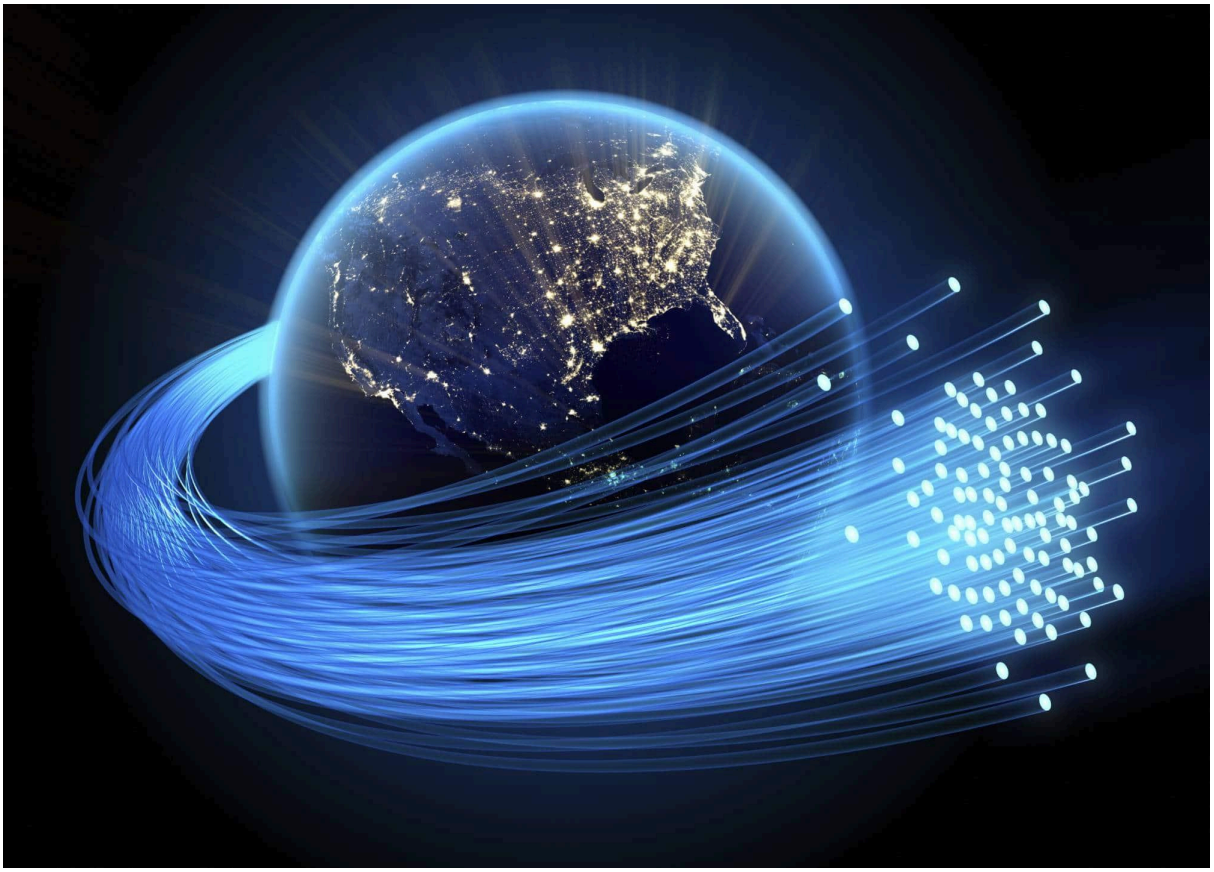
MOULA Lisa 21215357
DUPART Guillaume 28710554

LU2IN006 : Projet Final

Compte Rendu



Enseignants :
Romain Vacheret
Badmavasan Kirouchenassamy

Sommaire

I) Reformulation du sujet

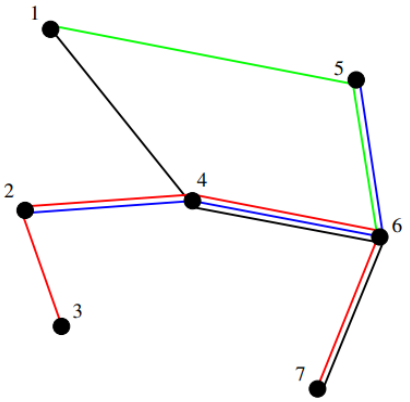


FIGURE 1 – Un exemple de réseau.

