OULD BOUALI Louisa 21100589
BENAMER BELKACEM Inès 21204927

LU2IN006 - Rapport de projet

I - Reformulation du projet

Ce projet consiste en l'étude et la gestion d'un réseau de fibres au sein d'une ville. L'objectif est de comparer la mise en place de ce réseau avec différentes structures de données afin de l'optimiser. Ce projet se divise en plusieurs parties : d'une part la mise en place du réseau, et d'autre part l'optimisation de ce réseau. Les différentes structures de données utilisées et étudiées dans le cadre de ce projet sont les listes chaînées, les tables de hachage, et les arbres quadratiques.

II - Description des structures manipulées

Les trois structures principales manipulées ici sont les listes chaînées, les tables de hachage, et les arbres quadratiques. Voyons-en une description plus détaillée.

- Liste chaînée :

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

/* Cellule d'une liste (chaînée) de chaînes */
typedef struct cellChaine
{
 int numero;              /* Numéro de la chaîne */
 CellPoint *points;       /* Liste des points de la chaîne */
 struct cellChaine *suiv; /* Cellule suivante dans la liste */
} CellChaine;

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

Pour la structure `Chaines`, nous avons plusieurs chaînes de points qui constituent les commodités du réseau à relier. Dans chaque `CellChaine`, il y a donc une liste de points, représentée par la structure `CellPoint` qui contient les coordonnées d'un point et un pointeur vers le point suivant.

- Table de Hachage :

```
typedef struct table {
   CellNoeud** table;
   int length;
} TableHachage;
```

Nous utilisons une table de hachage avec gestion des collisions par chaînage. Notre table de hachage est donc un tableau de `CellNoeud*`. Un `CellNoeud*` est la même structure que définie précédemment, une liste chaînée de `Noeud`. Length est la taille de la table de hachage.

- Arbre quadratique :

```
typedef struct arbreQuat{
   double xc, yc;          /* Coordonnées du centre de la cellule */
   double coteX;           /* Longueur de la cellule */
   double coteY;           /* Hauteur de la cellule */
   Noeud* noeud;           /* Pointeur vers le noeud du réseau */
   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;
```

Cette structure est un arbre quadratique possédant à chaque fois 4 fils, au sud-ouest, sud-est, nord-ouest, nord-est.
Chaque nœud interne possède un centre à partir duquel on va placer les points dans les sous-arbres `so`, `se` , `no` , `ne`. Les `coteX` et `coteY` correspondent à la largeur et la hauteur de la case actuelle considérée nous permettent de déterminer les centres des sous-arbres ainsi que leurs longueur et hauteur respectives. Le champ `noeud` de chaque arbre interne est égal à NULL
Les feuilles sont des arbres pour lesquels tous les sous-arbres sont égaux à NULL, car on ne peut pas aller "plus profondément" dans la recherche, on est arrivés à l'extrémité de l'arbre une fois dans une feuille. Les feuilles sont les parties de l'arbre qui contiennent les nœuds, ainsi leurs nœuds ne sont pas NULL.

III - Description des fichiers .h et .c

Tous les fichiers .h déclarent les structures et fonctions utilisées par les fichiers du même nom. Nous décrirons ici uniquement les fonctions principales les plus importantes. Une brève description des fonctions secondaires est disponible dans les fichiers .h, en commentaire des fonctions.

- **Chaine.c** :

```
Chaines* generationAleatoire(int nbChaines, int nbPointsChaine, int xmax, int
ymax);
```

Cette fonction nous permet de générer aléatoirement une structure de type Chaines* selon un certain nombre de paramètres : le nombre de chaînes que l'on souhaite créer, le nombre de points contenu dans chaque chaîne, ainsi que les coordonnées maximum des points à générer.
C'est une fonction assez simple dans son implémentation. On crée un point aux coordonnées aléatoires entre (0, 0) et (xmax, ymax), et on répète cette étape nbPointsChaine fois afin d'obtenir une chaîne. On répète ensuite ceci nbChaines fois afin d'obtenir la structure Chaines* désirée.

