Yoan LE NEVEZ
Anh Tu NGUYEN
Groupe 10

PROJET TME STRUCTURE DE DONNÉES

Ce projet se porte sur la réorganisation du réseau de fibres optiques d'une agglomération pour améliorer
la distribution des services de connectivité.
Dans une première partie, nous devons faire la reconstitution du réseau.
Puis, dans une seconde partie, la réorganisation du réseau.

Ici, un réseau est un ensemble de câbles, où chacun des câbles contient un ensemble de fibres optiques
et ces câbles relient des clients et le couple de clients reliés par une chaîne est une commodité.
L'objectif de ce projet est de reconstruire numériquement le réseau existant et de réorganiser l'attribution
des fibres optiques entre les opérateurs pour optimiser l'utilisation des ressources et réduire les longueurs excessives des chaînes de fibres. Nous pourrons visualiser les réseaux à l'aide des fonctions de SVG. Nous créerons pour toutes nouvelles structures, des fonctions de désallocation de mémoire pour éviter toute fuite de mémoire. Tous les jeux de tests ont été vérifiés et ne contiennent aucune fuite de mémoire.

La structure utilisée pour stocker les chaînes et les points est la structure dans le fichier chaine.c. Elle est composée de Cellule Point ayant ses coordonnées et pointant vers la cellule suivante dans la liste. De même, on a une Cellule Chaîne qui contient un numéro, une liste des points de la chaîne et une autre chaîne dans la liste. Enfin, on a une structure Chaîne qui contient le nombre de gamma, le nombre de chaînes et une liste des chaînes. Nous avons à notre disposition des fichiers.cha contenant des informations sur un réseau et grâce à la fonction lectureChaine(FILE *f), nous pouvons lire le fichier et allouer, remplir, et retourner une instance de notre structure juste au dessus. Nous avons de même la fonction ecrireChaine(Chaines *C, FILE *f) pour, à partir d'une chaîne, écrire dans un fichier.cha les infos d'une chaîne. Nous avons testé leur bon fonctionnement dans le jeu de test ChaineMain.c.

Nous avons de même une structure pour représenter le réseau, qui sera par la suite stockée par différentes méthodes. Elle est composée d'une Cellule de Noeuds contenant un Noeud, qui lui-même est composé d'un numéro, et de ses coordonnées avec une liste de ses cellules noeuds voisins, et la cellule de noeud contient également une liste de cellule de noeud contenant donc une liste de noeud. Ensuite, on a la Cellule Commodite qui possède les 2 extrémités des commodités, et une liste de commodités ce qui permet de définir les points devant être reliés par une chaîne . Et enfin, on a le réseau qui est composé des infos sur le nombre de nœuds, le nombre de gamme, une liste de nœuds, et une liste de commodités. Nous aurons plusieurs fonctions très utile pour la suite de notre projet telles que : ecrireReseau(Reseau *R, FILE *f) qui nous permettra d'écrire un réseau à partir d'un fichier (.res), les fonctions nbLiaisons() et nbCommondites() et une fonction

affichageReseauSVG() pour créer visuellement le réseau. Nous avons créé, pour la clarté et la compréhension du code, différentes fonctions de créations tels que creer_CellNoeud(), creer_CellCommodite(), une fonction permettant de mettre à jour les voisins maj_voisins(). Ces fonctions nous seront utiles pour les 3 méthodes de stockage.

Pour revenir à notre objectif, qui est d'optimiser l'utilisation des ressources et réduire les longueurs excessives des chaînes de fibres, nous allons comparer différentes méthodes de stockage des chaînes. C'est-à-dire, différentes structures de données pour stocker un réseau telles que : une liste des chaînes, une table de hachage et un arbre quaternaire.

La première méthode que nous avons utilisée est une liste de chaînes. Nous allons donc parcourir chaque chaîne, puis rechercher les nœuds dans une liste chaînée pour reconstruire le réseau grâce à une chaîne. On verra par la suite que cette structure n'est pas forcément la plus optimisée.

La deuxième méthode que nous utilisons est une table de hachage avec gestion des collisions par chaînage. Pour se faire, on a créé une structure avec le nombre d'éléments, la taille max, et un tableau de pointeur vers une liste de nœuds. Pour utiliser la table de hachage, nous utilisons une fonction qui nous donne la clef d'un noeud par rapport à ses coordonnées avec fonctionClef(x,y) qui pour tout x et y allant de 1 à 10, il n'y a aucune collisions, les résultats de clefs vont de 0 à 220 sans doublon. Alors pour répondre à la question 4.1, oui elle semble appropriée. A partir de la clef venant de cette fonction, on peut utiliser une fonction de hachage qui est fonctionHachage() sur la clef. Pour reconstruire le réseau à partir d'une chaîne à l'aide d'une table de hachage, on parcourt les chaînes de point et on utilise la table de hachage pour rechercher ou créer un noeud, ce qui devrait être très rapide dû à l'utilité de la table de hachage. On espère avoir un temps plus optimisé grâce à une recherche d'un noeud instantané, ce qui est le but
de cette méthode.

