Rapport de projet

LU2IN006 - Structure de Données
(TME 5 - TME 10)

ZHOU Jérémy 21126962
CHU Amélie 21206329

Année scolaire 2023-2024
Chargés: groupe 1 - Luc BLASSEL, Robin KOUBA

Table des matières

Figures du rapport:

1. Présentation du projet

Le but de ce projet est de réorganiser un réseau de fibres optiques d'une agglomération afin de l'améliorer.

Pour cela, on modélise le réseau par un ensemble de câbles qui contiennent chacun un ensemble de fibres optiques (au minimum une fibre optique par câble) reliant des clients.

Tout d'abord, on cherche à reconstituer le réseau.
Cependant, il n'existe pas de plan complet du réseau, mais pour pouvoir reconstituer le réseau, on sait que :

- Chaque opérateur possède quelques fibres optiques du réseau
- Chaque opérateur connaît les tronçons de fibres optiques qu'il utilise dans le réseau

Ensuite, on va chercher à optimiser le réseau. Il faut alors ré-attribuer les fibres de chaque opérateur car il y a des câbles qui sont sous-exploités et d'autres sur-exploités.

Tout au long de ce projet, nous allons pouvoir comparer les performances de 3 structures: la liste chaînée, la table de hachage et l'arbre quaternaire.

2. Description des fichiers

Le projet comprend 2 dossiers principaux et les fichiers suivants:

[Dossier *Ressources*]

- Les fichiers des structures et leur header associé:

ArbreQuat.c	ArbreQuat.h
Chaine.c	Chaine.h
File.c	File.h
graphe.c	graphe.h
Hachage.c	Hachage.h
Reseau.c	Reseau.h
SVGwriter.c	SVGwriter.h

- Les fichiers main utilisant les structures:

ReconstitueReseau.c	(Exercice 2 Q3)
comparaison_structure.c	(Exerccie 6 Q1)
comparaison_structure_graphe.c	(Exercice 6 Q3)
main_reorganiseReseau.c	(Exercice 7 Q5)

- Fichiers complémentaires:

[Dossier *instances*] : contient plusieurs instances .cha

EXO4_Q2.c	(Exercice 4 Q2)
00014_burma.res	00014_burma.cha
Makefile	

- Les fichiers tests:

test_ABQ.c
test_Chaine.c
test_File.c
test_graphe.c
test_Hachage.c
test_Reseau.c

[Dossier *Courbes_Resultats*]

- Les fichiers résultats:

- EXO6_Q1.txt
- EXO6_Q3_Abr_Hash.txt EXO6_Q3_Lc.txt EXO6_Q3_Abr.txt
- EXO7_Q5.txt
- courbe_abr_hash.png courbe_lc.png courbe_abr_hsh.png
- commande_gnu_abr_hash.txt
- commande_gnu_lc.txt
- commande_gnu_abr.txt

3. Description des structures

Le code contient plusieurs structures:

Liste chaînée

Description: Un ensemble de chaînes de type **Chaines**, constitué d'une liste chaînée de chaînes **CellChaine**, elle-même constituée de points de type **CellPoint**.

