Projet SDD

Olando Bazil 28608499
Cyrena Ramdani 3805942

Notre Makefile crée au total 5 exécutables qui permettent de tester les différents exercices:

- ChaineMain : exercice 1
- ReseauMain : exercice 2 - 3
- ReconstitueReseau : exercice 4 - 5 - 7
- speedtest6_1: exercice 6.1
- speedtest6_3 : exercice 6.3

I. Exercice 1

Chaine.h contient les structures :
- CellPoint
- CellChaine
- Chaines

Et les fonctions des fichiers:
- gestionChaine.c
- Chaine.c
- mesureChaine.c

Pour tester toutes les fonctions de l'exo1 il faut utiliser les commandes : *le nom de fichier html est facultatif*

```
./ChaineMain <Chaine_file> <nom_fichier_html>
```

En utilisant l'exécutable : ``ChaineMain``

Ce qui va
- Ouvrir <Chaine_file> et le lire pour créer une Chaine
- Ecrire dans le meme format dans un fichier ``Data.txt``
- Créer un fichier SVG nommé <nom_fichier_html>.html ou nomInstance.html si le 2e paramètre n'est pas précisé
- Compter le nombre total de point, la longueur de la 1ere chaine et de la longueur totale des chaînes

II. Exercice 2

Reseau.h contient les structures :
- CellNoeud
- Noeud
- CellCommodite
- Reseau

Et les fonctions des fichiers:
- gestionReseau.c
- Reseau.c
- ReconstitueReseau.c

Pour tester toutes les fonctions de l'exercice 2 il faut utiliser les commandes :

```
./ReconstitueReseau <Chaine_file> <methode Reseau : [1-3]>
```

methode_reseau est un chiffre entre 1 et 3 qui déterminera la manière dont sera reconstitué le réseau:
- 1 : Liste
- 2 : Table de Hachage
- 3 : Arbre Quaternaire

En utilisant l'exécutable : ``ReconstitueReseau``

Ce qui va
- Vérifier que tous les paramètres sont correctes
- Ouvrir <Chaine_file> et le lire pour créer une Chaîne
- Reconstitue le réseau à partir de la chaîne avec la méthode choisie et l'écrit dans un fichier ``Reconstitue.res``
- Afficher les caractéristiques liées à la chaîne et au réseau (exo 1 - 3)

III. Exercice 3

Les fonctions des fichiers:
- Reseau.c
- mesureReseau.c

Pour tester toutes les fonctions de l'exercice 3 il faut utiliser les commandes : *le nom de fichier html est facultatif*

```
./ReseauMain <Chaine_file> <nom_Instance>
```

En utilisant l'exécutable : ``ReseauMain``

Ce qui va
- Ouvrir <Chaine_file> et le lire pour créer une Chaîne
- Reconstitue le réseau à partir de la chaîne et l'écrit dans un fichier ``Data.res``
- Créer un fichier SVG nommé ``<nom_fichier_html>.html`` ou ``nomInstance.html`` si le 2eme paramètre n'est pas précisé a partir de la chaîne et du réseau obtenu pour faire la comparaison
- Compter le nombre de commodités, liaisons et noeuds

IV. Exercice 4

2. pour des entiers de 1 à 10 on obtient les clés suivantes :

```
4 8 13 19 26 34 43 53 64 76
7 12 18 25 33 42 52 63 75 88
11 17 24 32 41 51 62 74 87 101
16 23 31 40 50 61 73 86 100 115
22 30 39 49 60 72 85 99 114 130
29 38 48 59 71 84 98 113 129 146
37 47 58 70 83 97 112 128 145 163
46 57 69 82 96 111 127 144 162 181
56 68 81 95 110 126 143 161 180 200
67 80 94 109 125 142 160 179 199 220
```

Cette fonction clef semble appropriée car tous les nombres sont différents donc, on a une unicité pour chaque clé.

