Lyes SID ALI
Tran Dang Quang Le

Rapport projet Structures De Données

Sommaire:

1. Présentation du projet :

Durant ce projet, nous allons nous intéresser à la reconstitution d'un réseau de fibres optiques via différentes façons, à partir de fichiers « .cha » qui représente une instance de notre problème.

Un réseau est un ensemble de câbles représentés par des chaînes et de nœuds représentés par leurs coordonnées (x,y, chaque chaîne relie une commodité (une paire de nœuds).

Dans la première partie, nous supposerons que les nœuds sont des points et les câbles des chaînes, afin de simplifier dans un premier temps, ceci nous permettra par la suite de pouvoir traduire un fichier .cha en une instance Reseau. Nous pourrons grâce à cela comparer nos résultats et voir quelles modifications apporter.

Dans la deuxième partie, nous allons entamer le coeur du projet en utilisant une structure Reseau, et le but est de pouvoir comparer ces différentes structures de données : Liste chaînée, Tables de hachage, Arbre quaternaire. Pour cela le test à faire est le suivant :

chercher si un élément appartient à un ensemble

Dans notre cas, chercher si un nœud appartient au réseau, en intégrant pour chaque cas de structure de données sa propre structure et adapter l'algorithme .

Dans la troisième partie, on va comparer les résultats obtenus auparavant afin de déterminer quelle est la structure la plus adaptée en terme de temps d'exécution en faisant différents test pour chaque structure.

Dans la quatrième partie, on s'attaque aux graphes, le but est l'optimisation et la réorganisation du réseau, en le reconstituant sous forme de structure de graphe.

2. Organisation du code et fichiers :

_Le projet contient trois fichiers éxécutables :

ChainMain : Pour tester la reconstitution d'un fichier .cha en Chaines

ReconstitueReseau : Pour reconstituer le réseau en choisissant parmi les structures de données

main : Pour comparer et analyser les résultats obtenus précedemment

_ Il contient également les fichiers textes depuis lesquels on va partir :

00014_burma.cha : Pour les chaines
00014_burma.res : Pour le reseau

_ On trouvera des fichiers fournis comme par exemple **SVGWriter.c** qu'on utilisera pour afficher notre réseau sous forme de fichier html

_ Concernant les autres fichiers .c , chacun possède un fichier header .h où sont définies toutes les structures utilisées ainsi que la déclaration des fonctions, cela permet une programmation modulaires car tout ces fichiers sont liés, il est alors nécessaire d'utiliser un makefile.

_ le fichier **Makefile** compile tout les fichiers avec l'option -c afin d'avoir un fichier .o, pour finalement avoir les trois exécutables cités ci-dessus. Des erreurs de compilation peuvent survenir à cause de l'utilisation des fonctions la bibliothèque math.h, c'est pour cela que nous avons rajouté l'option **-lm** aux fichiers executables. Chauqe fichier contient une cible dans le makefile, et il y a également la cible **clean** qui nous permet de remettre à zéro le répertoire en supprimant tout les fichiers résultants de la compilation et l'exécution.

_ Les fichiers résultants des exécutions sont ceux là :

Instance.html : représente le réseau stocké dans une structure Chaines, dans un fichier html.
Reseau_LC.html : représente le réseau stocké dans une structure liste chainée, dans un fichier html.
Reseau_Hachage.html : représente le réseau stocké dans une structure table de hachage, dans un fichier html.
Reseau_Arbre.html : représente le réseau stocké dans une structure d'arbre quaternaire, dans un fichier html.

new_chaine.txt : représente le résultat de la fonction d'écriture d'un réseau stocké dans une structure Chaines.
new_reseau_LC.txt : représente le résultat de la fonction d'écriture d'un

réseau stocké dans une structure liste chainée.
new_reseau_Hachage.txt: représente le résultat de la fonction d'écriture d'un réseua stocké dans une structure table de hachage.
new_reseau_Arbre.txt: représente le résultat de la fonction d'écriture d'un réseua stocké dans une structure arbre quaternaire.

TempsCalculEn3methode.png : représente le résultat de la comparaison du temps d'exécution en utilisant les trois méthodes de structure de données.
TempsCalculArbre.png : représente le graphique obtenu du temps d'exécution du fichier main en fonction du nombre de chaînes en utilisant la structure d'arbre quaternaire.
TempsCalculHachage.png : représente le graphique obtenu du temps d'exécution du fichier main en fonction du nombre de chaînes en utilisant la structure table de hachage.
TempsCalculListe.png : représente le graphique obtenu du temps d'exécution du fichier main en fonction du nombre de chaînes en utilisant la structure liste chaînée.
Timedata.txt : c'est le fichier qui contient le résultat du temps d'exécution des trois fonction de reconstitution du réseau.