Structure dans le fichier **Chaines.h**:

```c
#ifndef __CHAINE_H__
#define __CHAINE_H__

#include <stdio.h>


/* Liste chainee de points */
typedef struct cellPoint{
  double x, y;                  /* Coordonnees du point */
  struct cellPoint *suiv;       /* Cellule suivante dans la liste */
} CellPoint;

/* Cellule d'une liste (chainee) de chaines */
typedef struct cellChaine{
  int numero;                   /* Numero de la chaine */
  CellPoint *points;            /* Liste des points de la chaine */
  struct cellChaine *suiv;      /* Cellule suivante dans la liste */
} CellChaine;

/* L'ensemble des chaines */
typedef struct {
  int gamma;                    /* Nombre maximal de fibres par cable */
  int nbChaines;                /* Nombre de chaines */
  CellChaine *chaines;          /* La liste chainee des chaines */
} Chaines;


Chaines* lectureChaines(FILE* f);                      // Exercice 1 Question 1
void ecrireChaines(Chaines* C, FILE* f);               // Exercice 1 Question 2
void afficheChainesSVG(Chaines* C, char* nomInstance);  // Exercice 1 Question 3
double longueurChaine(CellChaine* c);                  // Exercice 1 Question 4
double longueurTotale(Chaines* C);                     // Calcule et renvoie la longueur totale de toutes les
chaines de C
int comptePoints(CellChaine* c);                       // Exercice 1 Question 5
int comptePointsTotal(Chaines* C);                     // Calcule et renvoie le nombre de points totale dans C

Chaines* generationAleatoire(int nbChaines, int nbPointsChaine, int xmax, int ymax);  // Exercice 6 Question 2

//Fonctions de base
Chaines* creer_chaine();                            //Creer une instance de chaine et la renvoie
double distance(CellPoint* a, CellPoint* b);        //Calcule et renvoie la distance entre les points a et b

//Fonctions de desallocation
void liberer_chaine(Chaines* C);                    // Libere une Chaines C
void liberer_cellchaine(CellChaine* c);             // Libere une liste de CellChaine C
void liberer_cellpoint(CellPoint* p);               // Libere un CellPoint p

#endif
```

Table de hachage

Description: Une table de hachage dont chaque case contient une liste de **CellNoeud**.
Structure dans le fichier **Hachage.h**:

```c
#ifndef HACHAGE_H
#define HACHAGE_H

#include "Reseau.h"


typedef struct{
  int nbElement; //pas necessaire ici
  int tailleMax;
  CellNoeud** T;
} TableHachage ;


int f(int x, int y);                                                          // Exercice 4 Question 2
int h(TableHachage* t, int k);                                                // Exercice 4 Question 3
Noeud* rechercheCreeNoeudHachage(Reseau* R, TableHachage* H, double x, double y);   // Exercice 4 Question 4
Reseau* reconstitueReseauHachage(Chaines* C, int M);                          // Exercice 4 Question 5

//Fonctions de base
TableHachage* initTableHachage(int taille);        // Initialise et renvoie une table de hachage

//Fonction de desallocation
void liberer_tablehachage(TableHachage* t);        // Libere la table de hachage

#endif
```

Arbre quaternaire

Description: Un arbre quaternaire avec 4 fils représentant leur position par rapport au père:
- nord-ouest (x entre 0 et x_pere, et y entre y_pere et y_max)
- nord-est (x entre x_pere et x_max, et y entre y_pere et y_max)
- sud-ouest (x entre 0 et x_pere, et y entre 0 et y_pere)
- sud-est (x entre x_pere et x_max, et y entre 0 et y_pere

Structure dans le fichier **ArbreQuat.h**:

```c
#ifndef __ARBRE_QUAT_H__
#define __ARBRE_QUAT_H__

#include "Reseau.h"


/* Arbre quaternaire contenant les noeuds du reseau */
typedef struct arbreQuat{
    double xc, yc;          /* Coordonnees du centre de la cellule */
    double coteX;           /* Longueur de la cellule */
    double coteY;           /* Hauteur de la cellule */
    Noeud* noeud;           /* Pointeur vers le noeud du reseau */
    struct arbreQuat* so;   /* Sous-arbre sud-ouest, pour x < xc et y < yc */
    struct arbreQuat* se;   /* Sous-arbre sud-est, pour x >= xc et y < yc */
    struct arbreQuat* no;   /* Sous-arbre nord-ouest, pour x < xc et y >= yc */
    struct arbreQuat* ne;   /* Sous-arbre nord-est, pour x >= xc et y >= yc */
} ArbreQuat;


void chaineCoordMinMax(Chaines* C, double* xmin, double* ymin, double* xmax, double* ymax);        // Exercice 5
Question 1
ArbreQuat* creerArbreQuat(double xc, double yc, double coteX, double coteY);                       // Exercice 5
Question 2
void insererNoeudArbre(Noeud* n, ArbreQuat** a, ArbreQuat* parent);                                 // Exercice 5
Question 3
Noeud* rechercheCreeNoeudArbre(Reseau* R, ArbreQuat** a, ArbreQuat* parent, double x, double y);    // Exercice 5
Question 4
Reseau* reconstitueReseauArbre(Chaines* C);                                                         // Exercice 5
Question 5

//Fonctions de desallocation
void liberer_arbre(ArbreQuat* a);          // Libere l'ArbreQuat a

#endif
```