Le projet porte sur l'amélioration d'un réseau de fibres optiques dans une agglomération, divisé en deux étapes :

-Étape 1 : la reconstitution du réseau existant

-Étape 2 : La réorganisation du réseau.

Actuellement, les opérateurs détiennent des parties du réseau sans qu'il existe un plan complet. La première phase vise à établir ce plan tandis que la seconde consiste à redistribuer les fibres pour optimiser l'utilisation du réseau. L'objectif est de proposer des approches efficaces pour ces deux parties en testant divers algorithmes. Le réseau est représenté par des câbles renfermant plusieurs fibres optiques qui relient des clients et des concentrateurs. Les instances utilisées proviennent de bases de données standard pour évaluer l'efficacité des algorithmes.

II) Description des structures manipulées & Description globale du code

Dans un premier temps, nous allons décrire les structures de données que nous avons initialisées durant ce projet. Puis nous allons faire une description globale de notre code.

- Description des structures de données

Chaine.h		
CellPoint	**CellChaine**	**Chaines**
- Structure représentant un point dans une liste chaînée. - Attributs : - `x`, `y` : Coordonnées du point. - `suiv` : Pointeur vers la cellule suivante dans la liste.	- Structure représentant une chaîne dans une liste chaînée de chaînes. - Attributs : - `numéro` : Numéro de la chaîne. - `points` : Liste des points de la chaîne (une liste chaînée de `CellPoint`). - `suiv` : Pointeur vers la cellule suivante dans la liste.	- Structure représentant l'ensemble des chaînes. - Attributs : - `gamma` : Nombre maximal de fibres par câble. - `nbChaines` : Nombre de chaînes. - `chaînes` : Liste chaînée des chaînes (une liste chaînée de `CellChaine`).