4. Dans un premier temps on cherche si le Noeuds existe sinon on le crée tout en mettant à jour ses voisins. Retourne le résultat

Pour tester : utiliser ``ReconstitueReseau`` avec comme 2e paramètre `2`

```
./ReconstitueReseau <Chaine_file> <methode Reseau : 2>
```

V. Exercice 5

Pour tester : utiliser ``ReconstitueReseau`` avec comme 2e paramètre `3`

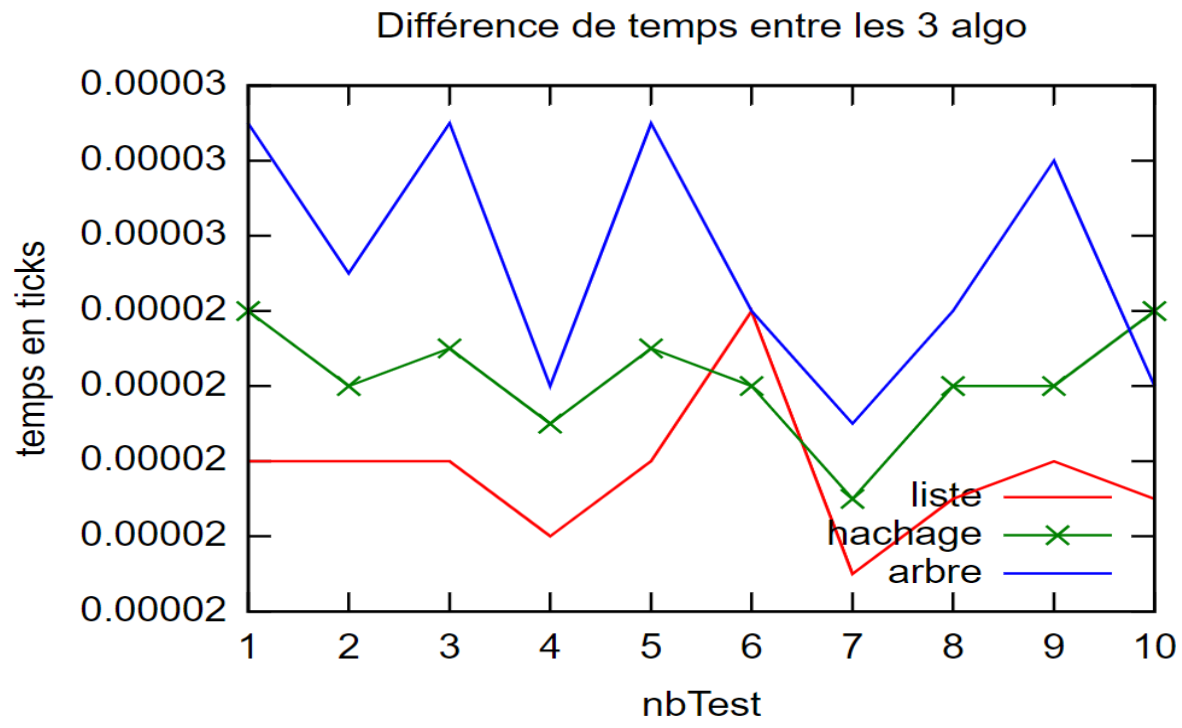
```
./ReconstitueReseau <Chaine_file> <methode Reseau : 3>
```

L'exercice 5 est totalement fonctionnel mais nous avons néanmoins un leak au niveau de RechercheCreerNoeud() que nous n'avons pas pu corriger. En effet, la fonction libererArbre(AbrQ*) se termine sur un Segmentation Fault ce qui nous empêche de libérer la mémoire allouée pour l'arbre. (il suffit de décommenter la ligne à la fin de reconstitueReseauArbre(Chaine*) pour l'observer).
Ce leak est donc présent dans toutes les fonctions relatives à l'arbre quaternaire.

VI. Exercice 6

Pour réaliser les graphiques suivants, nous avons utilisé Gnuplot.