Réseau

Description : représente un réseau qui comprend un ensemble de **Noeud** et leurs voisins dans des **CellNoeud**, et un ensemble de **CellCommodite** qui représente deux extremités d'une chaîne du réseau.

Structure dans le fichier **Reseau.h**:

```c
#ifndef __RESEAU_H__
#define __RESEAU_H__
#include "Chaine.h"
#include <stdio.h>

typedef struct noeud Noeud;

/* Liste chainee de noeuds (pour la liste des noeuds du reseau ET les listes des voisins de chaque noeud) */
typedef struct cellnoeud {
    Noeud *nd;                      /* Pointeur vers le noeud stocké */
    struct cellnoeud *suiv;         /* Cellule suivante dans la liste */
} CellNoeud;

/* Noeud du reseau */
struct noeud{
   int num;                        /* Numero du noeud */
   double x, y;                    /* Coordonnees du noeud*/
   CellNoeud *voisins;             /* Liste des voisins du noeud */
}; //Noeud

/* Liste chainee de commodites */
typedef struct cellCommodite {
    Noeud *extrA, *extrB;           /* Noeuds aux extremites de la commodite */
    struct cellCommodite *suiv;     /* Cellule suivante dans la liste */
} CellCommodite;

/* Un reseau */
typedef struct {
    int nbNoeuds;                   /* Nombre de noeuds du reseau */
    int gamma;                      /* Nombre maximal de fibres par cable */
    CellNoeud *noeuds;              /* Liste des noeuds du reseau */
    CellCommodite *commodites;      /* Liste des commodites a relier */
} Reseau;

Noeud* rechercheCreeNoeudListe(Reseau *R, double x, double y);
Reseau* reconstitueReseauListe(Chaines *C);
void ecrireReseau(Reseau *R, FILE *f);
int nbLiaisons(Reseau *R);
int nbCommodites(Reseau *R);
void afficheReseauSVG(Reseau *R, char* nomInstance);

//fonctions implementees par nous meme
Reseau* creer_reseau();
CellCommodite* creer_cellcommodite();
CellNoeud* creer_cellnoeud(Noeud* noeud);
Noeud* creer_noeud(double x, double y, int num);
void ajouter_voisin(Noeud* n, Noeud* voisin);
int nb_commodite(Reseau* r);

//Desallocation
void liberer_reseau(Reseau* r);
void liberer_cellnoeud(CellNoeud * c);
void liberer_commodite(CellCommodite* c);
void liberer_voisins(Noeud* n);
void liberer_Noeud(Noeud* n);

#endif
```

Graphe

Description: représente sous forme de graphe un réseau. Similaire à la structure réseau, le graphe va inclure des arêtes dans son implémentation.

