Raport Du Code

Rayane Nasri

2024-05-06

Projet : Réorganisation d'un réseau de fibres optiques

Sujet : Le but de ce projet est la réorganisation d'un réseau de fibres optiques d'une agglomération.

Le projet est consitué de deux parties : - Partie 1 : Reconstitution du réseau. - Partie 2 : Réorganisation du réseau.

La reconstitution du réseau a été réalisée en utilisant diverses structures de données, notamment : - Listes chainées. - Table de hachage. - Arbre quaternaire.

Cela nous a permis de visualiser et d'analyser les temps de calcul associés à chacune de ces structures, ce qui nous a permis de tirer des conclusions.

Description globale du code et des structures manipulées : Ce projet utilise **le langage C** comme langage de programmation principal.

Le code est segmenté en six parties distinctes :

- **Manipulation des chaines du réseau.** *Fichiers associés : Chaine.h, Chaine.c, SVGwriter.h, SVGwriter.c*
- **Reconstitution et stockage du réseau par liste chainée.** *Fichiers associés : Reseau.h, Reseau.c*
- **Reconstitution et stockage du réseau par table de hachage.** *Fichiers associés : Hachage.h, Hachage.c*
- **Reconstitution et stockage du réseau par un arbre quaternaire.** *Fichiers associés : ArbreQuat.h, ArbreQuat.c*
- **Calcul des temps d'excutions pour chaqu'une des structures.** *Fichier associé : timeCal.c*
- **Réorganisation du réseau par un graphe non orienté.** *Fichiers associés : Graphe.h, Graphe.c, Struct_File.h, Struct_File.c*

En règle générale, chaque fichier est divisé en deux sections distinctes. La seconde section est consacrée aux fonctions principales requises lors des TMEs.

La première section est dédiée aux fonctions auxiliaires, comme illustré ci-dessous :

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

/* ~~~~~~ Fonctions auxiliaires ~~~~~~ */
...
/* ~~~~~~ Fonctions principales ~~~~~~ */
...
```

Description des algorithmes utilisés :

En utilisant une structure Chaines, définie dans le code C ci-dessous, l'objectif est de concevoir une nouvelle structure nommée Reseau, pour stocker le réseau de fibres optiques de la structure Chaines.

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

```
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;
```

On vous invite à reprendre les deux fichiers headers *Chaines.h* et *Reseau.h* pour plus d'informations sur ces structures. L'algorithme est alors définit ainsi :

```
Algorithme reconstitueReseau(Chaines C):
    // Allocation de l'espace mémoire pour la structure Reseau
    Reseau <- AllouerMemoire()
    // Initialisation de la liste des noeuds et des commodités à NULL
    Reseau.noeuds <- NULL
```

```
Reseau.commodites <- NULL

// Initialisation d'une matrice carrée de taille n*n
n <- CalculerNombrePointsTotals(C)
Matrice <- InitialiserMatrice(n)

// Parcours des chaines de C
Pour chaque chaine dans C:
    // Variable temporaire pour stocker le premier point de la chaine
    PointTemporaire <- NULL

    // Parcours de chaque point de la chaine courante
    Pour chaque point dans chaine:
        // Vérification si le point a déjà été ajouté au réseau
        Si PointExisteDeja(point):
            PointCourant <- RecupererPoint(point)
        Sinon:
            PointCourant <- CreerPoint(point)

        // Si le point n'est pas le premier de la chaine
        Si point != PremierPoint(chaine):
            Si LiaisonDejaRencontree(pointPrecedent, pointCourant) == False:
                MettreAJourListeVoisins(pointPrecedent, pointCourant)

        // Si le point est le premier de sa chaine
        Si point == PremierPoint(chaine):
            PointTemporaire <- pointCourant