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

Ces deux fonctions nous permettent l'interaction entre structure Chaine* et fichiers. Elles nous permettent de passer de l'un à l'autre par la lecture ou l'écriture de fichier, et leur implémentation suit l'architecture d'une Chaines* dans son implémentation.

```
int comptePointsTotal(Chaines *C);
```

Cette fonction nous permet d'obtenir le nombre total de points contenus dans une Chaines*, ce qui peut être utile pour avoir une meilleure idée du réseau.

```
void libere_point(CellPoint *point);
void liberer_liste_points(CellPoint *liste);
void librer_chaine(CellChaine *chaine);
void liberer_liste_chaines(CellChaine *liste);
void liberer_structure(Chaines *structt);
```

Ces fonctions s'agissent des fonctions de libération de mémoire, indispensables afin d'éviter les fuites mémoire et pour assurer un code propre.

- **Reseau.c :**

```
Noeud* creeNoeud(int num, int x, int y);
CellCommodite* creeCommodite(Noeud* a, Noeud* b);
Reseau* creeReseau(int gamma);
CellNoeud *creeCellNoeud(Noeud *noeud);
```

Les fonctions de création sont la base de la constitution de réseau, car elles permettent de déclarer et d'allouer la mémoire pour les structures de bases qui vont nous permettre de manipuler notre réseau.

```
void ajouteVoisins(Noeud* n1, Noeud* n2);
```

Cette fonction est assez utile bien que simple, car elle permet de rendre les deux noeuds passés en paramètres voisins, en créant et ajoutant les CellNoeud associés à chacun des noeuds à la liste des voisins de l'autre.

```
Noeud* rechercheCreeNoeudListe(Reseau *R, double x, double y);
```

Cette fonction nous permet de trouver (potentiellement en le créant) un noeud dans un réseau, et de le renvoyer. Pour ce faire, la fonction va parcourir le réseau afin de déterminer si un noeud correspondant aux paramètres existe déjà dans le réseau. Dès la rencontre d'un tel noeud, ce dernier est renvoyé.
Dans le cas contraire, un noeud correspondant aux coordonnées recherchées est créé et ajouté dans le réseau.

```
void rechercheCreeCellCommodite(Reseau* R, Noeud* extrA, Noeud* extrB);
```

Cette fonction possède le même principe et champ d'application que la précédente. Elle cherche une commodité d'extrémités extrA et extrA dans un réseau donné, renvoie une telle commodité si elle existe, et la crée puis renvoie si l'extrémité n'existe pas.

```
void libererReseau(Reseau *reseau);
void libererCellNoeuds(CellNoeud *cells, int rm);
void libereNoeud(Noeud *noeud);
void liberercommodites(CellCommodite *commodites);
```

Comme pour les chaînes et dans le même but, nous avons mis en place des fonctions de libération de mémoire afin d'éviter les fuites et de permettre de terminer un programme en vérifiant bien que tous les emplacements mémoire alloués soient libérés.

```
Reseau* reconstitueReseauListe(Chaines *C);
```

Cette fonction est la première grosse fonction qui va nous permettre de compléter notre étude. Elle nous permet de reconstituer un réseau à partir d'une structure Chaines.

Initialisation du réseau :
La fonction commence par créer un réseau en utilisant la fonction creeReseau, avec pour paramètre C->gamma Ce réseau sera utilisé pour stocker les nœuds et les liaisons entre eux, et c'est le résultat qui sera retourné.

Parcours des chaînes de points :
En utilisant une boucle for, la fonction parcourt chaque chaîne de points dans la liste de chaînes C->chaines.
À chaque itération, une variable chaine est utilisée pour représenter une chaîne de points.

Traitement des points dans chaque chaîne :

Pour chaque chaîne de points, une boucle for parcourt chaque point de la chaîne.
Pour chaque point, on fait appel à la fonction rechercheCreeNoeudListe qui cherche l'existence du point de même coordonnées dans le réseau. Si le nœud existe déjà, il est simplement récupéré. Sinon, un nouveau nœud est créé et ajouté au réseau.

Création des liens entre les nœuds :
Après la création ou la récupération du nœud pour chaque point, la fonction vérifie s'il existe déjà un lien entre ce nœud et le nœud précédent dans la chaîne.
Si aucun lien n'existe, la fonction crée deux cellules de nœud : une pour le nœud actuel et une pour le nœud précédent, et les ajoute à la liste des voisins de chaque nœud. Cela établit ainsi une liaison bidirectionnelle entre les deux nœuds.

Création des commodités :
Une fois que tous les points d'une chaîne ont été traités, la fonction vérifie si un premier nœud a été défini. Si oui, cela signifie que la chaîne a au moins deux points.
Dans ce cas, une commodité est créée entre le premier et le dernier nœud de la chaîne à l'aide de la fonction creeCommodite. Cette commodité est ensuite ajoutée en tête de la liste des commodités du réseau, ajoutant effectivement la commodité au réseau.