Structure dans le fichier **graphe.h**:

```
#ifndef __GRAPHE_H__
#define __GRAPHE_H__
#include <stdlib.h>
#include <stdio.h>
#include "Reseau.h"

typedef struct {
int u ,v ; /* Numeros des sommets extremite */
} Arete ;


typedef struct cellule_arete {
    Arete * a ;         /* pointeur sur l'arete */
    struct cellule_arete * suiv ;
} Cellule_arete ;

typedef struct {
    int num ;           /* Numero du sommet (le meme que dans T_som) */
    double x , y ;
    Cellule_arete * L_voisin ; /* Liste chainee des voisins */
} Sommet ;

typedef struct {
    int e1 , e2 ;       /* Les deux extremites de la commodite */
} Commod ;

typedef struct {
    int nbsom ;         /* Nombre de sommets */
    Sommet ** T_som ;   /* Tableau de pointeurs sur sommets */
    int gamma ;
    int nbcommod ;      /* Nombre de commodites */
    Commod * T_commod ; /* Tableau des commodites */
} Graphe ;


//fonctions demandees

Graphe* creerGraphe(Reseau* r);                                     // Exercice 1 Question 1
int plus_petit_nb_aretes(Graphe* g, Sommet* u, Sommet* v);          // Exercice 1 Question 2
int* plus_petit_nb_aretes_liste(Graphe* g, Sommet* u, Sommet* v);   // Exercice 1 Question 3
int reorganiseReseau(Reseau* r);                                    // Exercice 1 Question 4

//fonctions implementees par nous meme

Graphe* initGrapheVide(int nbsom, int gamma, int nbcommod);         // Initialise et renvoie un graphe
Commod creerCommod(int e1, int e2);                                 // Initialise et renvoie une commodite
(structure)
Sommet* creerSommet(int num, double x, double y);                   // Initialise et renvoie un sommet
Arete* creerArete(int u, int v);                                    // Initialise et renvoie une arete
Cellule_arete* creerCellule_arete(Arete * a);                       // Initialise et renvoie une cellule arete

//Desallocation

void liberer_Graphe(Graphe* graphe);                                // Libere un graphe
void liberer_Commod(Commod* commod);                                // Libere une commodite
void liberer_Sommet(Graphe* graphe, Sommet* sommet);                // Libere un sommet
# endif
```

4. Ajouts hebdomadaires et réponses aux questions

TME 5 : Exercice 1 - 2

Les fichiers associés sont:
Chaine.c Chaine.h Makefile Reseau.h Reseau.c ReconstitueReseau.c test_Chaine.c

TME 6 : Exercice 3

Les fichiers associés sont:
SVGwriter.c SVGwriter.h Reseau.h Reseau.c test_Reseau.c

TME 7 : Exercice 4

Les fichiers associés sont:
Hachage.h Hachage.c test_Hachage.c EXO4_Q2.c

Question 2:

La fonction clef **f(x,y)** semble appropriée car il y a peu de collisions dans les nombres générés aléatoirement (code dans **EXO4_Q2.c**). On peut donc l'utiliser pour générer les clefs des points (x, y).

TME 8 : Exercice 5

Les fichiers associés sont:
ArbreQuat.h ArbreQuat.c test_ABQ.c

TME 9 : Exercice 6

Les fichiers associés sont:
comparaison_structure.c : main pour la question 1
EXO6_Q1.txt : résultats du fichier comparaison_structure.c
comparaison_structure_graphe.c : main pour la question 3
EXO6_Q3_Abr_Hash.txt : résultats pour la liste chaînée de la question 3
EXO6_Q3_Lc.txt : résultats pour la table de hachage et l'arbre de la question 3
EXO6_Q3_Abr.txt : résultats poussées pour l'arbre de la question 3

<u>Question 1</u> : Depuis le fichier **comparaison_structure.c** nous allons mesurer le temps de calcul pour les trois structures sur différentes instances passées en paramètre. Les instances sont identifiées par leur nombre de chaînes nbChaines. La taille de la table de hachage varie entre 100 et 500 avec un pas de 100. Les résultats sont dans le fichier **EXO6_Q1.txt.**