        // Si le point est le dernier de sa chaine
        Si point == DernierPoint(chaine):
            Comodite <- CreerComodite(PointTemporaire, pointCourant)
            AjouterComodite(Reseau.commodites, Comodite)

        pointPrecedent <- pointCourant

Retourner Reseau
```

L'algorithme de reconstitution reste identique pour les trois structures, leur distinction réside dans la vérification de l'appartenance du point au réseau.

En utilisant la structure Reseau, l'objectif est de déterminer si en suivant systématiquement le chemin le plus court entre les deux extrémités de chaque commodité, nous pouvons respecter la contrainte gamma. L'algorithme est alors définit ainsi :

```
Algorithme reorganiseReseau(Reseau reseau):
    // Création du graphe non orienté associé au réseau
    Graphe <- CreerGrapheNonOriente(reseau)

    // Création d'un tableau de chemins contenant les plus courts chemins entre les extrémi
    TableauChemins <- CreerTableauChemins(reseau.nbCommodites)

    // Création d'une matrice n*n où n est le nombre de nœuds du réseau
    MatriceAdjacence <- CreerMatriceAdjacence(reseau)

    // Parcours du tableau de chemins
    Pour chaque chemin dans TableauChemins:
        // Parcours de chaque nœud du chemin
        Pour chaque noeud dans chemin:
            Tant que il y a un suivant:
                // Notation du numéro du nœud courant et du suivant
                i <- numero(noeud)
                j <- numero(suivant)
                // Incrémentation de matrice[i][j] et matrice[j][i]
                MatriceAdjacence[i][j]++
                MatriceAdjacence[j][i]++

        // Vérification si matrice[i][j] > gamma
          Si MatriceAdjacence[i][j] > reseau.gamma:
              Retourner Faux

    // Si la vérification est complète sans retourner faux, alors retourner vrai
    Retourner Vrai
```

Les algorithmes mentionnés précédemment représentent les deux principaux piliers du projet.

A partir d'un graphe non orienté $G = (V, E)$, d'un sommet u et d'un sommet v, tel que $u, v \in V$, l'objectif est de construire le plus court chemin de u vers v. L'algorithme est alors définit ainsi :

```
Algorithme plus_court_chemin(Graphe G, Sommet u, Sommet v):
    // Création du tableau predecesseurs qui nous permet de stocker le plus court chemin
    Déclarer un tableau predecesseurs de taille G.nbSom
    Pour chaque indice i de 0 à G.nbSom - 1:
        predecesseurs[i] <- -1
    predecesseurs[u.num] <- -2

    // Création de la liste résultat à retourner
    Déclarer une file res

    // Initialisation de la file res
    Init_file(res)

    // Création de la file f
    Déclarer une file f
    Allouer mémoire pour f
    Si f est NULL:
        Afficher "Erreur lors de l'allocation mémoire."
        Retourner NULL
    Init_file(f)
    Enfiler(f, u.num)

    Tant que f n est pas vide:
        numero <- Defiler(f)
        Si numero est égal à v.num:
            Tant que numero n est pas égal à u.num:
                Enfiler_tete(res, numero)
                numero <- predecesseurs[numero]
            Enfiler_tete(res, u.num)

        som_def <- sommetNum(G, numero)
        tmp <- som_def.L_Voisins
        Tant que tmp n est pas NULL:
            Si predecesseurs[tmp.a.u] est égal à -1:
                Enfiler(f, tmp.a.u)
                predecesseurs[tmp.a.u] <- som_def.num
            Si predecesseurs[tmp.a.v] est égal à -1:
                Enfiler(f, tmp.a.v)
                predecesseurs[tmp.a.v] <- som_def.num
            tmp <- tmp.suivant
```

```
Retourner res
```

Cet algorithme nous a semblé très intéressant, ce qui nous a poussés à en discuter.

Réponses au questions et analyse des résultats obtenus : Dans cette partie, nous aborderons les questions posées dans l'énoncé du projet et analyserons les résultats obtenus lors des calculs des temps d'exécution du programme pour les différentes structures.

- **Q 4.2** Vous pouvez utiliser la fonction clef $f(x, y) = y+(x+y)(x+y+1)/2$. Tester les clefs générées pour les points (x, y) avec x entier allant de 1 à 10 et y entier allant de 1 à 10. Est-ce que la fonction clef vous semble appropriée ?

  Pour répondre à cette question, nous avons opté pour l'écriture d'un script Python qui calcule $f(x, y)$ pour x un entier compris entre 1 et 10 et y un entier compris entre 1 et 10. Il affiche le message "Collision !" dès qu'il rencontre deux fois la même clé. Voici le code Python correspondant :