Gestion des erreurs :
À chaque étape où une allocation de mémoire est effectuée (pour les nœuds, les liens ou les commodités), la fonction vérifie si l'allocation a réussi. Si une allocation échoue, la fonction libère la mémoire allouée pour le réseau jusqu'à présent et retourne NULL.

Retour du réseau reconstitué :
Une fois toutes les chaînes de points traitées, la fonction retourne le réseau reconstitué, contenant tous les nœuds, les liens entre eux et les commodités établies.

- **Hachage.c :**

```
double cle(double x,double y);
int hachage(double cle, int lenght);
```

Ces deux fonctions, qui sont utiles main dans la main, permettent à terme d'insérer un point dans un tableau. A partir des coordonnées d'un point, la fonction cle renvoie une clé associée à ces coordonnées. Cette clé est ensuite donnée à la fonction de hachage qui va utiliser la clé ainsi que la taille de la table afin de déterminer l'indice de la case du tableau dans laquelle il faudra insérer l'élément souhaité.

```
Noeud* rechercheCreeNoeudHachage(Reseau* R, TableHachage* H, double x, double y);
```

De même que dans la partie précédente, cette fonction va chercher un noeud dans un réseau, puis le renvoyer, et le créer s'il n'existe pas déjà.
Son fonctionnement est cependant différent de l'implémentation avec une liste chaînée. En passant par une table de hachage, on a alors la possibilité de non pas parcourir toute la chaîne jusqu'à trouver le nœud correspondant, mais de chercher immédiatement dans la case correspondante. Grâce aux fonctions cle et hachage, on détermine dans quelle case

devrait se trouver le nœud s'il existait, puis on cherche à l'intérieur de la liste chaînée de cette case à l'aide d'une boucle while.
Si le nœud n'est pas trouvé, on effectue les opérations nécessaires afin d'allouer, de définir et d'insérer le nœud dans le réseau.

```
Reseau* reconstitueReseauHachage(Chaines *C, int M);
```

Cette fonction nous permet de reconstituer un réseau à partir d'une Chaines* et d'une taille M de table de hachage.

Comme dans reconstitueReseauListe, on va parcourir chaque chaîne puis chaque point de la chaîne. Pour chaque point rencontré, on va chercher s'il existe un noeud correspondant dans le réseau, et le créer si ce n'est pas le cas à l'aide de la fonction rechercheCreeNoeudHachage. A chaque passage à un nouveau point, on ajoute les points adjacent à sa liste de voisins si il n'y appartiennent pas déjà (à l'aide de la fonction rendreVoisins). On retient également le premier et dernier point de chaque chaîne, et on va créer un commodité entre les deux si ils ne sont pas NULL, ne sont pas égaux, et s'il n'existe pas déjà une commodité entre les deux.
On répète ces étapes pour chaque chaîne de C, on obtient ainsi le réseau souhaité.

- **ArbreQuat.c :**

```
void chaineCoordMinMax(Chaines* C, double* xmin, double* ymin, double* xmax,
double* ymax)
```

Cette fonction nous permet de déterminer les coordonnées extrêmes d'une Chaines, et ce en la parcourant.

```
void insererNoeudArbre(Noeud *n, ArbreQuat **a, ArbreQuat *parent);
```

Cette fonction nous permet d'insérer un noeud dans un arbre. ArbreQuat** est un pointeur vers un ArbreQuat*, ce qui va nous permettre de modifier cet arbre à l'intérieur de la fonction. L'argument parent nous servira à initialiser les coordonnées et cotés de l'arbre si on doit en créer un.
On distingue plusieurs cas.
Insertion dans un arbre qui est NULL. On a alors besoin de créer un nouvel arbre dans lequel on insère le noeud, en utilisant la fonction creerArbreQuat. On instancie ensuite le noeud de cet arbre au noeud passé en paramètre, car on crée une feuille.
Insertion dans une feuille. Il existe déjà une feuille à cet endroit, il faut donc qu'on découpe l'arbre en 4 sous-arbres, puis qu'on insère chacun des deux noeuds dans le sous-arbre correspondant (le noeud déjà existant et le noeud qu'on souhaite insérer). On instancie ensuite le noeud de ce noeud à NULL afin d'indiquer qu'il s'agit d'un noeud interne et non plus une feuille.
Insertion dans un noeud interne. On détermine dans quel sous-arbre de l'arbre il faut insérer le noeud, puis on l'insère récursivement à cet endroit.

```
Noeud* rechercheCreeNoeudArbre(Reseau* R, ArbreQuat** a, ArbreQuat* parent, double x, double y);
```