nom	nbChaine	ABQ	LC	H100	H200	H300	H400	H500
0014_burma.cha	8	0.026275	0.005025	0.000372	0.000384	0.000431	0.000446	0.000428
00100_USA-road-d-NY-A-100.cha	15	0.026391	0.005922	0.001447	0.001484	0.001290	0.001392	0.001830
00048_att.cha	20	0.026617	0.006716	0.001410	0.001676	0.001536	0.001353	0.001432
00052_berlin.cha	25	0.027464	0.013486	0.001775	0.001556	0.001808	0.001905	0.001722
00076_eil.cha	30	0.019794	0.008567	0.001543	0.001513	0.001534	0.001460	0.001546
00144_pr.cha	40	0.028016	0.008979	0.002281	0.003027	0.002431	0.002925	0.002824
00150_ch.cha	50	0.032408	0.011522	0.004254	0.003792	0.005196	0.004710	0.010345
00417_fl.cha	120	0.045166	0.087403	0.016025	0.016681	0.016947	0.015919	0.015428
00493_d.cha	200	0.097658	0.322905	0.040854	0.053750	0.048780	0.043628	0.041049
00783_rat.cha	300	0.133395	1.188147	0.075645	0.063494	0.081392	0.111048	0.075261
01400_fl.cha	500	0.356860	2.917192	0.079649	0.128880	0.126538	0.123026	0.120557
02392_pr.cha	700	0.500188	8.221369	0.242484	0.203170	0.194920	0.188399	0.184951
03795_fl.cha	1000	1.164072	24.201309	0.669900	0.538205	0.501499	0.484462	0.476594
04461_fnl.cha	1400	1.714694	61.527702	0.851261	0.671525	0.613333	0.582865	0.560349
05000_USA-road-d-NY.cha	1700	33.311728	586.182016	22.716529	19.826696	19.025261	18.443591	18.193149
08000_USA-road-d-NY.cha	3300	27.077141	6163.957253	35.208126	17.886318	12.491195	10.384933	9.056933
09000_USA-road-d-NY.cha	4000	42.141464	15847.057837	142.009350	70.549039	44.038985	35.052714	23.743088

Figure 1 - Temps de calcul (en sec) des structures pour la reconstruction réseau sur des instances

Légende des figures 1-4:
Hx = table de hachage de taille x
ABQ = ABR = arbre quaternaire
LC = liste chaînée

On constate d'après la figure 1 que le temps de calcul augmente à mesure que le nombre de chaînes augmente pour les 3 structures. La **liste chaînée** est la structure la plus lente pour un grand nombre de chaines (à partir de 300 nbChaine). Cela s'explique par le fait que dans la fonction **reconstitueReseauListe(),** la recherche d'un nœud dans une liste est linéaire avec une complexité pire cas **O(n)** avec **n** le nombre d'éléments dans la liste.

Pour la **table de hachage**, en supposant une répartition uniforme des éléments , cette recherche est en **O(n/m) = O(α)** avec α le facteur de charge[1]. Plus ce facteur est faible, plus la recherche dans la table de hachage est rapide.
Pour l'**arbre quaternaire**, la recherche se fait en **O($\log_4$(n))** avec n le nombre d'éléments dans l'arbre.

Entre l'arbre et la table de hachage, si le facteur de charge est suffisamment faible, alors la recherche dans la table de hachage est plus rapide que celle de l'arbre. On remarque que pour 1400 chaînes, l'arbre est plus lent que les tables de hachage, mais à 4000 chaînes, seules les tables de hachages de tailles 400 et 500 sont plus rapides que l'arbre.
On peut en déduire que sur de plus grand nombre, l'arbre quaternaire est la structure la plus rapide des 3 structures.

Question 3 : En lançant le fichier **comparaison_structure_graphe.c**, nous allons mesurer le temps de calcul des 3 structures (avec reconstitueReseauListe(), reconstitueReseauArbre() et reconstitueReseauHachage()) pour des chaînes générées aléatoirement, de tailles nbChaines allant de 500 à 5000 avec un pas de 500. Pour la table de hachage, nous faisons la mesure pour 4 tailles: 50, 100, 500 et 1000.