Et enfin, la troisième méthode que nous utilisons est un arbre quaternaire. C'est un arbre où chaque
noeud possède quatre fils dans un espace 2D. Ici, chaque feuille sera un pointeur vers un nœud du réseau. Pour se faire, nous avons la structure contenant les coordonnées du centre de la cellule, la longueur et la hauteur de la cellule, un Noeud, et 4 sous arbre tel que sud-ouest pour x<xc et y< yc, sud-est pour x>=xc et y<yc, nord-ouest pour x<xc et y>=yc et enfin, nord-est pour x>=xc et y>=yc. Cette structure nous permet de déterminer rapidement si un nœud a déjà été stocké dans le réseau ou non tout comme la table de hachage avec la fonctionClef. On aura besoin de créer une fonction qui détermine les coordonnées minimales et maximales des points constituant les différentes chaînes du réseau. On a donc la fonction chaineCoordMinMax() qui va faire tous les points de la Chaine C et trouver les coordonnées existantes max et min pour x et y. De même, nous avons une fonction insererNoeudArbre qui nous permet d'insérer un nœud pour différents cas possibles tels que : si c'est un arbre vide, si c'est une feuille, ou bien si c'est une cellule interne. Pour reconstruire le réseau à partir d'une chaîne et à l'aide d'un arbre quaternaire, nous allons parcourir les chaînes de points et on va utiliser l'arbre tout en suivant les règles de partitionnement que l'insertion. On saura donc très rapidement s' il faut créer le nœud ou non.

On a donc déjà fait un jeu test dans le fichier ReconstitueReseau.c pour bien vérifier que chaque fonction marche correctement avant de pouvoir les comparer. Ce fichier contient une fonction main où nous donnons en paramètres : un fichier.cha et une option (1,2,3) tel que (liste,table,arbre) et retournes un fichier.res telle que testListe.res pour la liste ou testHachage.res pour la table ou encore testArbre.res pour l'arbre.

Nous avons de même un fichier main6.1.c qui va nous permettre de comparer la rapidité des 3 méthodes de stockage utilisant la même chaîne et on stockera le résultat des 3 méthodes dans temps_de_calcul.txt. On observe donc que le temps des 3 méthodes est relativement équivalent et ne nous rapporte pas plus d'informations. Or, du fait de la taille de la table de hachage, on remarque que plus la taille augmente, plus le temps mis par la fonction met du temps malgré que la différence est assez peu. Par exemple, pour une taille de 8, on aura entre 3 et 8 microsecondes pour les 3 méthodes, donc pour une taille de table de hachage faible et suffisante. Pour une table de hachage de taille 8000, le temps mit monte à 35 microsecondes ce qui reste faible mais on peut voir que la taille est en rapport avec le temps mit pour la table. Nous devons donc continuer d'essayer de comparer les 3 méthodes et trouver laquelle est la plus optimisée peu importe le nombre de chaînes présentes dans un réseau.

Nous allons donc créer une fonction generationAleatoire() dans le fichier Chaine.c qui va prendre un nombre de chaînes, de points de chaînes, et un xmax et ymax.
Cette fonction va nous permettre de tester nos 3 méthodes de stockage sur différents réseaux, et sur des réseaux bien plus longs.

Nous allons donc aller dans le fichier main.c, créer une chaîne avec un nombre de points de chaine de 100, un xmax et ymax de 5000 et faire varier le nombre de chaînes de 500 à 5000. Grâce à ce jeu de test, on va avoir une idée de quelle méthode est la plus optimisée pour notre problème pour tout type de taille. De même, pour la table de hachage, nous allons faire un cas à part où nous ne faisons pas varier le nombre de chaînes, mais seulement la taille de la table de hachage de 1000 à 10000. Le résultat de la comparaison entre les 3 méthodes est stocké dans le fichier graph.txt, et le résultat de la comparaison entre différentes tailles de table de hachage est stocké dans le fichier graph_hachage.txt. A À l'aide de ces données, nous avons pu créer des graphes pour mieux visualiser et comparer la rapidité de chaque méthode pour différents cas.

