Projet : Réorganisation d'un réseau de fibres optiques

LU2IN006 – Projet TME 5-10

Iarina Nistor

Yara Khamis

Mai 2024

Sommaire

1 Sujet du Projet

Ce projet concerne la restructuration d'un réseau de fibres optiques dans une agglomération. Le projet implique l'utilisation de plusieurs algorithmes pour répondre aux besoins de reconstitution et de réorganisation du réseau. Pour la reconstitution, on a visé à utiliser les listes, les tables de hachage et les arbres quaternaires, alors que pour l'optimisation de réseau les graphes sont utilisés. Ainsi, le but est de remarquer les différences en termes du temps et de complexité entre ces structures de données et voir laquelle est plus bénéfique dans une situation spécifique.

2 Structure du code

Les différents morceaux de codes sont répartis de la manière suivante :

Structures et fonctions générales :

- Chaine.c & Chaine.h
- Reseau.c & Reseau.h
- Hachage.c & Hachage.h
- ArbreQuat.c & ArbreQuat.h
- Struct_File.c & Struct_File.h
- Graphe.c & Graphe.h

Fonctions de chargement et d'enregistrement :

- Chaine.c & Chaine.h
- Reseau.c & Reseau.h

Jeux de tests pour chaque structure :

- ReconstitueReseau.c
- ChaineMain.c
- essaiGraphe.c
- essaiArbre.c
- essaiHash.c

Menu adapté à toutes les structures :

- ReconstitueReseau.c

Fonctions de calculs pour tester les performances des structures en fonctions de différents paramètres :

- Stats_main.c
- Stats2_main.c

La data avec laquelle on teste nos fonctions est stocke dans les fichiers **.cha* et **.res.*

Les résultats des mesures sont stockés dans le dossier *datasets* (au moment du calcul). Le dossier *datasets* contient également les scripts du terminal gnuplot pour réaliser les graphiques exploitant les résultats obtenus. Ces graphiques sont stockés sous le même nom que leur dataset dans le dossier *plot.*

Les fichiers Stats_main.c et Stats2_main.c testent les performances des structures de manière différente, et les deux fichiers fonctionnent.

Structures et prototypes pour les listes des chaines :

```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;

/* Celllule 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);
CellChaine* lectureLigne(FILE* f);
int lectureNombre(FILE *f, int* nbChain, int* gamma);

void ecrireChaines(Chaines *C, FILE *f);

void afficheChainesSVG(Chaines *C, char* nomInstance);

void freeChaines(Chaines* ch);
void freeCellChaine(CellChaine* cc);
void freeCellPoint(CellPoint* cp);

double longueurChaine(CellChaine *c);
double longueurTotale(Chaines *C);
int comptePointsTotal(Chaines *C);
CellPoint* nouveauCellPoint(double x, double y);
CellChaine* nouveauCellChaine(int num);
Chaines* nouvelleChaines(int nbChaines);


#endif
```

Structures et prototypes pour le réseau – *liste chainée* :

```c
#ifndef __RESEAU_H__
#define __RESEAU_H__
#include "Chaine.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 stock\'e */
    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 */
};

/* 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);

int estVoisin(Noeud* n1, Noeud* n2);
void ajouterVoisin(Noeud* n1, Noeud* n2);

void libererListeCommodites(CellCommodite* cc);
void libererReseau(Reseau* R);
void libererVoisins(CellNoeud* cnd);



void ecrireReseau(Reseau *R, FILE *f);
int nbLiaisons(Reseau *R);
int nbCommodites(Reseau *R);
void afficheReseauSVG(Reseau *R, char* nomInstance);
#endif
```

Manipulation pour le réseau – *hashmap* :

```c
#ifndef __HACHAGE_H__
#define __HACHAGE_H__

#include <stdlib.h>
#include "Reseau.h"

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

int clef(double x,double y);
int hachage(int taille, double x, double y);
Noeud* rechercheCreeNoeudHachage(Reseau* R, TableHachage* H, double x, double y);
Reseau* reconstitueReseauHachage(Chaines *C, int M);
void libererTableHachage(TableHachage* H, int tailleMax);
int calcTaille(int nbElem, double loadFactor);

#endif
```