```
l = []
for x in range(11):
    for y in range(11):

        n = y + (x + y)*(x + y + 1) // 2
        if n in l:
            print("Collision !")
        l.append(n)
```

  Lors de l'exécution du script, aucun message n'a été affiché sur le terminal, ce qui confirme le bon fonctionnement de la fonction f.

1. Analyse du temps d'excution de l'algorithme reconstitueReseau avec des listes chainées. En exécutant la fonction **reconstitueReseauLC(Chaines* C)** sur diverses instances de Chaines, débutant par un ensemble de 500 chaînes de 100 points jusqu'à 5000 chaînes de 100 points, nous avons généré le graphe suivant :

On peut facilement calculer la complexité théorique de la fonction **reconstitueReseauLC(Chaines* C)** en effet :

Soit n le nombre de chaines de l'ensemble. Soit m le nombre moyen de points dans chaqu'une des chaines. Soit $i \in [1, n.m]$ un entier.

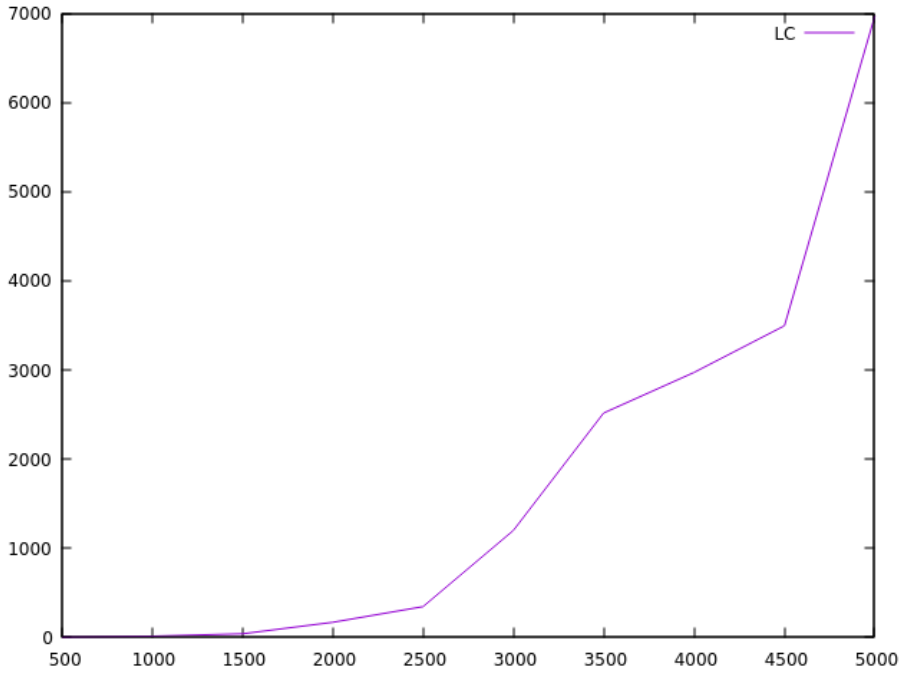

Figure 1: “Graphe 01”

On remarque deux boucles while imbriquées, la première est executée n fois et la deuxiéme m fois.

Si l'on considère ces deux boucles comme une seule executée $n.m$ fois, la complexité de la fonction **rechercheCreeNoeudListe(Reseau* R, double x, double y)** est donnée comme :

- $\Omega(1)$ pour le meilleur des cas.
- $O(i)$ pour le pire des cas.

La complexité de la fonction **reconstitueReseauLC(Chaines* C)** est donc donnée par :

- $\sum_{i=0}^{n.m} 1 = n.m$, pour le meilleur des cas on a $\Omega(n.m)$.
- $\sum_{i=0}^{n.m} i = (n.m)^2$, pour le pire des cas on a $O((n.m)^2)$

On peut vérifier si les résultats théoriques sont compatibles avec les résultas pratiques en trouvant deux réels α et β tel que : $\alpha.(n.m) < G < \beta.(n.m)^2$.

