auteurs` contenant le nombre total d'auteurs dans le fichier à parser (cela correspond aussi au nombre d'éléments du tableau `liste_auteurs`).
- Un tableau de chaine de caractères `liste_titres` contenant les titres de tous les articles avec en plus pour chaque titres les index des auteurs ayant écrit cet article (par exemple on peut avoir `liste_titres[i]="Titre|0|1"` si les auteurs `liste_auteurs[0]` et `liste_auteurs[1]` ont écrit l'article nommé *Titre*).
- Une variable `nb_titres` contenant le nombre total de titres dons le nombre total d'articles dans le fichier à parser (cela correspond aussi au nombre d'éléments du tableau `liste_titres`).
- Un tableau de tableau de variables `liste_sucesseurs` qui contient pour chaque auteur la liste de ses auteurs voisins qui correspond aux auteurs avec lesquels il a coécrit un article(par exemple, `liste_sucesseurs[i][0]` renvoie le premier auteur avec lequel l'auteur n° i a écrit un article …).
- Un tableau de variables `liste_nb_liens` qui contient pour chaque auteurs le nombre d'auteurs avec lesquels il a coécrit (cela correspond donc aussi aux nombres d'éléments contenus pour chaque élément du tables `liste_sucesseurs` donc par exemple

`liste_nb_liens[i]` renvoie le nombre d'éléments contenus dans `liste_sucesseurs[i]`).

- Un tableau de tableau de variables `hachage_auteurs` qui pour chaque hache de l'auteur donné contient la liste des index des auteurs avec ce hache(cela permet donc en cas de collisions de pouvoir comparer les auteurs avec le même hache et le tableau est initialisé avec une taille de 100000 car notre fonction de hachage nous renvoie un nombre modulo 100000, ainsi on avec le hache obtenu par la fonction de hachage on va aller à l'index du hache et comparer l'auteur haché avec tous les index déjà contenus a pour ce hache si l'auteur existe déjà alors on récupère son index sinon on l'ajoute à l'index du hache à la suite).
- Un tableau de variables `nb_auteurs_hache` qui pour chaque hache nous renvoie le nombre d'auteurs avec ce hache (de taille 100000 car notre fonction de hachage renvoie un entier modulo 100000). Par exemple, `nb_auteurs_hache[12]` nous renvoie le nombre d'auteurs pour lesquels notre fonction de hachage nous renvoie 12.

Ainsi, pour stocker le graphe en mémoire, nous avons choisi de réaliser une liste d'adjacence afin de stocker les voisins de chaque auteur. Nous avions d'abord pensé à réaliser une matrice d'adjacence qui aurait donc permis un traitement plus rapide pour les algorithmes de plus courts chemins mais son stockage en mémoire aurait été trop imposant c'est pourquoi nous avons optés pour une liste d'adjacence. De ce fait, pour le traitement de de cette liste, il nous fallait donc le nombre de successeurs pour chaque auteur d'où l'utilisation d'un autre liste annexe qui renvoie le nombre de successeurs pour chaque auteur.

c) Fonctionnement des sous programmes

i) Parsage

Les différentes fonctions concernant le parsage sont contenus dans les fichiers `parsage.c` et `xmlp.c` et sont documentés dans les fichiers `parsage.h` et `xmlp.h`. Pour commencer, la fonction de parsage principale est la fonction `parse` qui est la fonction qui détecte les balises ouvrante et fermantes et renvoie les bonnes informations aux bonnes fonctions. Tout d'abord, la fonction ouvre le fichier donné en argument qui sera le fichier à parser. Ensuite, on place la tête de lecture à la fin du fichier et on le remet au début pour connaitre le nombre de caractères à traiter et donc pouvoir afficher le pourcentage du traitement du parsage du fichier dans le terminal avec la fonction `printAvancement`. Ensuite, on crée un buffer qui nous permettra de lire ce qu'il y a entre les balises et donc dans un premier temps de lire le tag mais aussi de récupérer les données entre les tags. Ainsi, on commence par attendre le caractère "<" et ensuite si le caractère suivant est "/", on sait qu'il s'agit d'un tag fermant, donc on lit le fichier jusqu'au caractère ">" et on aura donc récupéré le tag que l'on envoie dans la fonction `handleCloseTag` qui agit différemment en fonction du tag. Si le caractère suivant n'est pas "/", il s'agit d'un tag ouvrant donc et on aura récupéré le tag que l'on envoie dans la fonction `handleOpenTag` qui agit différemment en fonction du tag. Après avoir lu le caractère ">", on lit chaque caractère jusqu'à lire un caractère "<" et l'on envoie les caractères lus dans la fonction `handleText` qui va pouvoir remplir notre structure pour l'article `data_t`. Et on refait tout ça jusqu'à arriver à la fin du fichier. Ensuite, le parsage passe par les différents fonctions `handleOpenTag`, `handleText`, `handleCloseTag`. Ainsi, on ne récupère que le texte si les balises sont des balises auteurs et titres afin e récupérer uniquement les titres et les auteurs que l'on

stocke et après la fermeture du tag titre, on envoie toutes nos informations concernant l'article dans notre structure graphe en passant par la fonction `addGraphe`.

