HASSANEIN Youssef
LIN Charles **Projet LU2IN006 - SDA** **2020-2021**

1. Reformulation du sujet courte

On considère un réseau de fibres optiques qui peut s'apparenter à un graphe simple non orienté. Chaque arête représente un cable qui relie deux points (ou noeuds) du réseau et qui contient une ou plusieurs fibres optiques.

Il existe deux types de points : les clients et les concentrateurs. Une chaine représente une chaine $u_1e_1u_2..u_ne_nu_{n+1}$ dans le graphe non orienté tel que les sommets u_1 et u_{n+1} sont des clients (ce sont les commodités de la chaine) et les points qui ne sont pas à l'extremité de cette chaine sont des concentrateurs. Un point peut être un client, un concentrateur ou les deux à la fois selon les chaines qu'il y a dans le réseau.

La première partie du projet consiste à reconstituer réseau en respectant le pseudo-code donné dans l'énoncé. On s'interesse a l'optimisation du test "$p \notin V$ ". Pour cela, on va etudier trois methodes qui correspondent a trois structures de donnees pour implementer l'ensemble V : une liste chaînee, une table de hachage et un arbre quaternaire.

La deuxième partie du travail consiste à réorganiser le réseau en utilisant un graphe non orienté représenté par des listes d'adjacence. On implémente notamment un parcours en largeur retournant le plus petit nombre d'arêtes et la chaine correspondante entre deux sommets u et v du graphe.

2. Description des structures manipulées, et la description globale de votre code : fichiers .h, .c, fonctions principales etc.

<u>Liste chainée</u>

Dans cette structure :

- Chaque nœud v sera repere par ses coordonnees, et on connaıtra la liste des pointeurs sur nœuds qui sont relies a v par un cable. Lors de la reconstitution du reseau, on attribuera a chaque nœud un numero entier unique qu'on lui choisira incrementalement.
- Chaque cable est donnee par des pointeurs sur ses deux nœuds extremites.
- Chaque commodite est une paire de pointeurs sur les nœuds du reseau qui doivent etre relies par une chaıne.

Ainsi, pour stocker les donnees du reseau, on utilisera la structure de donnees definie dans le fichier Reseau.h

Cette structure permet de stocker un Reseau comme une liste chaınee de Noeud et une liste chaınee de Commodite. Chaque Noeud v est donne par son numero, ses coordonnees et la liste chaınee des nœuds voisins, c'est-a-dire les nœuds qui sont li´es au noeud v par un cable. Une Commodite est simplement donnee par les deux nœuds qui seront a relier par une chaıne.

<u>Table de hachage</u>

On a une gestion des collisions par chaınage. La table de hachage va donc contenir un tableau de pointeurs vers une liste de nœuds. Lors du parcours de la liste des points constituant une chaıne, la table de hachage va nous permettre de d´eterminer rapidement si un nœud a deja ete stocke dans le reseau.

Arbre quaternaire

Comme pour la table de hachage, l'arbre quaternaire va nous permettre de d´eterminer rapidement si un nœud a déjà ete stocke dans le reseau.

Un arbre quaternaire est un arbre ou chaque nœud possede quatre fils. Dans un espace a deux dimensions, un arbre quaternaire represente une cellule rectangulaire. Son centre permet d'identifier les fils, qui representent les parties nord-ouest, nord-est, sud-est et sud-ouest de l'espace par rapport a ce centre.

Les nœuds de notre reseau peuvent etre stockes au niveau des feuilles de l'arbre quaternaire. En effet, on peut associer une donnee a chaque feuille de l'arbre, identifiee grace a ses coordonnees x et y. Dans le cadre de ce projet, la donnee associee a chaque feuille sera donc un pointeur vers un Noeud du reseau.

Pour pouvoir creer le nœud racine de l'arbre quaternaire qui contiendra tous les nœuds du reseau, il est necessaire d'identifier la longueur (coteX) et la hauteur (coteY) de la cellule. Pour cela, on peut utiliser les coordonnees minimales et maximales des points a stocker dans la structure.

Description globale

Les fichiers .c qui contiennent "Main" dans leur nom ainsi que ReconstitueReseau.c sont les fichiers permettant de tester nos différentes fonctions et de vérifier qu'il n'y a pas de fuite mémoire. Pour tester tous ces fichiers en une seule fois avec l'instance burma, il faut écrire dans le terminal make && make run. Les fonctions principales sont :

- pour la partie 1 : les fonctions rechercheCreeNoeudXXX et reconstitueReseauXXX où XXX représente la structure de données que l'on veut.
- pour la partie 2 : une fonction de parcours en largeur retournant le plus petit nombre d'arêtes et la chaine correspondante entre deux sommets u et v du graphe et une fonction reorganiseReseau :
    - qui cree le graphe correspondant au reseau,
    - qui calcule la plus courte chaıne pour chaque commodite,
    - qui retourne vrai si pour toute arete du graphe, le nombre de chaınes qui passe par cette arete est inferieur a γ, et faux sinon. On creera une matrice permettant de compter le nombre de chaınes passant par chaque arete {u, v}.

3. Description des jeux d'essais utilisés pour valider votre code ou évaluer les performances de vos programmes

ChaineMain.c permet de tester les fonctions de lecture et d'écriture.