On obtient ainsi le graphe suivant :

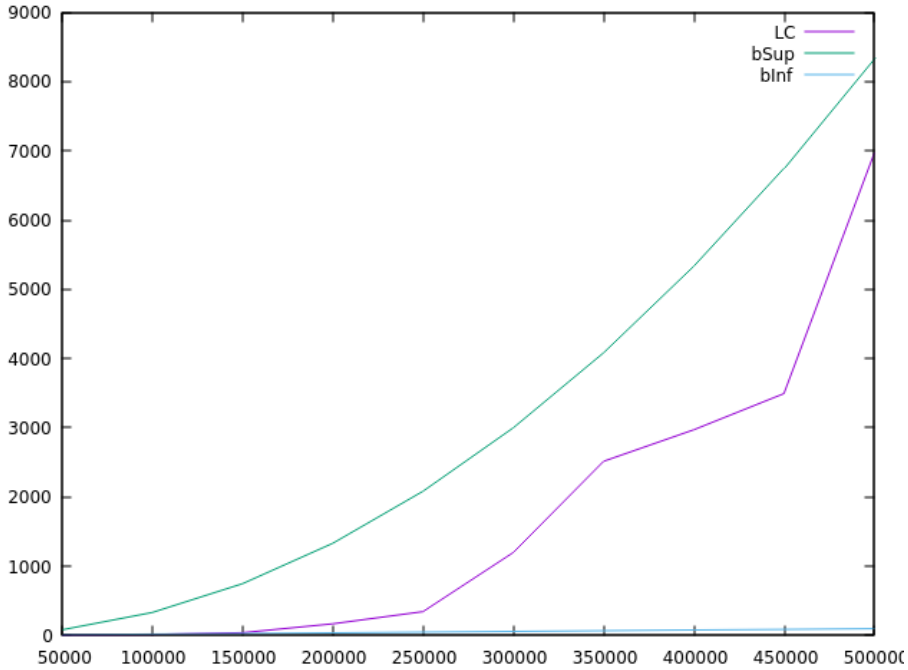

Figure 2: "Graphe 02"

Il est évident que la complexité moyenne n'est pas linéaire, mais plutôt quadratique.

Ainsi, on peut en déduire que la liste chaînée n'est pas la structure de données appropriée pour ce problème.

2. Analyse du temps d'excution de l'algorithme reconstitueReseau avec des tables de hachage de tailles variant de 1000 à 5000. En exécutant la fonction **reconstitueReseauH(Chaines* C, int m)** sur diverses instances de Chaines, débutant par un ensemble de 500 chaînes de 100 points jusqu'à 5000 chaînes de 100 points, et avec des valeurs de m = {500, 1000, 5000}, nous avons généré les graphes suivant :

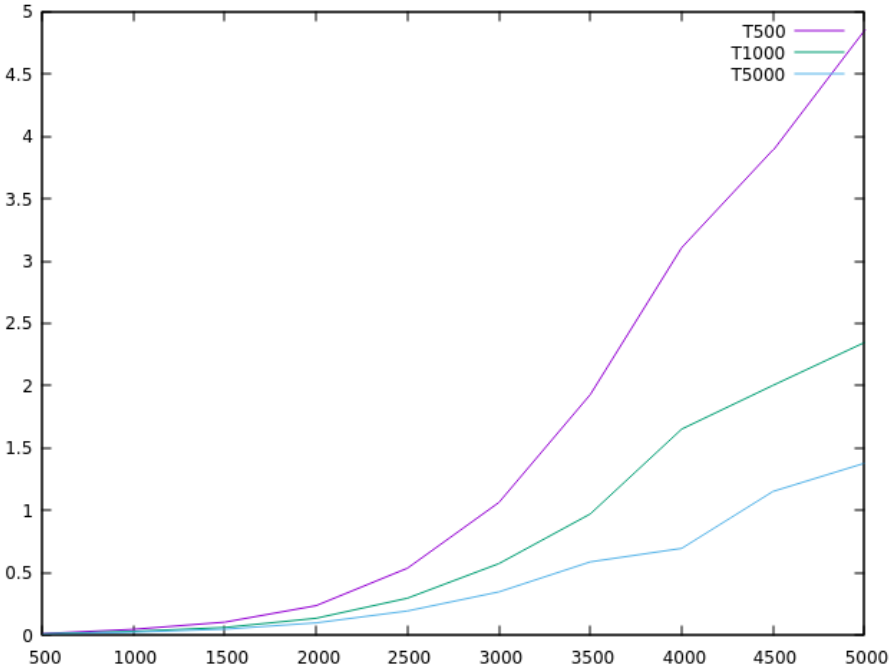

Figure 3: "Graphe 03"

On peut calculer la complexité théorique de la fonction **reconstitueReseauH(Chaines* C, int M)** en effet :

> Soit n le nombre de chaines de l'ensemble. Soit m le nombre moyen de points dans chaqu'une des chaines. Soit $i \in [1, n.m]$ un entier. Soit α le facteur de charge de la table de hachage.
>
> On remarque deux boucles while imbriquées, la première est executée n fois et la deuxiéme m fois.
>
> Si l'on considère ces deux boucles comme une seule executée $n.m$ fois, la complexité de la fonction **rechercheCreeNoeud-**

Hachage(Reseau* R, double x, double y) est donnée comme :

- $\Omega(1)$ pour le meilleur des cas.
- $O(\alpha)$ pour le pire des cas.

La complexité de la fonction **reconstitueReseauH(Chaines* C, int M)** est donc donnée par :