ii) Hachage

Pour le hachage, le tableau de hachage est contenu dans la structure graphe et les fonctions de hachages sont contenues dans le fichier `hachage.c` et documenté dans le fichier `hachage.h`. La fonction utilisée pour le hachage est la fonction hache qui prends en arguments une chaine de caractère à hacher et qui nous retourne un entier modulo 100000 correspondant au hash de la chaine. La fonction est simple, pour chaque caractère, on le multiplie par un nombre premier a et on lui ajoute un autre nombre premier b et on somme le tout pour chaque caractère de la chaine.

La fonction la plus importante pour le hachage est la fonction `indexation_auteur` qui prends en arguments une liste d'auteurs et qui renvoie l'index des auteurs s'ils existent déjà ou -1 sinon. Pour chaque auteur de la liste, il le hache avec la fonction de hachage et en cas d'auteurs avec le même hash, on récupère les index des auteurs et on compare notre auteur en cours avec les auteurs avec le même hash. S'il s'agit des mêmes auteurs, alors on récupère son index sinon son index sera -1 et on laissera la fonction `addGraphe` s'occuper de l'ajout de l'index à la table de hachage.

iii) Graphe

Pour le graphe, les fonctions sont placées dans le fichier `graphe.c` et documentées dans le ficher `graphe.h`. La fonction principale est `addGraphe` qui prends en argument la structure data contenant les informations de l'article et qui les ajoutes au graphe. Elle commence par créer une liste qui contient tous les auteurs de l'article et le hache pour récupérer leur index s'ils existent. Ensuite pour chaque auteur, s'il n'existe pas on les ajoute à la liste auteurs du graphe et on récupère leurs index et on lui ajoute ses voisins de l'article courant dans la liste des successeurs.

Les fonctions suivantes sont les fonctions d'exportation et d'importation du graphe depuis un fichier externe. Pour exporter le graphe dans un fichier externe, on passe par la fonction `printGraphe` en lui indiquant le graphe à sauvegarder et le fichier dans lequel sauvegarder. Elle commence par imprimer sur la première ligne le nombre d'auteurs présents dans le graphe, sur la deuxième ligne, elle affiche le nombre d'auteurs voisins que possède chaque auteur, sur la troisième ligne, elle affiche pour chaque auteurs l'index de ses voisins, sur la quatrième ligne, elle affiche la liste des auteurs, sur la cinquième ligne, le nombre de titres et sur la sixième ligne, la liste des titres avec pour chaque titre l'index des auteurs ayant écrit l'article.

La fonction `importGraphe` qui prends en argument un graphe vide et un fichier d'entrée et qui lit le fichier pour importer les données du fichier directement dans la structure. Son fonctionnement repose sur le fonctionnement inverse de la fonction `printGraphe`.

iv) Dijkstra

Dans un premier temp, j'ai commencé à implémenter un code afin d'obtenir le chemin le plus court entre deux auteurs avec une matrice d'adjacence. Mais à la suite du problème mémoire rencontré, j'ai dû changer toute la structure du code en gardant le même squelette.

Dans un premier temp, je teste la robustesse du code, c'est à dire les cas grotesques. Le premier test compare le nom des deux auteurs afin de pouvoir régler le cas trivial d'un test entre le même auteur. Puis, j'assigne comme numéro d'auteur au deux auteurs –1 qui est impossible car les numéros commencent à 0. Je parcours la liste d'auteur afin de pouvoir récupérer le numéro d'auteur de chacun des deux auteurs puis je fais un second test qui va voir si l'un ou les deux auteurs existent. Tout simplement, si l'un des deux numéros est égale à -1 l'auteur n'existe pas et le programme s'arrête. Une fois ces numéros récupérés, je parcours la liste contenant le nombre de liens contenue dans la structure graphe. Je vérifie à l'index de la source et du puit s'il a au minimum une œuvre co-écrites car si c'est le contraire l'algorithme ne sert à rien. Le 3-ème et dernier test simple vérifie s'il existe un lien direct donc d'un coût unitaire entre la source et le puit. Pour commencer l'implémentation de cet algorithme de plus court chemin, je crée deux buffers de type Int. Comme tout chemin est de coût unitaire, je n'ai pas à chercher les liens de coût les plus bas. Je pars de l'auteur un et je vais vers le premier voisin présent dans la liste de successeurs. Je test depuis ce successeur s'il est en lien avec le puit. Si cela est le cas, je note dans mon tableau de Dijkstra à l'indice du numéro buffer le prédécesseurs - coût total puis à l'indice du puit de même. Si ce n'est pas le cas je retourne en arrière et vérifie les autres voisins directs. Si aucun n'est en contact avec, alors on s'aventure plus en profondeur au fur et à mesure en changeant les numéros des deux auteurs buffer. Si c'est aucun chemin trouvé j'affiche un message d'erreur sinon j'affiche le chemin en remontant en sens inverse le tableau Dijkstra.