SORBONNE UNIVERSITÉ

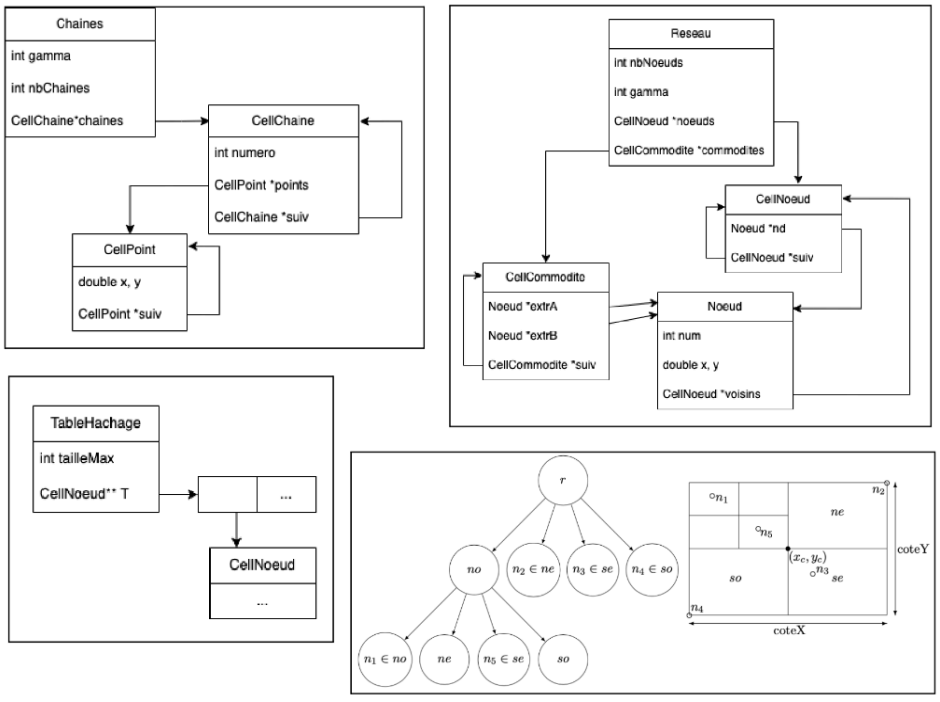
Reseau.h		
Noeud	**CellNoeud**	**CellCommodite**
- Structure représentant un nœud dans le réseau. - Attributs : - `num` : Numéro du nœud. - `x`, `y` : Coordonnées du nœud. - `voisins` : Liste des voisins du nœud (une liste chaînée de `CellNoeud`).	- Structure représentant une cellule dans une liste chaînée de nœuds. - Attributs : - `nd` : Pointeur vers le nœud stocké. - `suiv` : Pointeur vers la cellule suivante dans la liste.	- Structure représentant une commodité, c'est-à-dire une paire de nœuds à relier. - Attributs : - `extrA`, `extrB` : Nœuds aux extrémités de la commodité. - `suiv` : Pointeur vers la cellule suivante dans la liste.
Reseau		
- Structure représentant l'ensemble du réseau. - Attributs : - `nbNoeuds` : Nombre de nœuds dans le réseau. - `gamma` : Nombre maximal de fibres par câble. - `nœuds` : Liste des nœuds du réseau (une liste chaînée de `CellNoeud`). - `commodites` : Liste des commodités à relier (une liste chaînée de `CellCommodite`).		

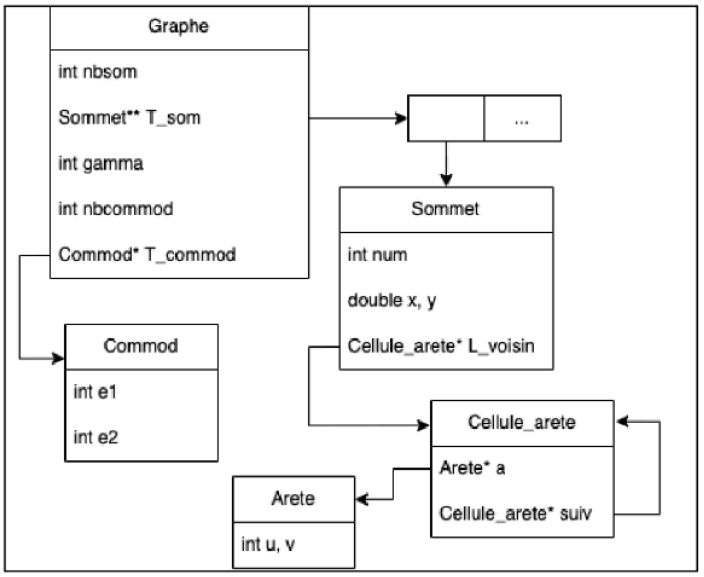
Hachage.h
TableHachage
- Structure représentant une table de hachage. - Attributs : - `nbElement` : (non nécessaire ici) Nombre d'éléments dans la table. - `tailleMax` : Taille maximale de la table. - `T` : Tableau de pointeurs vers des listes chaînées de nœuds (`CellNoeud`).

ArbreQuat.h
ArbreQuat
- Structure représentant un arbre quadtree. - Attributs : - `xc`, `yc` : Coordonnées du centre de la cellule. - `coteX` : Longueur de la cellule. - `coteY` : Hauteur de la cellule. - `nœud` : Pointeur vers le nœud du réseau. - `so` : Sous-arbre sud-ouest, pour `x < xc` et `y < yc`. - `se` : Sous-arbre sud-est, pour `x >= xc` et `y < yc`. - `no` : Sous-arbre nord-ouest, pour `x < xc` et `y >= yc`. - `ne` : Sous-arbre nord-est, pour `x >= xc` et `y >= yc`.