Manipulation pour le réseau – *arbre quaternaire* :

```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);
ArbreQuat* creerArbreQuat(double xc, double yc, double coteX, double coteY);
void insererNoeudArbre(Noeud* n, ArbreQuat** a, ArbreQuat* parent);
Noeud *rechercheCreeNoeudArbre(Reseau *R, ArbreQuat **a, ArbreQuat *parent, double x,
double y);
Reseau* reconstitueReseauArbre(Chaines* C);
void libererArbreQuat(ArbreQuat *arbre);

#endif
```

Manipulation pour le réseau – *graphe non-oriente* :

```c
#ifndef __GRAPHE_H__
#define __GRAPHE_H__

#include <stdlib.h>
#include <stdio.h>

#include "Struct_File.h"
#include "Reseau.h"


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

typedef struct graphe {
    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;

Graphe* creerGraphe(Reseau* r);
void libererGraphe(Graphe* G);

int* chaineCheminPlusCourt(Graphe* G, int u, int v, int* taille_chaine);
int cheminPlusCourt(Graphe* G, int u, int v);

#endif
```

3 Réponses aux questions

Figure 1: La comparaison entre créer une réseau à partir d'une chaine par une liste et par un arbre (de 500 a 1000)

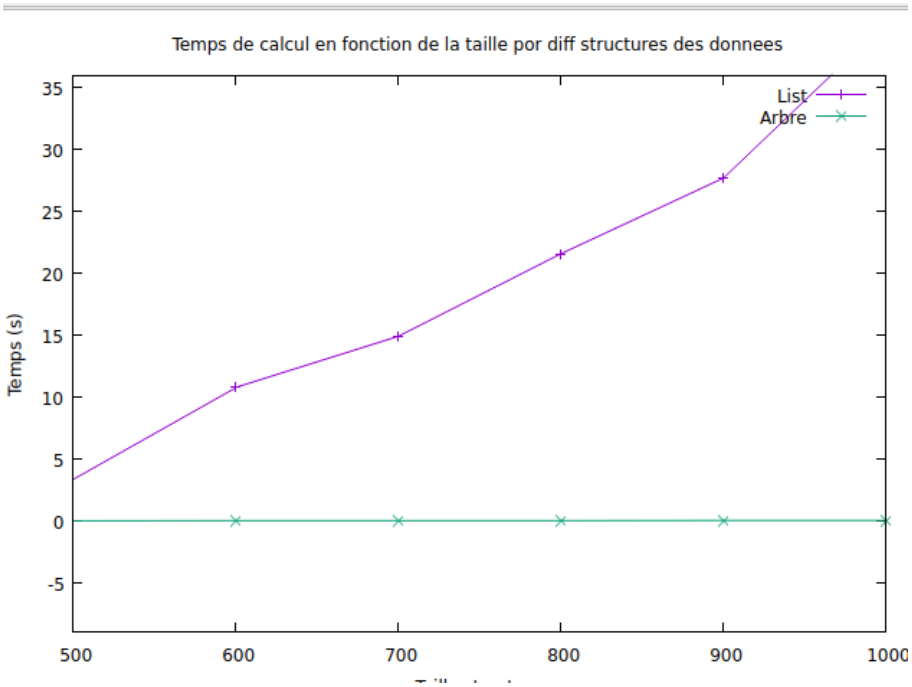
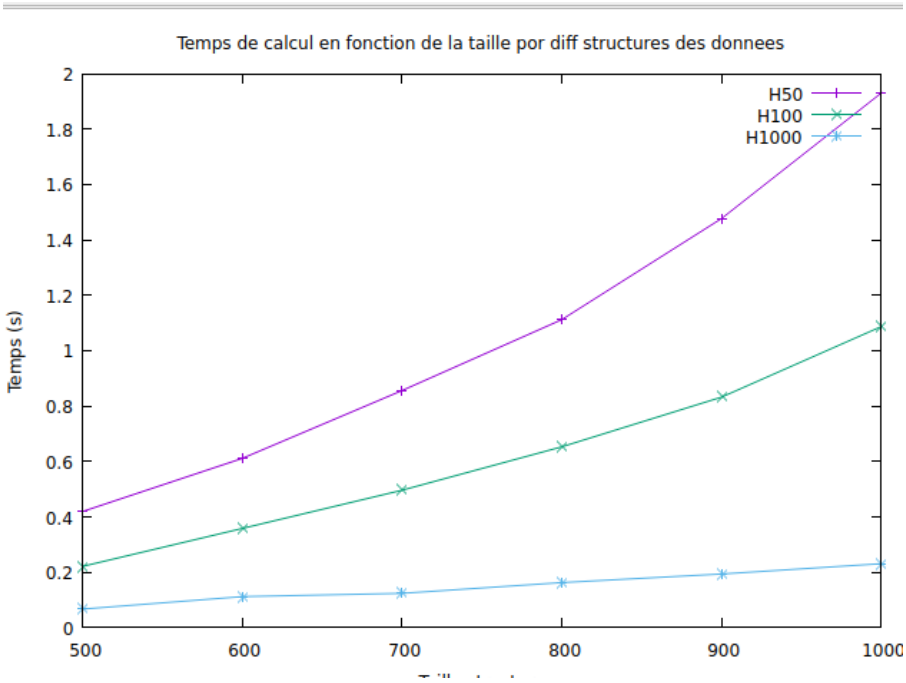