Ces fichiers sont ceux qui vont nous permettre de vérifier si nos résultats sont correctes, en les comparant entre eux.

3. Fonctions principales et algorithmes :

Le coeur du projet tourne autour de la reconstitution du réseau, dont le pseudo code est :

On utilise un ensemble de nœuds V qui est initialisé vide:
On parcourt une à une chaque chaîne:
    Pour chaque point p de la chaîne:
        Si p 6 ∈ V (on teste si le point n'a pas déjà été rencontré auparavant)
            On ajoute dans V un nœud correspond au point p.
        On met à jour la liste des voisins de p et celles de ses voisins.
        On conserve la commodité de la chaîne.

La ligne en gras est celle qui nous intéresse, car on cherche à déterminer si un un élément appartient à un ensemble, si un nœud appartient à un réseau, le reste du code est quasi le même pour les 3 structures de données, mais pour cette ligne la nous devons nous adapter pour chaque cas.
Avant de voir cela plus en détail, on va passer en revu quelques fonctions importantes.

lectureChaine(FILE* f) : (Chaine.c)

cette fonction est primordiale car c'est grâce à elle que nous pouvons lire un réseau puis le retranscrire en différentes structures, le but ici est de simplement parcourir les lignes du fichier, tout en allouant de l'espace mémoire à chaque point et chaîne rencontrée, et attribuer les valeurs lues aux variables de la structure, puis finalement, on retourne une Chaines.

libererChaines(CellChaîne *chaine): (Chaine.c)

cette fonction permet de libérer la mémoire occupée par une Chaines, afin d'éviter les fuites mémoire lors de l'exécution, elle s'appuie sur d'autres fonctions de libération de la mémoire, en libérant pas à pas chaque case alloué.

nbLiaisons(Reseau *R): (Reseau.c)

On parcours la liste des nœuds du réseau, ainsi que la liste des voisins de chacun d'eux, et on incrémente la variable res, puis on divise le résultat par deux car chaque liaison est comptée deux fois, car la relation de voisinage est bidirectionnelle.

libererReseau(Reseau *res) : (Reseau.c)

cette fonction libère toute la mémoire occupée par un réseau, en libérant une a une les commodité, puis chaque nœud ainsi que ses voisins.

mettreAJourVoisins(Reseau *R, Noeud* n, Noeud* v) : (Reseau.c)

cherche si le nœud v n'est pas déjà voisin de n, puis crée une liaison avec le nœud voisin, en les ajoutant mutuellement à la liste des voisins.

CreerTableHachage : (Hachage.c)

Alloue de la mémoire à une TableHachage puis initialise tout ses éléments a NULL. Remarque : pour la fonction de hacahge, on a choisi d'utiliser le type unsigned long, afin d'éviter les erreur quand on manipule des données trop volumineuse ou pour éviter d'avoir un nombre négatif.

libererTableHachage(TableHachage * H) : (Hachage.c)

libère la mémoire occupée par une table de hachage, en libérant tout les nœuds du tableau, puis le tableau lui même.

determinerDirection(ArbreQuat* arbre, double x, double y) : (ArbreQuat.c)

cette fonction retourne le fils d'un arbre, en déterminant grâce aux coordonnées x et y quel fils retourner parmi les 4 directions (ne-no-se-so). Nous utiliseront cette fonction pour simplifier les prochaines.

InsererNoeudArbre : (ArbreQuat.c)

Cette fonction distingue trois cas :

- *Arbre vide :* On initialise sa longueur et hauteur en divisant par deux celle du parent, puis on cherche à savoir les coordonnées du centre de la cellule en fonctions de celles du nœuds passé en paramètre, et de l'arbre parent.
- *Feuille :* on garde un pointeur sur le nœud se trouvant dans la feuille, puis on l'affecte a NULL, en on insère ce nœud et le nouveau nœud dans l'arbre de façon récursive, on utilisera la fonction determinerDirection pour savoir dans quel coté insérer les nœuds.
- *Cellule interne :* On procède de manière récursive en passant en jusqu'à arriver au cas arbre vide, on choisi la direction grâce a la fonction determinerDirection.