Les résultats sont sauvegardés dans les fichiers **EXO6_Q3_Abr_Hash.txt** et **EXO6_Q3_Lc.txt.** Suite à cela, nous traçons les courbes liées avec en abscisse le nombre de points et en ordonnée le temps de calcul en secondes.

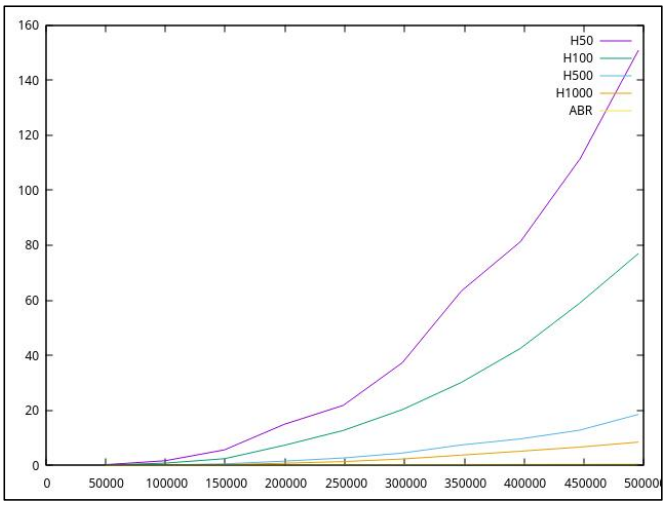

Figure 2 - ReconstitutionReseauArbre/Hachage (en sec) en fonction du nombre de points

[1] Le facteur de charge α se calcul en α=n/m avec n le nombre de points total et m la taille de la table de hachage, il correspond au nombre moyen de points par case

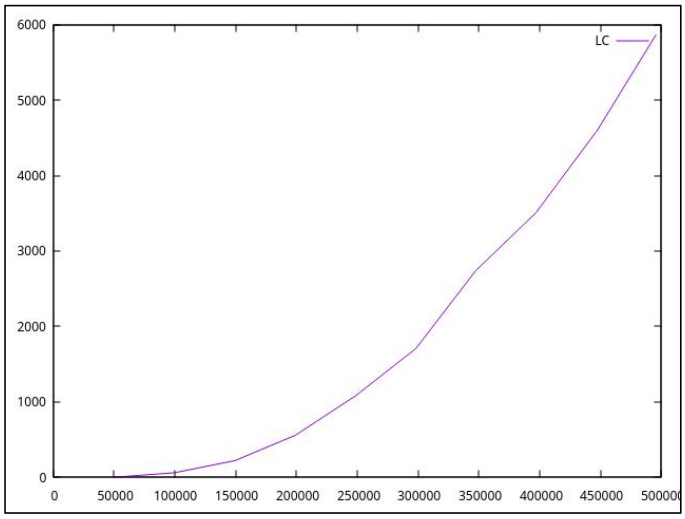

Figure 3 - ReconstitutionReseauListe (en sec) en fonction du nombre de points

Question 4:

On constate comme prévu (fig2 et fig3) que pour la liste chaînée la courbe a une allure quadratique, ce qui correspond à **$O(n^2)$,** la complexité théorique de reconstitueReseauListe(). Elle est la moins performante des trois structures.
La table de hachage a elle aussi une courbe à l'allure quadratique, mais qui est de plus en plus aplatie à mesure que la taille de la table augmente. Cela correspond bien à **$O(n^2/m)$** la complexité théorique de reconstitueReseauHachage() dont la valeur de n^2 diminue lorsque m tend vers l'infini.
Enfin, la courbe de l'arbre quaternaire croît de façon très lente et est bien plus rapide que nos tables de hachage. Afin de mieux percevoir l'allure de la courbe ABQ, nous avons réalisé d'autres mesures sauvegardées dans le fichier **EXO6_Q3_Abr.txt.**