Graphe.h		
Arete	**Cellule_Arete**	**Sommet**
- Structure représentant une arête dans un graphe. - Attributs : - `u`, `v` : Numéros des sommets extrémités.	- Structure représentant une cellule dans une liste chaînée d'arêtes. - Attributs : - `a` : Pointeur sur l'arête. - `suiv` : Pointeur vers la cellule suivante dans la liste.	- Structure représentant un sommet dans un graphe. - Attributs : - `num` : Numéro du sommet (le même que dans `T_som`). - `x`, `y` : Coordonnées du sommet. - `L_voisin` : Liste chaînée des voisins (une liste chaînée de `Cellule_arete`).
Commod	**Graphe**	
- Structure représentant une commodité, c'est-à-dire une paire de sommets à relier. - Attributs : - `e1`, `e2` : Les deux extrémités de la commodité.	- Structure représentant un graphe. - Attributs : - `nbsom` : Nombre de sommets. - `T_som` : Tableau de pointeurs sur sommets. - `gamma` : Nombre maximal de fibres par câble. - `nbcommod` : Nombre de commodités. - `T_commod` : Tableau des commodités.	

Voici une représentation schématique des structures :





- Description global du code

Chaine.c	**creer_chaines** : Crée un nouvel ensemble de chaînes. **creer_cellChaine** : Crée une nouvelle cellule de chaîne. **creer_cellPoint** : Crée un nouveau point. **liberer_cellPoint**: Libère la mémoire allouée pour un point. **liberer_cellChaine** : Libérer la mémoire allouée pour une cellule de chaîne.6. **liberer_chaines** : Libérer la mémoire allouée pour un ensemble de chaînes. **ins_en_tete_chaine** : Insère une nouvelle chaîne au début de l'ensemble de chaînes. **ins_en_tete_point** : Insère un nouveau point au début d'une chaîne. **lectureChaines** : Lit les données à partir d'un fichier pour créer un ensemble de chaînes. **ecrireChaines** : Écrit les données des ensembles de chaînes dans un fichier. **longueurTotale** : Calcule la longueur totale de toutes les chaînes. **comptePointsTotal** : Compte le nombre total de points dans toutes les chaînes. **afficheChainesSVG** : Génère un fichier SVG représentant visuellement les chaînes. **longueurChaine** : Calcule la longueur d'une chaîne donnée. **generationAleatoire** : Génère aléatoirement des ensembles de chaînes.
Réseau.c	**creer_Noeud**: Crée un nouveau nœud avec numéro,coordonnée x et coordonnée y. **liberer_Noeud**: Libère la mémoire allouée pour un nœud, y compris ses voisins. **creer_CellNoeud**: Crée une nouvelle cellule de nœud contenant une référence à un nœud. **liberer_CellNoeud**: Libère la mémoire allouée pour une cellule de nœud. **creer_CellCommodite**: Crée une nouvelle cellule de commodité contenant des références à deux nœuds. **liberer_CellCommodite**: Libérer la mémoire allouée pour une cellule de commodité. **creer_Reseau**: Crée un nouveau réseau avec un paramètre de densité gamma. **liberer_Reseau**: Libérer la mémoire allouée pour un réseau, y compris ses nœuds et commodités. **rechercheCreeNoeudListe**: Recherche un nœud dans le réseau en fonction de ses coordonnées et le crée s'il n'existe pas. **maj_voisins**: Met à jour les voisins d'un nœud dans le réseau en ajoutant un nouveau voisin. **reconstitueReseauListe**: Reconstitue un réseau à partir d'un ensemble de chaînes en créant des nœuds et en les reliant. **nbLiaisons**: Compte le nombre total de liaisons dans le réseau. **nbCommodites**: Compte le nombre total de commodités dans le réseau. **ecrireReseau**: Écrire les informations sur les nœuds, liaisons et commodités du réseau dans un fichier. **afficheReseauSVG**: Génère un fichier SVG représentant visuellement le réseau avec ses nœuds et liaisons.