Figure 2 : La comparaison porte sur la création d'un réseau à partir d'une table de hachage de différentes tailles : 5, 100 et 1000, pour un nombre de chaînes variant de 500 à 1000.



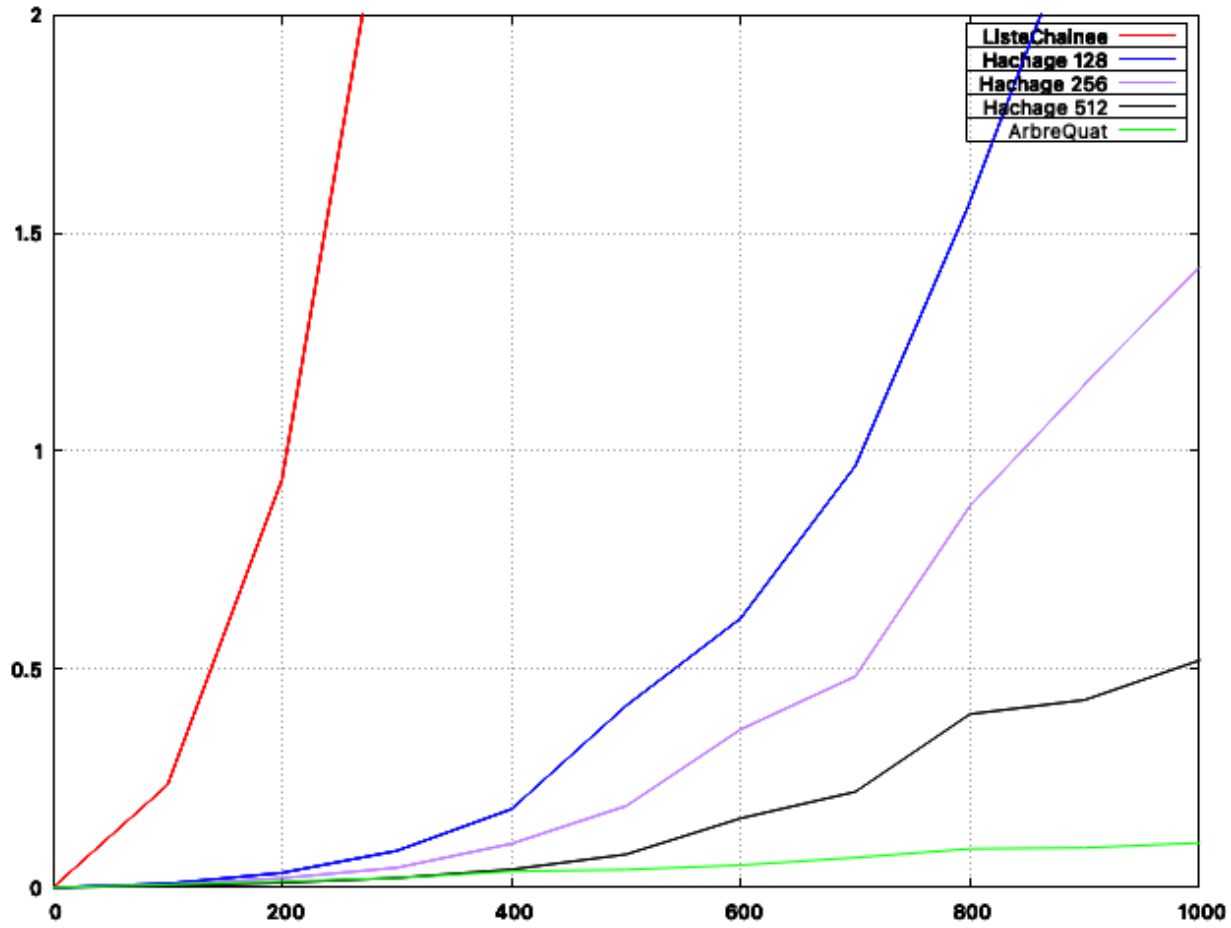


Figure 3 : La comparaison porte sur la création d'un réseau à partir d'une table de hachage de différentes tailles : 128, 256 et 512, une liste et un arbre pour un nombre de chaines variant de 0 à 1000.

Également, la fonction clef donné ne me parait pas approprié car ne retourne pas des entiers, fonction cantor désignée uniquement pour non négative entiers du coup on va faire une petite modification pour notre programme qui prend des doubles
Aussi, la fonction de hachage donné me parait appropriée pour les tailles de tableaux des hachages qu'on a testé, car il n'y a pas des collisions.
Mais, on l'a modifié pour que la calcule de clé soit avec des entiers.

4 Jeux d'essais

Un fichier a été créé qui contient un jeu de tests :

- ReconstitutionReseau.c - qui permet l'utilisateur de choisir ses propres données à manipuler dans le format préétabli et avec la structure/complexité qu'il veut

Ceux-ci testent l'ensemble des fonctions défini pour les 3 structures.
L'utilisation de l'utilitaire *valgrind* lors de l'exécution des exécutables après compilation permet de vérifier l'absence d'erreurs et de fuites mémoires.

5 Analyse des programmes

En exécutant le programme qui mesure les temps de calcul pour chaque méthode sur différentes instances fournies (en faisant varier le nombre de chaînes et la taille de la table de hachage), nous pourrions observer les tendances suivantes :

Liste chaînée : Les temps de calcul augmentent de manière linéaire avec le nombre de chaînes, ce qui est attendu car plus il y a de chaînes, plus il y a de points à traiter lors de la construction du réseau. Cependant, le temps de calcul de la liste chaînée pourrait rester relativement stable ou augmenter de manière moins significative par rapport aux autres méthodes en fonction de la complexité de l'algorithme utilisé.

