Antoine Lecomte et Zhang Yuxiang

Rapport sur l'amélioration du réseau de fibres optiques

Dans ce projet, nous visons à améliorer le réseau de fibres optiques d'une agglomération. Le projet se divise en deux parties principales : la reconstitution du plan du réseau et la réorganisation des attributions de fibres entre les opérateurs. La première partie du travail concerne la reconstruction du réseau à partir des tronçons de fibres optiques détenus par différents opérateurs, tandis que la seconde partie vise à optimiser l'utilisation des fibres en réaffectant les connexions entre les clients.

Dans le réseau, un câble contient un ensemble de fibres optiques et relie deux points du plan. Les points peuvent être des clients ou des concentrateurs, et chaque opérateur possède plusieurs chaînes de fibres optiques. Une chaîne relie toujours deux clients, et un point peut être à la fois un client et un concentrateur. Les instances des problèmes proviennent de bases de données bien connues contenant des réseaux géographiques et non géographiques adaptés pour ce projet.

Afin de nous y retrouver pendant l'avancée du projet, et pour compiler nos fichiers plus simplement, nous avons créé un Makefile, qui à lui seul peut permettre à un lecteur (extérieur au travail sur ce projet) de comprendre beaucoup plus facilement comment les fichiers fonctionnent entre eux et de manière générale pour comprendre rapidement ce dont chaque partie du code est responsable, et son utilité pour ce projet.

Le Makefile est utilisé pour compiler plusieurs programmes à partir de différents ensembles de fichiers source. Voici comment les fichiers fonctionnent ensemble :

1. Compilateur et options de compilation :
    - Le compilateur utilisé est GCC.
    - Les options de compilation sont définies dans CFLAGS, comprenant -Wall, -Wextra pour activer des avertissements utiles, et -std=c99 pour spécifier le standard C99.

2. Fichiers source :
    - Les fichiers sources sont répartis en plusieurs ensembles (SRCS1, SRCS2, SRCS3) correspondant à différents programmes ou parties de programmes.
    - Les fichiers source incluent des fichiers communs (Chaine.c, Reseau.c, SVGwriter.c, Hachage.c, ArbreQuat.c, temps_exec.c, Struct_File.c, Graphe.c) et des fichiers spécifiques à chaque programme.

3. Fichiers objets correspondants :
    - Pour chaque ensemble de fichiers source, il existe un ensemble correspondant de fichiers objets (OBJS1, OBJS2, OBJS3) générés à partir des fichiers source en remplaçant l'extension (.c) par (.o).

4. Nom de l'exécutable :
    - Les noms des exécutables à générer son spécifiés dans la variable EXEC.

5. Règles de compilation :
    - Chaque programme a sa propre règle de compilation. Par exemple, ChainMain, ReconstitueReseau, temps_exec.
    - Chaque règle spécifie les dépendances (fichiers objets) et la commande de compilation pour générer l'exécutable correspondant.
    - Les options de liaison (-lm) sont incluses pour lier avec la bibliothèque mathématique.

6. Règle de compilation pour chaque fichier objet :
    - Il y a une règle générique pour compiler chaque fichier source .c en un fichier objet (.o).

7. Nettoyage des fichiers :
    - La règle clean est utilisée pour supprimer les fichiers objets et les exécutables générés lors de la compilation.

Nous avons donc utilisé trois fonctions main sur l'ensemble du projet.

Réponses aux questions :

Q4.2 :

Pour x et y allant de 1 à 10, les clefs générées sont toutes différentes, ce qui assure l'unicité de chaque clef dans cet intervalle et cela suggère que la fonction de clé fonctionne bien pour générer des clés uniques à partir des coordonnées (x,y) des points. De plus, cette fonction de hachage nous permet d'éviter les collisions et donc un gain de temps précieux en termes de performances pour notre programme, devant en générer un nombre important.

Q6.1 & 6.4 :

En ayant réalisé 100000 itérations avec chaque méthode, nous remarquons que la méthode qui nécessite une liste chaînée semble avoir un temps d'exécution moyen plus rapide que les deux autres structures de données. L'arbre quaternaire a des temps d'exécutions moyens plus élevés que la liste chaînée, mais la différence est peu significative et cette méthode reste performante. La table de hachage a néanmoins des temps d'exécution plus élevés que les autres structures malgré avoir vérifié pour différentes tailles de tables.

En utilisant cette fois la méthode de génération aléatoire de points, nous constatons que les données d'entrée influent grandement sur les performances. En utilisant les données de la question 6.3, on obtient des résultats beaucoup plus élevés qu'auparavant. Pour une seule itération, la liste chaînée a un temps d'exécution légèrement plus long qu'une table de hachage très petite, l'arbre quaternaire n'est pas performant et les temps de parcours sont trop importants en comparaison. Et enfin, pour la

table de hachage, plus on augmente la taille de celle-ci, meilleures sont les performances. Cela est dû au fait que le risque de collisions est beaucoup plus faible que pour des tables plus petites. Cependant, en diminuant le nombre de points, les valeurs maximales de x et y ainsi que le nombre de chaînes dans le réseau, la différence de performances entre des tables de hachage plus ou moins grandes s'amoindrit, et l'arbre quaternaire offre des performances toutes aussi satisfaisantes que la liste chaînée.

En résumé, la table de hachage de très grande taille est la plus efficace pour reconstituer un réseau avec beaucoup de données, mais dans le cas où le réseau est petit, privilégier la liste chaînée ou l'arbre quaternaire pour obtenir les meilleures performances.

Q7.5 :

La contrainte gamma est respectée pour le réseau du fichier 00014_burma.cha, car aucune arête ne contient un nombre de liaisons supérieur ou égal à 3.

Pour améliorer la fonction reorganiseReseau, il peut être bon d'optimiser son efficacité en réduisant la complexité temporelle et spatiale de la fonction. Il est peut-être aussi possible de trouver des structures de données différentes que celles utilisées, qui pourraient améliorer son efficacité. Enfin, nous pensons que paralléliser les parties de l'algorithme pourrait nous aider à gagner en performances si nous exécutons la fonction sur une machine multicœurs (car dans ce cas plusieurs processus peuvent s'exécuter en même temps).