On revient maintenant à la reconstituant du réseau, nous allons étudier deux fonctions :

La recherche et création d'un nœud :

- ***Liste :***

la fonction parcours la liste des nœuds du réseau, si elle trouve un nœud correspondant aux coordonnées, elle le retourne, sinon on crée un nouveau nœud, ainsi qu'une nouvelle cellule qu'on fera pointer sur le nœuds, et on insère la cellule en tête de liste, puis pour garder le pointeur sur la tête de liste on fait pointer R→neouds sur cette nouvelle cellule.

- ***Table de hachage :***

on accède directement à la case de la table qui correspond au nœud grace à la fonction de hachage, puis on parcours la liste de cette case (car résolution des collisions par chainage), si on trouve le nœud on le retourne, sinon on crée un nouveau nœud et sa cellule qu'on insère en tête de liste de la case correspondante dans la table de hachage.

- ***Arbre Quat :***

  _Si l'arbre est vide on crée le nœud, on l'ajoute à la liste de nœuds du réseau puis dans l'arbre grâce à la fonction insesrerNoeudArbre()

  _Si c'est une feuille on regarde si le nœud stocké dans la feuille est celui recherché alors on le retourne, sinon on crée un nouveau comme pour le cas d'un arbre vide.

  _Si c'est une cellule interne, on retourne le résultat de la récursivité de la fonction en parcourant l'arbre jusqu'à tomber sur une feuille ou un arbre vide.

La reconstitution du réseau :

Cette fonction est quasi la même sur les trois structures, à quelques détails près.

_Voici son fonctionnement :

On commence par allouer un Réseau en initialisant ses champs, puis pour chaque chaîne on parcours ses points, en affectant à la variable extrA le premier point (noeud) uniquement lors du 1er tour de boucle du parcours des points, ceci afin de garder un pointeur sur la commodité, puis pour chaque point on va vérifier si il existe déjà sinon on le crée, tout en prenant soin de créer la liaison avec le point précédent grâce à la fonction mettreAJourVoisins( ), une fois la boucle des points terminé, on affecte à extrB la valeur de la variable d'itération de la boucle qui correspond au

dernier point, puis on crée la commodité en l'ajoutant en tête de liste à la liste des commodités du réseau. Et on refait ça pour chaque chaîne.

_Voici les différences qu'il va falloir intégrer:

- ***Liste :***
  Pas d'ajout ou modification à faire par rapport au fonctionnement de base de la fonction.

- ***Table de hachage :***
  On crée une table de hachage afin de stocker les nœuds, lors de la recherche du nœud, si on le trouve on le retourne sinon on l'ajoute dans la table créée précédemment, afin d'optimiser la recherche en cherchant directement dans cette table. À la fin, on libère la table.

- ***Arbre Quat :***
  On crée un arbre en déterminant ses coordonnées en partant des points de la chaîne pour trouver le max et le min, puis pour chaque point, on va déterminer dans quel fils insérer le nœud dans l'arbre avec la fonction determinerDirection( ), afin d'optimiser la recherche en cherchant directement dans l'arbre si le nœud est présent ou pas. À la fin, on libère l'arbre.

4. Analyse des résultats :

1) Après avoir exécuté la fonction ./main, le temps de calcul CPU de la reconstruction du réseau à l'aide de différentes méthodes varie considérablement. Les résultats sont enregistrés dans le fichier timedata.txt :

- Première colonne : Elle contient le nombre total de points. Ici, le nombre de points par chaîne est de 100, et le nombre de chaînes varie de 500 à 5000 par pas de 500. Ce nombre augmente à chaque itération.

- Deuxième colonne: Liste chaînée, les calculs utilisant la méthode des listes chaînées peuvent nécessiter plusieurs heures d'exécution si les données d'entrée sont volumineuses. Selon les données de la deuxième colonne du fichier timedata.txt, la première exécution de cette méthode dure 2,05 secondes, mais après 10 itérations, sa durée augmente progressivement pour atteindre environ 10 000 secondes (soit environ 3 heures d'exécution) lors de la dernière itération.

Cela est compréhensible car avec la méthode des listes chaînées, la complexité dans le pire des cas est de □(□2) pour trouver un point dans une chaîne y compris n éléments.