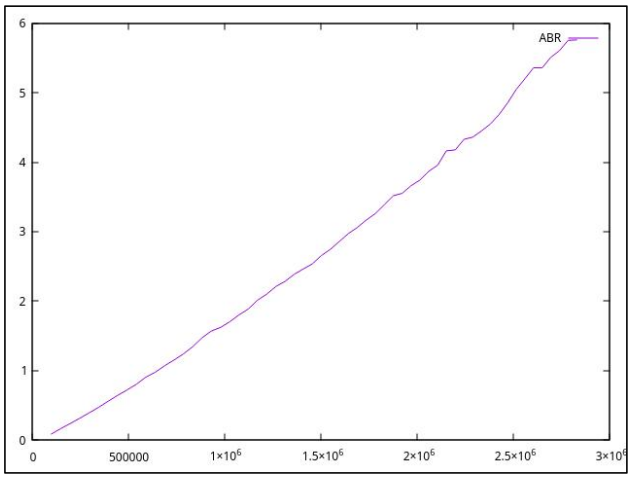

Figure 4 - ReconstitutionReseauArbre (en sec) en fonction du nombre de points

On constate dans la figure 4 une allure assez linéaire mais aussi similaire à **O(n.log(n))** qui est la complexité théorique de reconstitueReseauArbre().
Afin de pouvoir comparer la complexité de la fonction reconstitueReseau() pour la structure de l'arbre et de la table de hachage, on a effectué un calcul de limite:

$$\lim_{n\to\infty} \frac{n \cdot \log(n)}{\frac{n^2}{m}} = \lim_{n\to\infty} \frac{\log(n)}{\frac{n}{m}} = \lim_{n\to\infty} m \cdot \frac{\log(n)}{n} \to 0$$

On remarque que lorsque n tend vers l'infini et pour m la taille de la table fixé, la complexité l'arbre quaternaire est inférieure à celle de la table de hachage, la structure de l'arbre est donc meilleure en terme de recherche.

TME 10 : Exercice 7

Les fichiers associés sont:
File.c File.h graphe.h graphe.c test_graphe.c EXO7_Q5.txt
main_reorganiseReseau.c : main pour la question 5

Question 3 : Pour stocker l'arborescence des chemins, nous allons utiliser un tableau d'entiers qui va stocker le prédécesseur d'un sommet, puis nous allons sauvegarder la chaîne entre deux sommets u et v dans un tableau d'entiers (int*) t dont la taille équivaut au nombre total de nœuds dans le réseau. Ainsi t = [u, s1, s2, …, v, 0, ..., 0] avec des 0 dans les cases vides. En remontant de pères en pères depuis extrémité v, on va construire le tableau t en faisant correspondre t[i] au sommet de distance i par rapport à u.
Par exemple, la chaîne 1-2-3 va être stockée comme suit [1,2,3,0,...0] avec d[1] = 0, d[2] = 1, d[3] = 2 dans le tableau de distance.

Afin de simplifier le code, nous somme parti sur une allocation **statique** de t de taille fixe. Une alternative serait de faire une allocation de taille nombre total de nœuds puis de ré-allouer de façon à ne pas conserver les cases vides.

Question 5 : Avec le fichier **main_reorganiseReseau.c**, nous testons plusieurs instances de différentes tailles avec la fonction **reorganiseReseau(),** qui renvoie 0 s'il existe un câble sur-exploité (supérieur à gamma) sinon 1. On enregistre ces résultats dans **EXO7_Q5.txt.**

Nom_fichier	nbChaines	gamma	temps	t
00014_burma.cha	8	3	0.000012	0
00100_USA-road-d-NY-1-100.cha	15	6	0.000058	1
00048_att.cha	20	7	0.000056	1
00052_berlin.cha	25	11	0.000063	1
00076_eil.cha	30	8	0.000094	1
00144_pr.cha	40	10	0.000239	1
00150_ch.cha	50	8	0.000365	1
00417_fl.cha	120	22	0.001983	1
00493_d.cha	200	36	0.004386	1
00783_rat.cha	300	39	0.010195	1
01400_fl.cha	500	99	0.030151	1
02392_pr.cha	700	119	0.064348	1
03795_fl.cha	1000	161	0.167159	1
04461_fnl.cha	1400	374	0.251289	1
05000_USA-road-d-NY.cha	1700	143	0.369545	0
08000_USA-road-d-NY.cha	3300	499	1.209327	1
09000_USA-road-d-NY.cha	4000	617	1.670757	1
10000_USA-road-d-NY.cha	4700	1050	1.688889	1