Hachage.c	**creer_TableHachage**: Crée une nouvelle table de hachage avec une taille donnée. **liberer_TableHachage**: Libérer la mémoire allouée pour une table de hachage, y compris ses cellules et leurs contenus. **clef**: Calcule la clé à partir de coordonnées x et y selon une formule donnée. **hachage**: Calcule la valeur de hachage en fonction d'une clé et de la taille de la table de hachage. **rechercheCreeNoeudHachage**: Recherche un nœud dans le réseau en utilisant une table de hachage et le crée s'il n'existe pas. **reconstitueReseauHachage**: Reconstitue un réseau à partir d'un ensemble de chaînes en utilisant une table de hachage pour optimiser la recherche et la création des nœuds.
ArbreQuat.c	**chaineCoordMinMax** : Détermine les coordonnées minimales et maximales des chaînes de points. **creerArbreQuat** : Crée un arbre quaternaire avec des paramètres spécifiés. liberer_ArbreQuat : Libère la mémoire allouée pour un arbre quaternaire. direction : Détermine la direction pour l'insertion d'un point dans un arbre quaternaire. **insererNoeudArbre** : Insérer un nœud dans un arbre quaternaire. **insererNoeudReseau** : Insérer un nœud dans un réseau. **rechercheCreeNoeudArbre** : Recherche ou crée un nœud dans un arbre quaternaire et l'associe au réseau. **reconstitueReseauArbre** : Reconstruit un réseau à partir de chaînes de points en utilisant un arbre quaternaire pour gérer les nœuds et les commodités.
Graphe.c	**creerArete** : Crée une arête entre deux sommets donnés **creerCellule_arete** : Créé une cellule contenant une arête.- **libererCellule_Arete**: Libère la mémoire allouée pour une cellule contenant une arête. **creerSommet** : Crée un sommet avec un numéro et des coordonnées spécifiés. **libererSommet** : Libérer la mémoire allouée pour un sommet. **majAretes** : Met à jour les arêtes d'un sommet dans un graphe en ajoutant les arêtes correspondant à ses voisins. **creerGraphe** : Crée un graphe à partir d'un réseau donné. **libererGraphe** : Libère la mémoire allouée pour un graphe.

III) Réponses aux questions

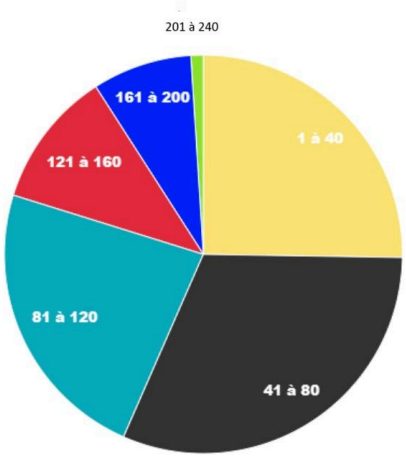
Cette partie de notre compte rendu traite la réponse aux questions du projet. La plupart de ces questions sont basées sur l'écriture du code en lui-même. Nous allons donc traiter dans ce compte rendu seulement les questions basées sur l'analyse du résultats de certaines fonctions :

- Question 4.2 : "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 ? "

Après avoir écrit une fonction main dans le fichier HachageMain.c qui prend des valeurs de 1 à 10 pour x et pour y. Nous avons constaté que les valeurs retournées par la fonction clé sont bien répartie et unique. La fonction clef nous semble donc appropriée.