Nous avons donc 3 graphes : un pour la liste, un pour la table et l'arbre, et un pour la table de sa taille qui varie. Nous pouvons nous demander pourquoi avoir mit la liste dans un graphe seul et la table avec l'arbre, cependant en analysant les graphes nous pouvons remarquer la haute différence de temps entre
la liste et les 2 autres méthodes. Nous avons un temps > 5 secondes pour la liste tandis que pour la table de hachage et l'arbre, nous restons < 1 seconde.
Nous savons donc déjà que la liste chaînée n'est pas forcément la meilleure méthode de stockage pour notre problème actuel qui je le rappelle, est de reconstituer un réseau et de le réorganisé le plus efficacement possible pour qu'il soit optimisé, c'est pour cela que l'on compare nos 3 méthodes qui nous permettent de faire cela.
Maintenant analysons la table de hachage en fonction de sa taille qui varie. Nous remarquons que + la taille augmente, et + le temps diminue. Cela est le contraire de notre analyse précédente (main6.1.c). Cela pourrait s'expliquer par le nombre élevé de points et de chaînes qui, nécessite donc une assez grande taille de table pour éviter toutes collisions.

En ayant une grande table, nous utilisons plus de mémoire, mais nous évitons au maximum le risque de collisions quand il y a de nombreux points et chaînes, ce qui rend au final une table grande plus rapide.

Enfin, en comparant la table de hachage et l'arbre quaternaire, nous remarquons que l'arbre est bien inférieur à la table de hachage. La table de hachage varie entre [0.04;0.08] tandis que l'arbre varie entre [0.02;0.04]. Nous pouvons donc en conclure que l'arbre quaternaire est la méthode à prioriser, puis la table de hachage également. Or, la liste chaînée est à éviter dans notre cas.

Enfin, nous avons pour terminer notre projet, voulu optimiser le réseau en minimisant la somme totale des longueurs des chaînes. Pour ce faire, nous avons pour cette partie une structure qui représente un graphe contenant des arêtes, une liste d'arêtes, des sommets avec une liste de sommets, des commodités et le graphe contenant un tableau de pointeurs sur sommets, et un tableau des commodités. Nous utilisons de même un fichier nommé Struct_File contenant les fonctions estFileVide,Enfile,Defile.

Pour pouvoir optimiser notre réseau, nous allons donc créer un graphe et faire un parcours en largeur, et vérifier que Gamma est bien respecté et calculer le chemin le plus court pour chaque commodité.

Nous avons fait un jeu de test pour tester toutes les fonctions ainsi créé et fait un graphe, le graphe
stocké par le jeu de test est output_graph.svg et nous avons fait le jeu de test sur 2 instances, où les 2 fichiers résultats sont test_graphe_05000_USA-road-d-NY.res et test_graphe_07397_pla.res. Cependant, nous pourrions améliorer la fonction reorganiseReseau en réduisant la complexité de ce dernier, qui est de $0(n^2)$. De même, au lieu d'utiliser une matrice, on pourrait utiliser une table de hachage ou un arbre quaternaire.

Pour conclure ce projet, la structure la plus adaptée est l'arbre quaternaire, puis la table de hachage. La structure du graphe créée est intéressante pour optimiser un réseau et pourrait être encore améliorée pour minimiser la somme totale des longueurs des chaînes.

<u>COMMANDES POUR JEUX DE TESTS :</u>

ecrireChaines et lectureChaines :
gcc -Wall -o ChaineMain ChaineMain.c Chaine.c
-> Fichier résultat : testChaine.cha testChaine.html

reconstitueReseauListe/reconstitueReseauHachage/reconstitueReseauArbre :
gcc -Wall -o ReconstitueReseau ReconstitueReseau.c Hachage.c Reseau.c ArbreQuat.c Chaine.c SVGwriter.c -lm
-> Fichiers résultats : testHachage.res/.html, testListe.res/.html, testArbre.res/.html
Question 6.1 :
gcc -Wall -o main6.1 main6.1.c Chaine.c Reseau.c Hachage.c SVGwriter.c -lm
-> Fichier résultat : temps_de_calcul.txt

Comparaison temps + Graphe :

gcc -Wall -o main main.c ArbreQuat.c Reseau.c Hachage.c Chaine.c SVGwriter.c -lm
-> Fichiers résultats : graph.txt, graph_hachage.txt courbe_vitesse.ps

Optimisation du réseau partie :
gcc -Wall -o mainGraphe mainGraph.c Graphe.c Chaine.c Reseau.c SVGwriter.c Struct_File.c -lm
-> test_graph...res , output_graph.html