- $\sum_{i=0}^{n.m} 1 = n.m$, pour le meilleur des cas on a $\Omega(n.m)$.
- $\sum_{i=0}^{n.m} \frac{i}{M} = \frac{(n.m).(n.m+1)}{M}$, pour le pire des cas on a $O(\frac{(n.m)^2}{M})$.

On peut vérifier si les résultats théoriques sont compatibles avec les résultas pratiques en trouvant deux réels α et β tel que : $\alpha.(n.m) < G < \beta.\frac{(n.m)^2}{M}$.

On obtient ainsi les graphe suivant pour une table de hachage de taille 1000:

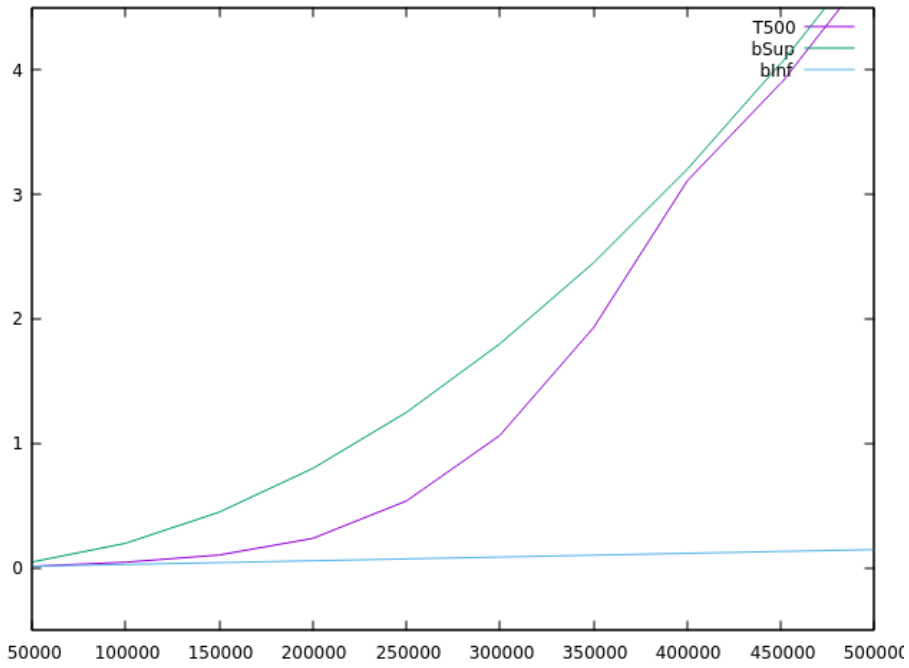

Figure 4: "Graphe 04"

Il est évident également que la complexité moyenne n'est pas linéaire, mais plutôt quadratique.

Par conséquent, nous pouvons conclure que la table de hachage est une structure de données adaptée à ce problème, bien que peut-être pas la meilleure. Nous pourrons confirmer cela lorsque nous examinerons l'arbre quaternaire. Cependant, elle s'avère bien plus efficace qu'une liste chaînée.

3. Analyse du temps d'excution de l'algorithme reconstitueReseau avec un arbre quaternaire. En exécutant la fonction **reconstitueReseauH(Chaines* C, int m)** sur diverses instances de Chaines, débutant par un ensemble de 500 chaînes de 100 points jusqu'à 5000 chaînes de 100 points, nous avons généré le graphe suivant :

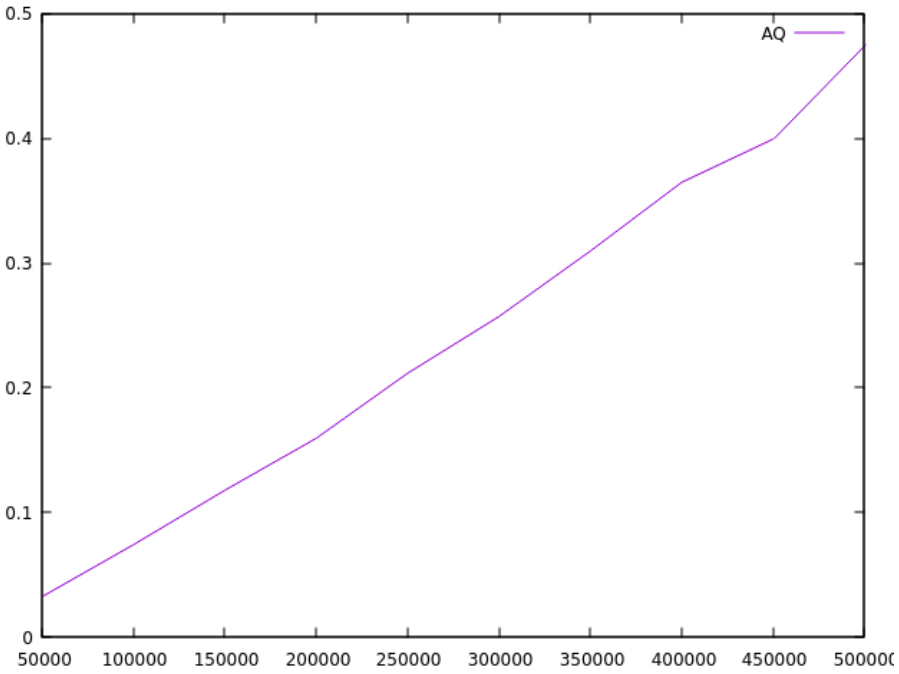

Figure 5: "Graphe 05"

Avec la même démarche qu'auparavant on a trouvé une complexité théorique en :

- $\theta(log_4((n.m)!))$

Initialement, cette idée nous a paru absurde, car en pratique, elle représente la structure la plus efficace, comme le suggère le graphe précédent. La présence du facteur de factorielle était surprenante, cependant, nous avons rapidement réalisé que $\theta(log_4((n.m)!)) \subset \theta((n.m).log_4(n.m))$ ce qui est une complexité d'accès bien connue en algorithmique.