Voici une capture d'écran montrant les différentes valeurs de la fonction HachageMain, nous pouvons y voir les différentes valeurs de clef.

```
x = 5.00; y = 2.00; clef = 30; hach = 5
x = 5.00; y = 3.00; clef = 39; hach = 1
x = 5.00; y = 4.00; clef = 49; hach = 2
x = 5.00; y = 5.00; clef = 60; hach = 0
x = 5.00; y = 6.00; clef = 72; hach = 4
x = 5.00; y = 7.00; clef = 85; hach = 5
x = 5.00; y = 8.00; clef = 99; hach = 1
x = 5.00; y = 9.00; clef = 114; hach = 4
x = 5.00; y = 10.00; clef = 130; hach = 3
x = 6.00; y = 1.00; clef = 29; hach = 9
x = 6.00; y = 2.00; clef = 38; hach = 4
x = 6.00; y = 3.00; clef = 48; hach = 6
x = 6.00; y = 4.00; clef = 59; hach = 4
x = 6.00; y = 5.00; clef = 71; hach = 8
x = 6.00; y = 6.00; clef = 84; hach = 9
x = 6.00; y = 7.00; clef = 98; hach = 5
x = 6.00; y = 8.00; clef = 113; hach = 8
x = 6.00; y = 9.00; clef = 129; hach = 7
x = 6.00; y = 10.00; clef = 146; hach = 2
```



Voici un graphe montrant la répartition des valeurs de clé (allant de 1 à 240). Nous pouvons remarquer que les clés sont plutôt bien répartie de 1 à 120. Les valeurs ne sont pas trop grandes donc la fonction clef semble appropriée.

IV) Description des jeux d'essais

- **Test de Chaine.c :**

Nous avons créé un main '***ChaineMain.c***' qui permet d'exécuter les fonctions de lecture et d'écriture dans un fichier et les fonctions de calculs des nombres de points totaux dans une chaîne ainsi que sa longueur. Nous avons aussi testé la fonction "***afficheChaineSVG***" qui permet de créer un fichier en HTML et qui permet de visualiser le réseau. Nous avons testé ces fonctions avec un fichier '***0001400_bruma.cha***' et nous avons créé une chaîne puis nous avons essayé la fonction '***ecrireChaine***' avec un fichier ***TestEcriture.cha***. Nous pouvons remarquer qu'il est identique à '***0001400_bruma.cha***'. Voici ce que la fonction afficheSVG affiche (voir figure 1).

Nous avons également testé la fonction '***generationAleatoire***' qui crée des chaînes aléatoires puis on les écrit dans un fichier '***testEcritureCAleatoire.cha***' en utilisant la fonction d'écriture. Voici un exemple d'une génération d'un schéma avec des chaînes aléatoires (voir figure 2).