Table de hachage : Le temps de calcul dépendra non seulement du nombre de chaînes mais aussi de la taille de la table de hachage. Le temps de calcul diminue lorsque la taille de la table de hachage augmente, car cela réduit le nombre moyen de collisions et donc le temps nécessaire pour gérer les collisions.

Cependant, les temps de calcul varient également en fonction du nombre de chaînes, avec une tendance à augmenter à mesure que le nombre de chaînes augmente. Cela peut être dû à une augmentation de la complexité lorsque le nombre de chaînes et donc le nombre total de points à traiter augmentent.

Arbre : Le temps de calcul de la reconstitution par arbre peut être influencé par la structure des données et la complexité de l'algorithme utilisé pour la construction de l'arbre. Les temps de calcul restent relativement stables et généralement plus faibles que ceux des autres méthodes, même lorsque le nombre de chaînes augmente.

Il est possible que le temps de calcul de la reconstitution par arbre soit le plus faible dans certaines situations, en particulier lorsque le nombre de chaînes est élevé et que la distribution spatiale des points est uniforme.

En combinant ces observations, nous pouvons conclure que la méthode de reconstitution par arbre semble offrir les meilleurs temps de calcul, suivie de la table de hachage avec une taille de table plus grande, puis de la liste chaînée. Cependant, le choix de la méthode la plus appropriée dépendra également des spécificités de chaque application, telles que la distribution spatiale des points, la taille de l'ensemble de données et les contraintes de mémoire.