6.1



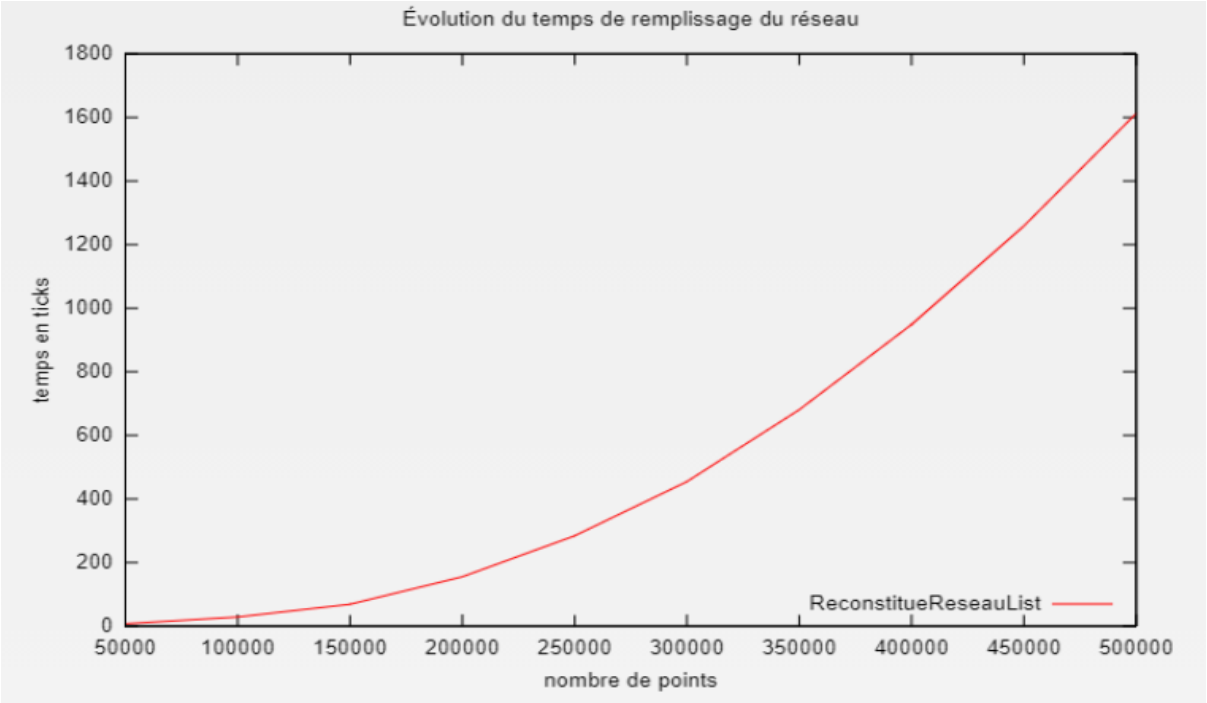
Pour avoir un échantillon de données

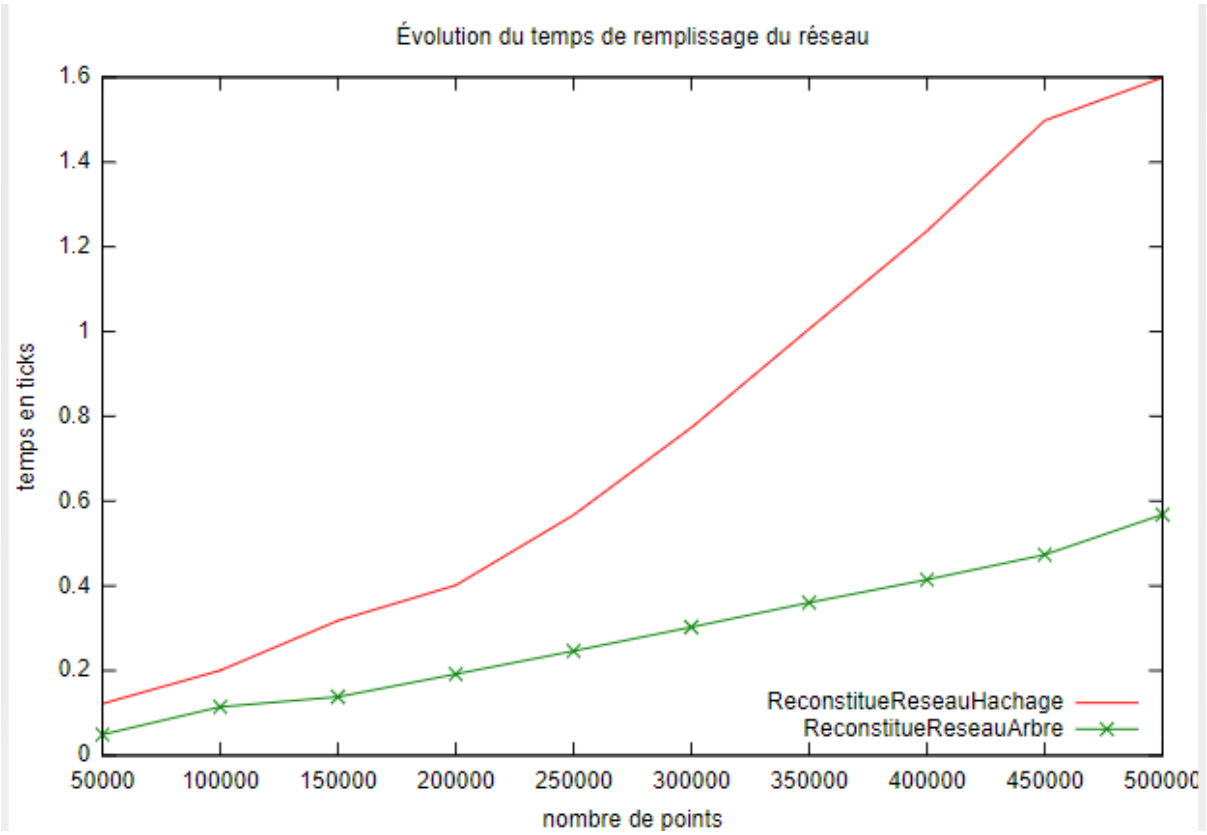
```
./speedtest6_1 (taille matrice de hachage)
```