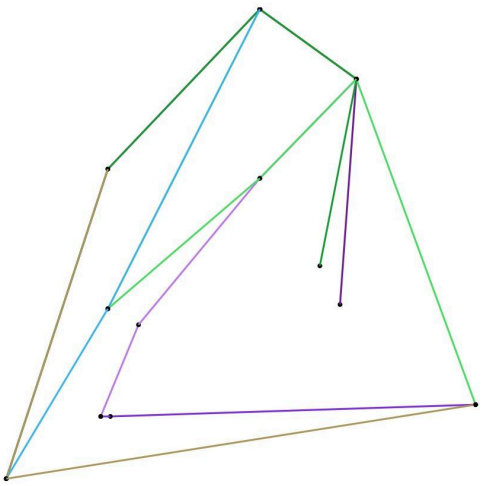


Figure 1

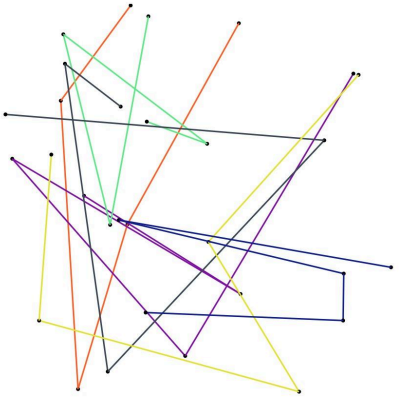


Figure 2

- **Test des fonctions de reconstitution de réseau**

Nous avons créé un main dans le fichier '***ReconstitueReseau.c***' qui utilise la ligne de commande pour prendre un fichier .cha en paramètre et un nombre entier indiquant quelle méthode l'on désire utiliser(1 pour les listes ,2 pour les tables de hachage et 3 pour les Arbres) pour reconstituer un réseau .Les résultats de test de ces fonctions sont donnés dans les fichiers ***"testEcriture<la méthode utilisée>.res"***.

Après exécution, nous pouvons voir les schéma ci dessous et les fichiers ***testEcriture<la méthode utilisée>.res*** sont identiques à ***00014_bruma.res*** :

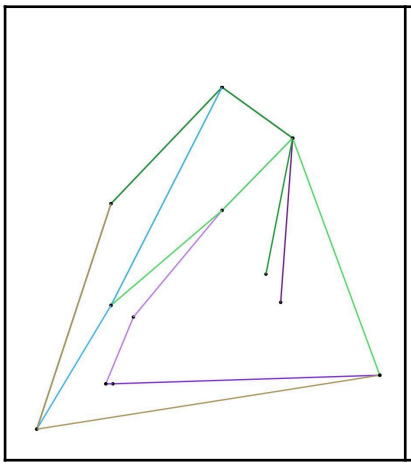


Figure 3 : Liste

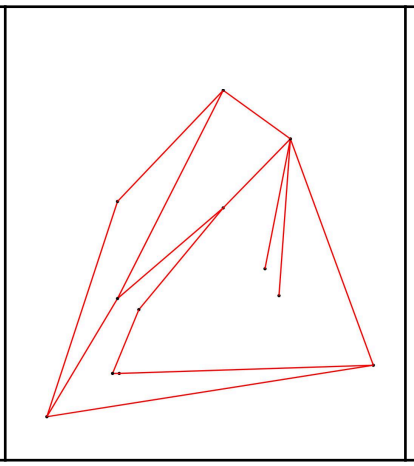


Figure 4 : Hachage

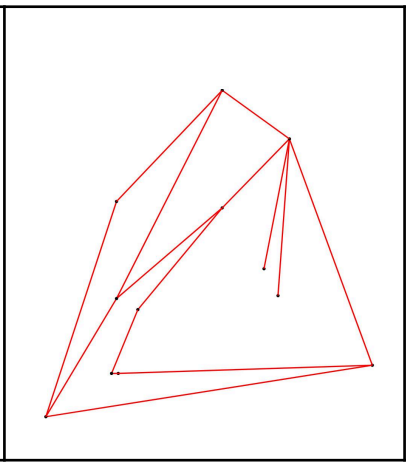


Figure 5 : Arbre

V) Analyse commentée et statistique des performances

Dans cette dernière partie, nous allons analyser et commenter les statistiques et performances des différentes méthodes : liste, hachage et arbres. Puis nous allons déterminer laquelle des 3 méthodes est la plus performantes et la plus précise.

Nous observons que le programme s'exécute très rapidement (voir trop rapidement). Il nous est impossible de déduire quelles méthodes est la plus efficace avec si peu de points.

En revanche, lorsque le nombre de points augmente, nous observons une nette différence. En effet, les méthodes basées sur une table de hachage ou un arbre quaternaire semble beaucoup plus rapide que par rapport à la méthode des listes (une liste peut prendre plusieurs heures à s'exécuter en fonction du nombre de points données, comparé à quelques secondes pour un arbre).