ReconstitueReseau.c utilise la ligne de commande pour prendre un fichier .cha en paramètre et un nombre entier indiquant quelle méthode l'on désire utiliser (liste chainée, table de hachage ou arbre quaternaire) pour reconstituer le réseau.

HachageMain.c permet de tester la fonction clef f (x, y) = y + (x + y)(x + y + 1)/2.

ArbreQuatMain.c permet de tester les fonctions d'insertion. On insère 6 nœuds situés à des coordonnées précises et on regarde si les insertions se sont faites au bon endroit.

GrapheMain.c permet de tester la fonction de parcours en largeur et la fonction reorganiseReseau décrite ci-dessus.

4. Réponses aux questions et analyse commentée (et statistique) des performances des programmes (analyse des résultats expérimentaux)

Q4.2 :
Cette fonction clef semble appropriée ! Les clefs générées sont différentes pour x et y entiers allant de 1 à 10.
La fonction f est en fait une bijection de N^2 -> N (démonstration simple, voir maths discrètes)

Les instances 07397_pla.res et 05000_USA-road-d-NY.res ne contiennent que des entiers !! Et même pour les réels (burma.res), on a jamais la même clef générée.

Remarque: $R+^2$ n'est pas dénombrable et il n'existe pas de bijection de $R+^2$ -> N. Si on a des points très très proches, alors on peut avoir la même clef générée mais ici on considère des réseaux de fibres optiques (les clients/concentrateurs sont suffisamment éloignés les uns des autres pour ne pas avoir la même clef générée)

Exercice 6 (voir ex6.c, ex6_1.txt, ex6_1.pdf, ex6_3.txt, ex6_3.pdf)
ex6_1.pdf -> temps de calcul reconstitueReseau burma pour chaque structure de données en fonction de la taille de la table de hachage
ex6_3.pdf -> temps de calcul reconstitueReseau chaines de points aléatoires pour chaque structure de données en fonction du nombre de points total des chaines

1. La fonction reconstitueReseauArbreQuat est la plus lente des trois.
Celle modélisée avec une table de hachage devient de plus en plus rapide en fonction de la taille M de la table de hachage et la vitesse de l'algorithme devient constante lorsque M est supérieure au nombre total de points (ou nœuds) (12 nœuds avec 00014_burma.res). Plus le load factor alpha (nbPointsTotal/M) est petit, moins on a de collisions et plus l'algorithme est rapide et lorsque alpha <= 1, la vitesse de l'algorithme devient constante. Lorsque M = 1, on a la même vitesse que l'algorithme avec une liste chainée (10 microsecondes)

4. Le temps de calcul pour la fonction reconstitueReseauListe en fonction du nombre de chaines suit une allure polynomiale et est beaucoup plus lente que pour les deux autres structures.
Ensuite, reconsitueReseauHachage est meilleure que celle de l'arbre quat et plus on diminue le load factor (alpha allant de 10 à 5/8), plus l'algorithme est rapide.

/* ---------- */

Soit n le nombre de nœuds (ou points) et M la taille de la table de hachage
Soit le load factor alpha = n/M

/* Analyse complexité pire-cas recherche */
On a une complexité moyenne en O(1+alpha) et pire-cas en O(n) pour la fonction rechercheCreeNoeudHachage.
On a une complexité pire-cas en O(n) pour la fonction rechercheCreeNoeudListe (point pas présent ou en dernière position de la liste).

On a une complexité pire-cas en O(h) pour la fonction rechercheCreeNoeudArbreQuat avec h la hauteur de l'arbre quat et dans le pire des cas h peut être très grand: si on a 2 points très proches (au sud-ouest par exemple), on aura alors beaucoup de cellules internes à parcourir avant d'atteindre la feuille que l'on cherche.
Remarque: on ne peut pas avoir 2 points de mêmes coordonnées dans l'arbre quat
Dans le cas d'un arbre quaternaire parfait (points (ou nœuds) du réseau uniformément repartis), on a 4^h <= nbNoeudsArbreQuat < 4^(h+1) et donc h = log4(nbNoeudsArbreQuat) en partie entière

/* Analyse complexité pire-cas reconstitue */

On en déduit que :
LC -> O(n²)
TH -> O(n*(1+alpha))
AQ -> O(n*h) h: hauteur de AQ

/* Unweighted Graph Breadth First Search */

Complexité pire-cas en O(|V|+|E|). Dans le pire des cas (la plus courte chaine reliant src à dest est de longueur maximale, càd on défile le sommet dest en dernier), on doit enfiler/défiler (theta(1)) chacun des sommets du graphe et pour chacun de ces sommets, on doit parcourir les voisins de ce sommet. Donc on a O(|V|+ $\sum_{v \in V} d(v)$) = O(|V|+2|E|) = O(|V|+|E|)

Q7.5 :

On obtient 0 pour les instances burma, usa et pla. Cela veut dire que si on prend la plus courte chaine pour chaque commodité, le nombre maximal de fibres optiques (gamma) dans un câble n'est plus respecté ...
Pour améliorer la fonction, au lieu d'utiliser une matrice qui prend beaucoup d'espace mémoire (nbSommet²/2 * sizeof(int)) inutilement, on peut ajouter un champ de type int pour chaque arête. En effet, la densité des graphes pour chacune des instances étant très faible (densité = nbLiaisons/(nbNoeuds*(nbNoeuds-1)/2)), on obtient des matrices creuses.