d) Gestion des signaux

Pour la gestion des signaux, nous avons simplement changé l'action par défaut de SIGINT (CTRL+C) de manière à afficher un message dans le terminal avec le nombre de fois que la commande a été réalisée durant le programme et au bout de la 5ème fois, cela tue le processus. Son fonctionnement repose sur la fonction `sigaction` qui provient de la librairie `signal.h` . Ainsi, lors de la réception du signal SIGINT, le programme exécute la fonction `signalHandler` qui réalise l'affichage et tue le processus.

e) Programme de tests

Pour les différents tests, j'ai réalisé une fonction `compare` qui permet de comparer 2 fichiers à partir de la commande `cmp` du terminal bash. Ainsi, le fichier `tests.c` permet de tester quelques fonctions de bases comme l'exportation de la structure du graphe dans un fichier binaire, l'importation dans la structure graphe à partie d'un fichier binaire et aussi la fonction `decode_html` permettant de décoder certains caractères comme les « &ouml ; » en « ö ». Les résultats de ces tests sont affichés sur la sortie erreur du terminal avec des couleurs en cas de succès ou d'échec.

3)Problèmes rencontrés et solutions apportées

a) Stockage du graphe en mémoire

Pour stocker le graphe en mémoire, nous avions d'abord pensé à réaliser une matrice d'adjacence en stockant -1 si les auteurs n'ont pas d'articles en commun et ensuite stocker l'index du titre de l'article si les auteurs on écrit un article en commun. Plusieurs problèmes se sont alors posés, si les

mêmes auteurs ont plusieurs articles en commun, alors ça sera le dernier titre en commun qui sera stocké car le dernier titre remplacera l'ancien mais aussi des problèmes pour la mémoire car on stocke beaucoup d'informations inutiles (tous les -1 stockés ne servent à rien et prennent de la place pour rien). Nous avons donc eu des problèmes de dépassement de mémoire disponible en stockant des int dans la matrice. Nous avions envisagé de stocker des booléens a la place des int avec 0 si les auteurs n'ont pas d'articles en commun et 1 si les auteurs ont au moins un article en en commun mais la taille mémoire aurait été encore trop imposante d'où le passage à la liste de successeurs qui certes prendra plus de temps pour déterminer le plus court chemin mais qui sera beaucoup moins volumineuse en mémoire.

b) Temps de parsage

Lors de la première exécution du programme de parsage et de réalisation du graphe, je me suis rendu compte que le programme n'avait traité que 20% de la base de données *dblp.xml* en 4 heures. Cela était due au fait de chercher pour chaque auteur s'il existe déjà dans la structure du graphe. En effet, le premier programme effectuait cette tache de manière naïve en parcourant toute la liste et donc si l'auteur n'existait pas encore, alors on devait parcourir tous les auteurs déjà existants, la complexité de la recherche est alors du type $O(n^2)$. Pour résoudre ce problème, nous avons donc utilisé une table de hachage dont son fonctionnement est documenté ci-dessus. Ainsi, on ne parcoure pas n fois la liste des auteurs mais un nombre bien plus faible et uniquement en cas de collision. La complexité de la recherche de l'auteur se rapproche donc d'une complexité linéaire du type O(n) et donc réduit drastiquement le temps de réalisation du graphe en passant a environ 15 min pour le fichier *dblp.xml*. On pourra réduire encore drastiquement ce temps en utilisant une fonction de hachage plus complexe que celle que nous avons réalisé afin de réduire le nombre de collisions et donc réduire le nombre de comparaisons à réaliser ou bien on pourra encore utiliser une $2^{ème}$ fonction de hachage en cas de collision qui si les auteurs sont différents renverra bien deux hashs différents.

4)Synthèse

Pour conclure, le programme actuel est fonctionnel mais uniquement sur des petits fichiers. Pour qu'il fonctionne avec la grosse base de données, il faudrait résoudre les problèmes de segmentation fault qui apparaissent lorsque le nombre d'articles à traiter devient conséquent. En effet, le fait d'avoir changé le code a de multiples reprises nous as rajouté un nombre d'erreurs conséquentes et pour lesquelles nous n'avons pas trouvé de solutions viables (par exemple les "invalid read of size 1" sur un `strlen` avec une chaine de caractère qui est pourtant bien allouée). En effet, nous avons d'abord dû adapter nos fonctions de graphes en passant d'une matrice d'adjacence à une liste d'adjacence puis nous avons dû les réadapter avec la table de hachage ce qui fait que les fonctions de graphes ne sont pas les plus optimales possibles.