	500	1000	1500	2000	2500	3000	3500	4000	4500	5000	Total
Liste	31.10	180.01	486.9	971.25	1587.57	2333.44	3295.99	4427.28	5750.55	7282.66	26349.66
Hachage	0.16	0.44	0.77	1.11	1.42	1.77	2.12	2.47	2.86	3.28	16.4
Arbre	0.06	0.14	0.24	0.30	0.40	0.50	0.59	0.69	0.79	0.84	4.55

Dans le tableau ci-dessus, nous pouvons observer les différences entre les trois méthodes. Lorsque le nombre de points est faible, il est difficile de discerner la performance entre les tables de hachage et les arbres, mais il est clair que leur efficacité est supérieure à celle des listes. Cependant, à mesure que le nombre de points augmente, la différence entre

les tables de hachage et les arbres devient plus perceptible, bien que cette différence ne soit pas toujours significative. En revanche, le temps nécessaire pour manipuler une liste avec un grand nombre de points est considérablement plus long. La complexité pour la liste est en $o(n^2)$, pour une table de hachage la complexité idéale serait $o(n/m)$ avec n le nombre de points et m la taille, donc la complexité varie selon ces 2 facteurs, on peut en déduire la complexité dans le pire cas ($O(n)$) et celle dans le meilleur cas ($\Omega(1)$). L'arbre quant à lui à une complexité de $o(\log4(n))$.

Voici les différents graphes obtenu pour chacune des méthodes :

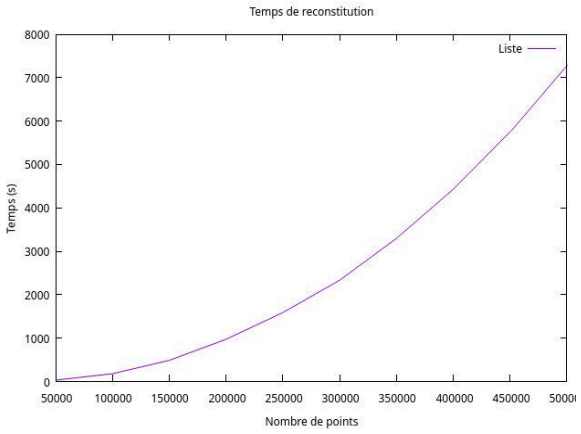


Figure 6 : temps pour les listes

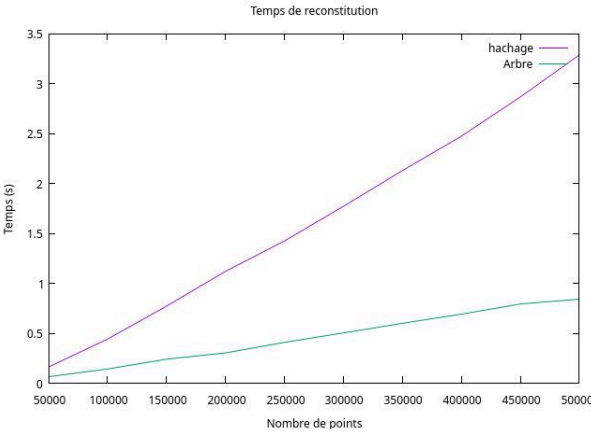


Figure 7 : temps pour hachage et arbre

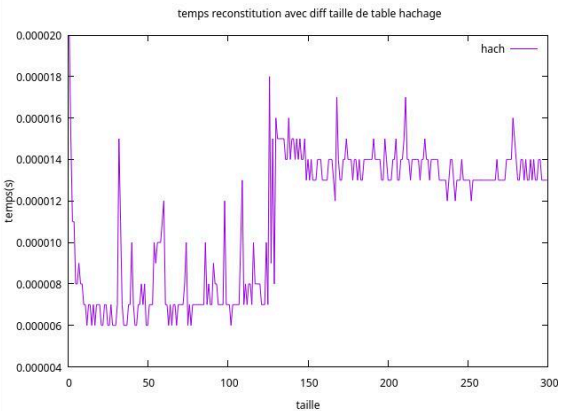


Figure 8 : temps selon la taille de la table de hachage