On peut vérifier si les résultats théoriques sont compatibles avec les résultas pratiques en trouvant deux réels α et β tel que : $\alpha.(log_4((n.m)!)) < G < \beta.(log_4((n.m)!))$.

On obtient ainsi les graphe suivant :

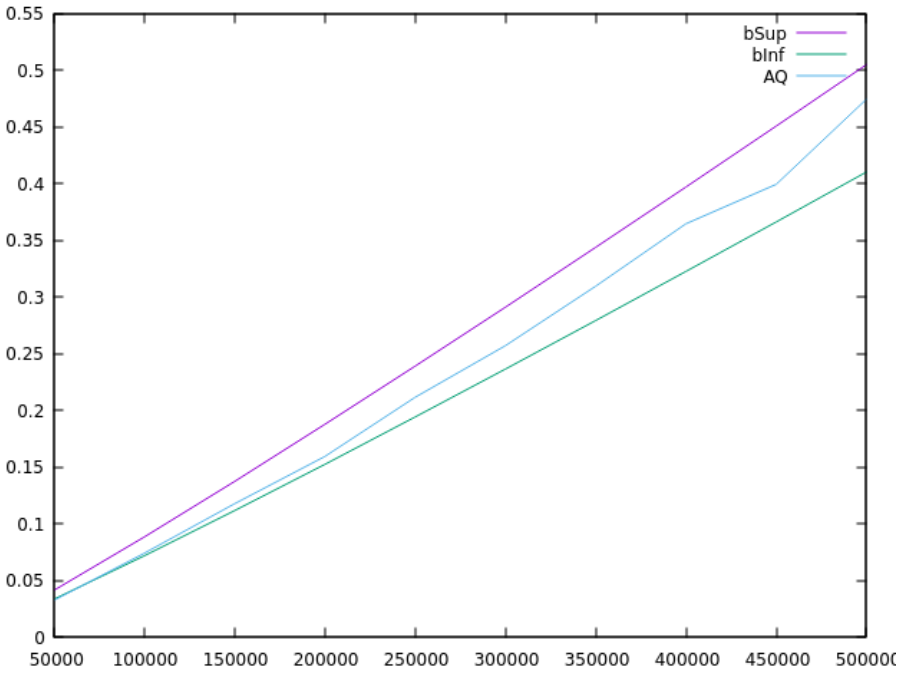

Figure 6: “Graphe 06”

Ainsi, nous pouvons en conclure que parmi les trois structures étudiées, l'arbre quaternaire est la plus adaptée pour ce problème.

4. Analyse du temps d'excution de l'algorithme reorganiseReseau. En analysant le programme sur les instances 1, 2, 3, 4 et 5, il est clair que le temps d'exécution augmente rapidement.

Cette accélération pourrait être attribuée à la matrice "matrix", qui semble rapidement consommer beaucoup de mémoire.

Une solution pourrait consister à utiliser une table de hachage pour suivre les liaisons déjà rencontrées et ainsi optimiser l'utilisation de la mémoire.

Description des jeux d'essais utilisés : Pour confirmer le bon fonctionnement de nos fonctions, nous avons développé des fichiers main tels que **ReconstitueReseau.c**, **ChaineMain.c** et **GrapheMain.c**.

Ces fichiers regroupent l'ensemble des éléments de code de chaque partie.

Reprenons par exemple le fichier **ReconstitueReseau.c** dont le contenu est donnée ci-dessous :

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

#include "Reseau.h"
#include "Hachage.h"
#include "ArbreQuat.h"

int main(int argc, char* argv[]){

    char* nomf = argv[1];
    int i = atoi(argv[2]);

    FILE* fic1 = fopen(nomf, "r");