Pour avoir ce graphique on utilise la même technique que pour les Tme précédent à savoir d'utiliser clock(); entre chaque algorithme.

Au niveau des datas, on a quasiment le même temps à 1*10^-5 près. Cependant on remarque qu'au niveau du temps dans le graphe, qu'au niveau du temps, l'ordre de rapidité est (du plus rapide au plus lent) ;
liste -> hachage -> arbre
Comme nous pouvons le voir sur le graphe ci dessous pour les 3 algorithme (même en faisant changer la taille de la table de hachage cela n'apport pas de changement significatif)

6.3.





Pour la table de hachage nous avons remarqué que le plus optimal était de prendre pour la taille de la table de hachage le nbchaines. Car on ne peut pas prendre une valeur fixe étant donné que l'on passe de 500*100 à 5000*100 points et également car grâce à cela.

pour chaque case du tableau il y aura 100 points dans le meilleur cas ce qui nous donne une complexité d'O(100) pour chercher chaque point.
Pour avoir un échantillon de données

```
./speedtest6_3
```

pour avoir ces graphique on utilise la même méthode que pour la 6.1

6.4

Nous observons avec assez de facilité que reconstituer un réseau à partir de sa liste de nœud est extrêmement long concernant le premier graphique, et que le temps de remplissage est exponentiel contrairement aux deux autres méthodes.
En effet à 500 000 nœuds la reconstitution du réseau avec la table de hachage nous fait gagner 100 fois plus de temps, mais la version la plus rapide, reste la reconstitution du réseau grâce à l'arbre.

VII. Exercice 7 :

7.2. *int tailleCheminCourt(Graphe* g, int u, int v);*
On adapte l'algorithme proposé en cours pour le parcours en largeur

7.3: *ListeEntier* cheminCourt(Graphe* g, int u, int v);*
Ici on utilise le parcours en largeur vue en cours, on stocke donc les différents niveaux de l'arborescence de "l'arbre" de racine u dans un tableau D tout en mettant à jour une matrice adjacente ayant pour rôle de garder en mémoire les arêtes de cette arborescence. Une fois le sommet v rencontre on fait appel à la fonction getChaine(..) qui crée la ListeEntier* ou la chaîne {u,v}sera retourné. Afin de retourner le chaîne la plus courte entre u et v on procède à l'envers en partant de v pour arriver à u.

En résumé :

- les niveaux de l'arborescence sont stockés dans le tableau D (comme dans la question précédente)
- les chemins issus de u sont stockés dans la matrice "matriceAr"
- la chaîne qui contient le chemin le plus court entre u et v sera sous forme de ListeEntier*

7.4

On crée une matrice adjacente stockant tous les passages dans chaque arête puis on effectue chaque chaque chemin entre les commodites du graphe tout en mettant à jour cette matrice. Si la valeur d'une arête est supérieur à gamma on arrête de parcourir les commodités et on retourne 0.

La complexité la plus grande serait donc d'avoir un graphe ou le nombre de chaînes qui passe par arête est inférieur à gamma. La plus petite complexité serait d'avoir les gamma 1ere chaînes passant par la même arrêté

7.5

En testant notre fonction nous nous rendons compte que plus le graphe est énorme , plus notre matrice peine à tenir le rythme. Ainsi un nombre trop conséquent de sommet demande une matrice de taille (nbsommet ^ 2) ce qui n'est pas viable.

De plus cette fonction dépend de la manière dont on implémente la fonction cheminCourt car il peut y avoir différents chemins de même taille pour relier 2 points sans passer par les mêmes arêtes donc cela fausser le résultat de réorganiseRéseau.

Pour améliorer cette fonction on pourrait changer la structure et stocker le nombre de passage directement dans l'arrête (car elle est stockée une seule fois). Ce qui nous éviterait d'avoir une matrice de taille exponentielle.