Figure 5 - Résultats et temps de calcul (en sec) de **reorganiseReseau()** en fonction des instances

Pour les instances **00014_burma.cha** et **05000_USA-road-d-NY.cha**, la fonction reorganiseReseau() renvoie 0, c'est-à-dire que les réseaux sont saturés, et qu'il existe un câble dont le nombre de fibre est supérieur à gamma.

Une amélioration de la fonction à envisager est de pouvoir optimiser le réseau. En suivant ce raisonnement, la chaîne construite entre deux extrémités n'est pas la chaîne la plus courte, le but étant de mieux répartir les fibres. Le raisonnement est similaire au parcours en largeur, sauf que le prochain sommet choisi parmi les voisins non visités est celui dont le nombre de fibres est le plus petit.

L'algorithme de réorganisation:

M = matrice sommet-sommet
sommet_prec = numéro du sommet **extrA**
Enfiler **extrA** dans la file

Tant que la file n'est pas vide:
| **n** = defiler file
| Pour tous les voisins **vi** du nœud **n**:
(Réitérer jusqu'à ce que tous les **vi** soient enfilés)
| Enfiler dans la file le voisin **vj** ∉ file tel que M[**u**][**vj**] <= M[**u**][**vi**]
| Mettre à jour le tableau de distance
| Mettre à jour le tableau de pères
| Mettre à jour la matrice M[sommet_prec][**n**->num]
| sommet_prec = n->num

Avec cet algorithme, on s'assure de répartir au mieux les fibres parmi les câbles.

5. Tests

Tout au long du projet, les structures implémentées et leurs fonctions ont été testées dans un fichier main associé avec des assertions:

test_ABQ.c :	test de la structure Arbre quaternaire
test_Chaine.c :	test de la structure Chaines
test_File.c :	test de la structure File
test_graphe.c :	test de la structure graphe
test_Hachage.c :	test de la structure Table de Hachage
test_Reseau :	test de la structure Reseau

Chaque fichier test a été lancé avec **valgrind** pour vérifier qu'il n'engendre aucun problème lié à la mémoire de l'ordinateur ou à la syntaxe du code. Ils sont de plus testés avec la librairie **<assert.h>.**

Pour compiler les fichiers test, il faut lancer la commande du Makefile:
$ make test

Pour les courbes, il est possible de les afficher avec:
$ gnuplot -p < commande_gnu_lc.txt
*$ gnuplot -p < commande_gnu_abr_hash.tx*t
*$ gnuplot -p < commande_gnu_abr.tx*t

6. Comparaison des structures

Suite à ce projet qui met en œuvre trois structures de données, on peut constater que pour une recherche ayant peu d'éléments, la liste chaînée est la plus adaptée, d'autant plus qu'en terme d'espace mémoire elle moins coûteuse que les deux autres structures.

Toutefois lorsque le nombre d'éléments devient très conséquent, la table de hachage et l'arbre quaternaire sont à privilégier. Afin d'avoir une table de hachage optimal, il faut générer une table de hachage dont la taille réduit grandement le facteur de charge, et ce calcul est possible en divisant le nombre de points total sur le facteur de charge.

En comparant les complexités de la table de hachage et de l'arbre quaternaire, on constate que la complexité de la table de hachage est plus grande. De plus pour un très grand nombre d'éléments, la table de hachage va être de plus en plus coûteuse en espace mémoire. C'est pourquoi, pour optimiser le temps de calcul, l'arbre quaternaire est la structure à favoriser pour la recherche d'éléments.