    if(fic1 == NULL){
        fprintf(stderr, "FILE: %s, LINE: %d, Erreur lors de l'ouverture du fichier.\n", __FI
        return 0;
    }

    Chaines* C = lectureChaines(fic1);
    fclose(fic1);

    Reseau* res;
    switch(i){
        case(1):
            res = reconstitueReseauListe(C);
```

```
            FILE* fic2 = fopen("reseau.txt", "w");
            if(fic2 == NULL){
                fprintf(stderr, "FILE: %s, LINE: %d, Erreur lors de l'ouverture du fichier.\
                return 0;
            }

            afficheReseauSVG(res, "Affichage_reseau_LC");
            ecrireReseau(res, fic2);
            fclose(fic2);
            break;

        case(2):
            res = reconstitueReseauHachage(C, 10);

            fic2 = fopen("reseau.txt", "w");
            if(fic2 == NULL){
                fprintf(stderr, "FILE: %s, LINE: %d, Erreur lors de l'ouverture du fichier.\
                return 0;
            }

            afficheReseauSVG(res, "Affichage_reseau_Hach");
            ecrireReseau(res, fic2);
            fclose(fic2);
            break;

        case(3):
            res = reconstitueReseauArbre(C);

            fic2 = fopen("reseau.txt", "w");
            if(fic2 == NULL){
                fprintf(stderr, "FILE: %s, LINE: %d, Erreur lors de l'ouverture du fichier.\
                return 0;
            }

            afficheReseauSVG(res, "Affichage_reseau_Arbre");
            ecrireReseau(res, fic2);
            fclose(fic2);
            break;

        default:
            break;
    }

    detruireChaine(C);
    detruireReseau(res);
```

```
    return 0;
}
```