- Troisième colonne: Table de hachage, les calculs utilisant la méthode des tables de hachage réduisent considérablement le temps de calcul CPU. Selon la troisième colonne du fichier timedata.txt, le temps de traitement de la table de hachage est nettement plus rapide que celui de la liste chaînée, passant de 0,14 seconde à 115 secondes lors de la dernière itération. Bien que cette méthode soit plus rapide, elle peut entraîner des collisions si la fonction de hachage n'est pas optimisée. Il est également important de faire attention à la taille de la table de hachage en fonction du nombre d'éléments. La complexité dans le pire des cas est de □(□).

- Dernière colonne: Arbre binaire, les données de la dernière colonne du fichier timedata.txt montrent le temps d'exécution de la méthode des arbres binaires. Parmi les trois méthodes utilisées pour stocker des données, cet algorithme est le plus optimisé en termes de temps et de mémoire. Le temps d'exécution est très court, passant de 0,025 seconde pour la première exécution à 0,429 seconde pour la dernière, après 10 itérations. La complexité dans le pire des cas est de □(log□(□)).

2) Graphiques : Avec les données du fichier, je peux représenter les résultats graphiquement dans le fichier TempsCalculEn3Methode.png. J'ai également créé des graphiques distincts pour chaque méthode : TempsCalculListe.png, TempsCalculHachage.png, et TempsCalculArbre.png.

5. Réponses aux questions :

3.3) Le code à été testé sur plusieurs instances qui ont été fournies (USA, burma, pla), le code fonctionne parfaitement, on a obtenu les résultats attendus.

4.2) la fonction clef nous semble approprié car après les avoir testé sur plusieurs points, on obtiens des résultats très variés allant jusqu’à 500 (taille de la table) , afin d’éviter au mieux les collisions. Cependant on a eu un problème avec le fichier USA car la clé de hachage était invalide (chiffre trop grand) , on a pu résoudre ce problème en utilisant le type unsigned long qui était plus adapté.

6.4) Réponse dans la partie : 4. Analyse

7.3) Pour améliorer la fonction reorganiseReseau( ), on peut utiliser un algorithme de recherche du chemin le plus court plus efficace, comme par exemple A* ou Dijkstra, dont voici le fonctionnement :

- Dijkstra :

Il fonctionne en sélectionnant le nœud non visité le plus proche du nœud de départ à chaque étape et en mettant à jour les distances les plus courtes jusqu'à ce que tous les nœuds aient été visités.

Complexité :

La complexité temporelle de Dijkstra dépend du type de structure de données utilisée pour maintenir les distances les plus courtes. la complexité temporelle est généralement de O((V + E) log V), où V est le nombre de nœuds et E est le nombre d'arêtes dans le graphe.

- A* :

Il évalue chaque nœud en utilisant le coût du chemin depuis le nœud de départ et une estimation du coût restant (heuristique) jusqu'à la destination. A* utilise une file de priorité pour explorer les nœuds avec les coûts les plus faibles en premier.

Complexité :

La complexité temporelle d'A* dépend de l'efficacité de l'heuristique utilisée. Dans le pire des cas, A* peut avoir une complexité exponentielle, mais dans des cas pratiques avec une heuristique admissible (qui ne surestime pas le coût restant), la complexité temporelle est généralement proche de celle de Dijkstra, soit O((V + E) log V).

6. Conclusion :

Durant ce projet, on s’est confronté à plusieurs difficultés, notamment beaucoup d’erreurs de segmentation, des fuites mémoires, des résultats pas satisfaisants, mais on a pu surmonter tout cela en analysant le code, en débuggant, en utilisant valgrind, des printf...etc.
Au final on obtient un résultat assez satisfaisant, qui peut toujours être amélioré, par exemple en utilisant des fonctions de tris afin d’optimiser la recherche d’un nœud dans un réseau, on s’est beaucoup concentré sur la complexité temporelle des fonctions de reconstitution, mais la complexité spatiale a été négligée, c’est un bon point qu’il faudra aborder afin d’optimiser au mieux le code et de trouver la meilleur compromis entre temps et mémoire.
La visualisation des résultats on la trouve plutôt pas mal, avec les fonctions SVG qui permettent de comparer nos résultat et de vérifier leur validité.

Il serait intéressant de diversifier les résultats en utilisant plusieurs instances afin que le projet soit le plus complet et exhaustif possible, et qu'il couvre tout les cas.

Nous vous remercions d'avoir pris le temps de lire ce rapport.