Cette fonction reprend la logique de la fonction analogue pour les tables de hachage. On recherche un nœud dans l'arbre et on le renvoie, en le créant s'il n'existe pas.
Si on se trouve au niveau d'un arbre NULL, on n'a pas trouvé le nœud donc on le crée, on l'ajoute dans l'arbre (avec insererNoeudReseau) et le réseau, et on le renvoie.
Si on se trouve au niveau d'une feuille : si le nœud de la feuille correspond aux coordonnées en paramètre, on le renvoie, sinon on crée un nœud, on l'ajoute à l'arbre (avec insererNoeudReseau) et au réseau, puis on le renvoie.
Si on se trouve au niveau d'un nœud interne, on cherche la position du point à insérer par rapport au centre de l'arbre, puis on continue récursivement la recherche dans cet arbre.

```
Reseau* ReconstitueReseauArbre(Chaines* C);
```

Cette fonction nous permet d'obtenir un réseau à partir d'une Chaines*, en utilisant un arbre quadratique.
On crée tout d'abord un arbre avec les données sur le centre, la longueur et la largeur récupérées depuis la Chaines* C grâce à la fonction chaineCoordMinMax.
On parcourt toutes les chaînes de C. Pour chaque chaîne, on parcourt sa liste de points. On stocke le premier et dernier point dans des variables afin d'éventuellement créer une commodité correspondante si nécessaire. Pour chaque point parcouru, on cherche s'il existe dans l'arbre et le réseau à l'aide de rechercheCreeNoeudArbre, qui va le créer si nécessaire. Pour chaque point toujours, on ajoute ce point à la liste des voisins de son précédent, et inversement s'ils ne sont pas déjà voisins.

- **Graphe.c :**

```
Graphe* creerGraphe(Reseau* r);
```

La fonction creerGraphe crée un graphe à partir d'un réseau donné. Elle vérifie d'abord si le réseau existe. Ensuite, elle alloue de la mémoire pour la structure Graphe et initialise ses variables avec les valeurs correspondantes du réseau. Enfin, elle alloue de la mémoire pour deux tableaux, un pour les sommets du graphe et un pour les commodités, en vérifiant à chaque étape si l'allocation de mémoire réussit.

```
int plus_petit_nb_aretes(Graphe *graphe, int u, int v);
```

La fonction plus_petit_nb_aretes calcule le plus petit nombre d'arêtes nécessaires pour aller d'un sommet u à un sommet v.
Elle vérifie d'abord si le graphe est valide et si les numéros de sommets u et v sont dans la plage valide des indices du tableau de sommets.
Ensuite, elle crée deux tableaux, l'un pour suivre les sommets visités et l'autre pour stocker les distances minimales entre les sommets.

Elle parcourt ensuite les sommets en utilisant une file pour le parcours en largeur. À chaque itération, elle explore les voisins du sommet en cours et met à jour les distances si elle trouve un chemin plus court.
Finalement, elle renvoie la distance minimale entre les sommets u et v, puis libère la mémoire allouée pour les tableaux et la file.

```
void MiseAjourAretes(Graphe *g, Sommet *s, Noeud *n);
```

Cette fonction met à jour les arêtes d'un sommet en fonction de ses voisins. Elle parcourt les voisins d'un nœud donné et crée des arêtes entre ce nœud et ses voisins dans le graphe, si elles n'existent pas déjà.

```
int reorganise_reseau(Reseau *reseau);
```

Cette fonction restructure le réseau en vérifiant si le nombre d'apparitions de chaque arête dans les chemins entre les commodités est inférieur à une valeur spécifiée (gamma). Elle crée d'abord un graphe à partir du réseau donné, puis une matrice pour suivre les apparitions des arêtes dans les chemins. Ensuite, elle génère la plus courte chaîne entre chaque paire de commodités et met à jour la matrice en conséquence. Si le nombre d'apparitions dépasse gamma pour une arête donnée, la fonction retourne faux (0), sinon elle retourne vrai (1).

```
void generate_plus_petit_chaine(Graphe *graphe, int u, int v, ListeEntier *liste);
```

Cette fonction génère la plus courte chaîne entre deux sommets dans un graphe donné en utilisant un parcours en largeur. Elle utilise une file pour suivre les sommets à visiter et un tableau pour suivre les prédécesseurs de chaque sommet sur le chemin le plus court.