On remarque que ce fichier teste les fonctions suivantes :

- **lectureChaines(FILE* f)**
- **detruireChaine(Chaines* C)**
- **reconstitueReseauListe(Reseau* R)**
- **ecrireReseau(Reseau* R, FILE* f)**
- **afficheReseauSVG(Reseau* R)**
- **detruireReseau(Reseau* R)**
- **reconstitueReseauHachage(Reseau* R, int m)**
- **reconstitueReseauArbre(Reseau* R)**

Étant donné que toutes les fonctions des fichiers *Reseau.c*, *Hachage.c* et *ArbreQuat.c* sont construites autour des fonctions principales **reconstitueReseauListe(Reseau* R)**, **reconstitueReseauHachage(Reseau* R, int m)** et **reconstitueReseauArbre(Reseau* R)**, le bon fonctionnement de ces trois fonctions garantit la validité de la plupart des fonctions codées dans ces fichiers.

En ce qui concerne les fonctions de libération **detruireReseau(Reseau* R)** et **detruireChaine(Chaines* C)**, l'exécution du programme avec Valgrind nous permet de déterminer si ces fonctions fonctionnent correctement ou non.

De plus l'avantage avec ce fichier, c'est le fait qu'on peut tester nos fonctions sur plusieurs instances de chaines (**00014_burma.cha**, **07397_pla.cha**, **05000_USA-road-d-NY.cha**) sans modifier le code.

Conclusion : Ce projet nous a permis de comprendre l'importance capitale du choix de la structure de données utilisée.

Bien que chaque structure de données ait ses points forts, certaines structures sont plus adaptées à certains problèmes que d'autres.

Il est indéniable que l'arbre quaternaire représente la solution la plus efficace pour notre problème. Cependant, la différence ne se manifeste que lors du traitement de données massives. Il est également important de ne pas sous-estimer la complexité de mise en œuvre de ce type de structures.

Ainsi, lorsque nous traitons un petit volume de données, il est préférable d'utiliser une liste chaînée ou une table de hachage. En revanche, pour des volumes massifs de données, il est évident que l'arbre quaternaire est la meilleure solution.

On observe cependant que la table de hachage représente un compromis intéressant : bien qu'elle puisse perdre légèrement en capacité par rapport à l'arbre

quaternaire, elle offre en revanche de nombreux avantages en termes de temps d'implémentation.