- **main_chaines.c :**
  Ce main nous permet de tester les fonctions du fichier Chaine.c en lisant un fichier puis en écrivant un autre fichier qui contient les mêmes chaînes.

- **main_reseau.c :**
  Grâce à ce programme, nous pouvons tester les différentes fonctions de reconstitution de réseau en demandant à l'utilisateur la structure qu'il veut tester, puis en créant un fichier correspondant

- **compare_all :**
  Ce programme permet de lire un fichier grâce à la ligne de commande, puis calcule et écrit les temps de reconstitution du réseau avec les différentes structures dans un fichier 'comparaison_temps.txt'

- **compare_all_alea.c :**

Ce main permet de générer aléatoirement un grand nombre de chaînes et de points, avec un nombre de chaînes variable, puis détermine les temps de reconstitution du réseau à partir des différentes structures. Il écrit ensuite les résultats dans un fichier 'compare_temps_alea.txt'

IV - Réponses aux questions

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 ?

Pour répondre à cette question, nous avons créé un petit main qui teste différentes valeurs comme suggéré. Nous avons donc obtenu 100 valeurs de clé qui semblaient parfaitement réparties entre 0 et 100. Il n'y avait pas de répétition (2 couples qui avaient la même clé) et la répartition des valeurs semblait assez aléatoire. Nous en déduisons donc que cette fonction de génération de clé semble satisfaisante car elle maximise le nombre de clé possibles à obtenir et minimise les collisions.

Exercice 6 : Comparaison des structures

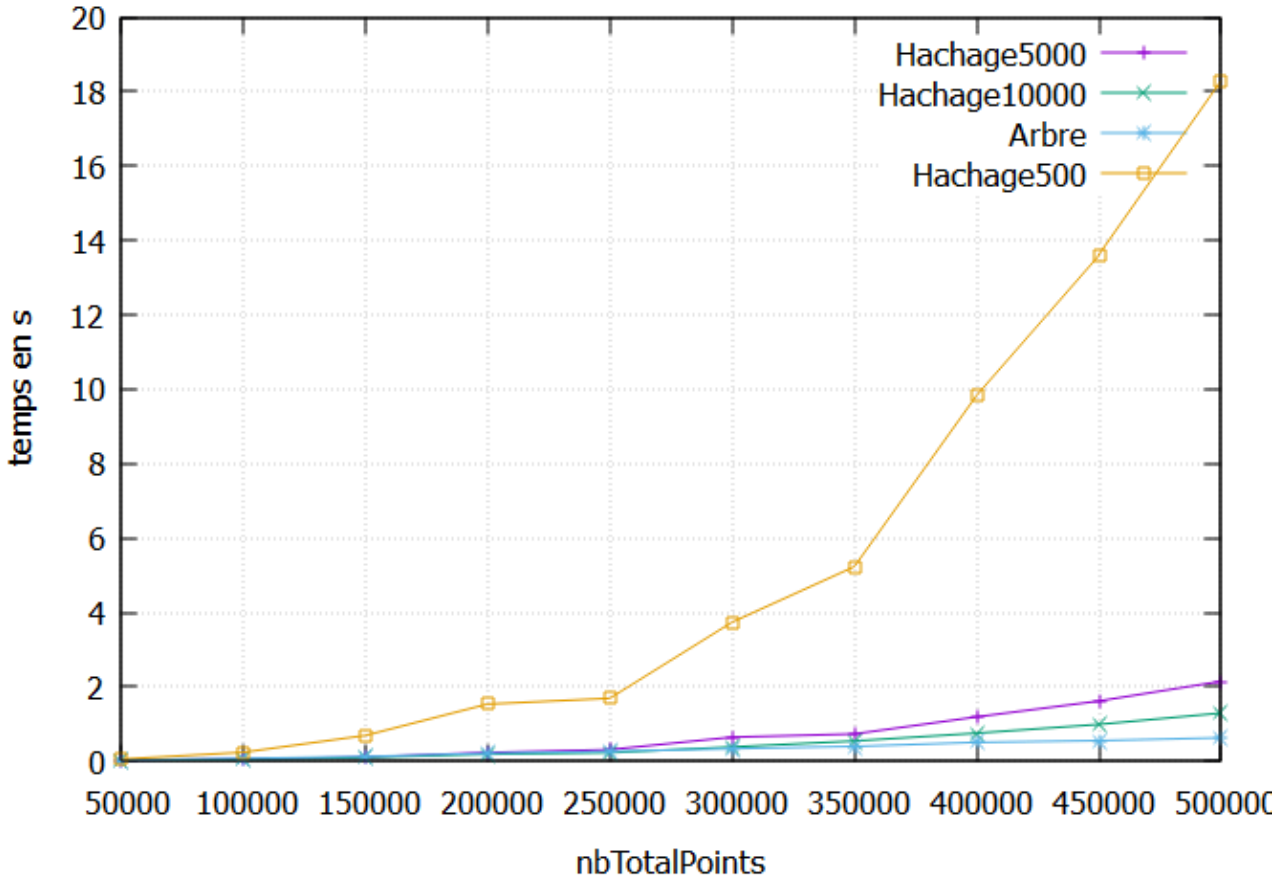
Dans cette partie, nous comparons les temps de calcul des trois structures afin de trouver la structure la plus appropriée.
Il s'agit donc de comparer liste chaînée, table de hachage et arbre quadratique.
Pour la table de hachage, nous avons choisi de faire varier la taille du tableau en commençant à une taille de 500 et en augmentant la taille de 500 à chaque fois.

complexité de la fonction rechercheCreeNoeudListe avec n le nombre de points du réseau : $O(n)$ et $\Omega(n)$.
On n'a pas d'autre moyen de trouver le point que de parcourir toute la chaîne, et l'emplacement est indéterminé on n'a pas de moyen de rendre le calcul plus rapide. Le meilleur cas est si le nœud recherché est le premier de la liste, et le pire cas est si le nœud n'existe pas, auquel cas on doit parcourir toute la liste (n valeurs) puis créer le nœud.

complexité de la fonction rechercheCreeNoeudHachage avec n le nombre de points du réseau et m la taille de la table de hachage = $\Theta(n/m)$.
Bien que la complexité pire cas pour la table de hachage soit de $O(n)$, et la complexité meilleur cas de $\Omega(1)$, le pire cas est un cas va très rarement être atteint. En effet, pour que la complexité réelle soit du même ordre que $O(n)$, il faudrait que tous les points soient dans la même case du tableau. Cependant, comme vu précédemment, nous avons une fonction de hachage qui est plutôt efficace et répartit bien les valeurs de clé retournées. Donc la plupart du temps, les valeurs seront assez bien réparties et on aura plutôt une complexité en $\Theta(n/m)$, avec n le nombre de points total et m la taille de la table de hachage.

complexité de la fonction rechercheCreeNoeudArbre avec n le nombre de points du réseau : $\Theta(\log_4(n))$.

Cette fonction est la fonction la plus efficace car elle ne nécessite pas de parcours de liste chaînée pour la recherche, contrairement à la liste chaînée et à la gestion des collisions pour la table de hachage. On fait uniquement des comparaisons de coordonnées.



Le temps de calcul de la fonction rechercheCreeNoeud Hachage se rapproche du temps de calcul pour l'arbre, lorsque la taille de la table est proche du nombre de points. En effet, on a alors le calcul de l'indice en O(1), et l'accès au nœud recherché et O(1) également car il y aura très peu voire pas de collisions.

Exercice 7 : Optimisation du réseau

Pour les fichier fournie càd 00014_burma.cha, 05000_USA-road-d-NY.cha et 07397_pla.cha : On a toujours un résultat de 0 donc les réseaux correspondants ne sont pas bien organisés

Afin d'améliorer la fonction, on peut ajouter un champ nb_passages à la structure Arete *(Vu que on ne duplique pas les arrête, on duplique juste les Cellule_arete, donc on a toujours le même pointeur) qu'on initialise à 0. Cet attribut comptera le nombre de fois où on est passés par l'arête dans les plus petites commodités.

On modifie également la fonction generate_plus_petit_chaine changera en ìnt generate_plus_petit_chaine(Graphe *graphe, int u, int v, ListeEntier *liste).

Cette fonction genère la liste d'arêtes parcourues et actualise le champs nb_passages et retourne le maximum de nb_passages pour chaque arête parcourue.

Il nous reste alors à tester, à chaque passage, si le nb_passages est supérieur à gamma. Si c’est le cas, on ne continue pas car on aura déjà atteint le maximum.