Projet : Réorganisation d'un réseau de fibres optiques

Nous considérons dans ce projet la réorganisation d'un réseau de fibres optiques d'une agglomération. L'énoncé de ce projet se subdivise en deux parties :

— Partie 1 : Reconstitution du réseau.
— Partie 2 : Réorganisation du réseau.

Ces deux parties sont divisées en exercices, qui vont vous permettre de concevoir progressivement le programme final. Il est conseillé de suivre les étapes données par ces différents exercices, car chaque exercice est l'application de notions introduites en cours et en TD en parallèle au projet. Par ailleurs, il est impératif de travailler régulièrement afin de ne pas prendre de retard et de pouvoir profiter des séances correspondantes pour chaque partie du projet.

Vous avez jusqu'à la fin du semestre (séances 6 à 11) pour mener à bien ce projet. Bien qu'il ne soit pas noté, nous vous recommandons de le travailler sérieusement : il constitue un excellent entraînement pour le TME solo final. Le projet devra être soumis sur Moodle avant la fin du semestre (la date exacte sera précisée ultérieurement) et pourra, le cas échéant, être pris en compte lors du jury de fin de semestre.

Cadre du projet

Dans ce projet, nous considérons une agglomération dont les services municipaux désirent améliorer le réseau de fibres optiques de ses administrés. Un *réseau* est un ensemble de *câbles*, chacun contenant un ensemble de *fibres optiques* et reliant des *clients*.

La première partie du projet consiste à reconstituer le plan du réseau de l'agglomération. En effet, plusieurs opérateurs se partagent actuellement le marché et possèdent chacun quelques fibres du réseau. Le réseau ayant régulièrement grossi, il n'existe pas à ce jour de plan complet du réseau. En revanche, chaque opérateur connaît les tronçons de fibres optiques qu'il utilise dans le réseau. En partant de l'hypothèse qu'il y a au moins une fibre optique utilisée par câble, il est ainsi possible de reconstituer le réseau dans son intégralité.

Une deuxième partie du travail va consister à réorganiser les attributions de fibres de chacun des opérateurs. En effet, la répartition des fibres n'ayant jamais été remise en cause, certains câbles sont sous-exploités alors que d'autres sont sur-exploités. Chaque opérateur possède une liste de paires de clients qu'il a reliés l'un à l'autre par une chaîne de tronçons de fibres optiques, suivant les disponibilités des fibres. Certaines chaînes sont donc très longues. Ces problèmes de sur-exploitation et de longueurs excessives peuvent être résolus, ou tout du moins améliorés, en réorganisant le réseau et en attribuant aux opérateurs des chaînes moins longues et mieux réparties dans le réseau : ce sera l'objet de la seconde partie du projet.

L'objectif de ce projet est de proposer à l'agglomération les meilleures méthodes possibles pour réaliser ces deux parties, et donc nous allons donc tester plusieurs algorithmes.

Modélisation et notations

Un *câble* du réseau est un fourreau (ou une gaine) contenant exactement $\gamma > 0$ fibres optiques. Les câbles relient deux points du plan. Dans le réseau, il existe deux types de points :

— un *client* qui peut représenter une entreprise cliente ou encore un local technique de l'opérateur.

— un *concentrateur* permettant de relier des tronçons de fibres optiques (pour former des chaînes).

Plus précisément, les tronçons de fibres optiques de deux câbles qui arrivent à un même concentrateur peuvent être reliés à ce point. Les tronçons de fibres optiques ainsi reliés bout à bout forment alors des *chaînes* dans le réseau. Une chaîne relie toujours deux points clients : on appelle ce couple de points *une commodité* (ce sont les extrémités de la chaîne). Il existe plusieurs opérateurs dans l'agglomération et chaque opérateur possède plusieurs chaînes de fibres optiques. Par ailleurs, un point du réseau peut être un client, un concentrateur ou les deux à la fois.

Un exemple

La figure 1 représente une instance de notre problème.

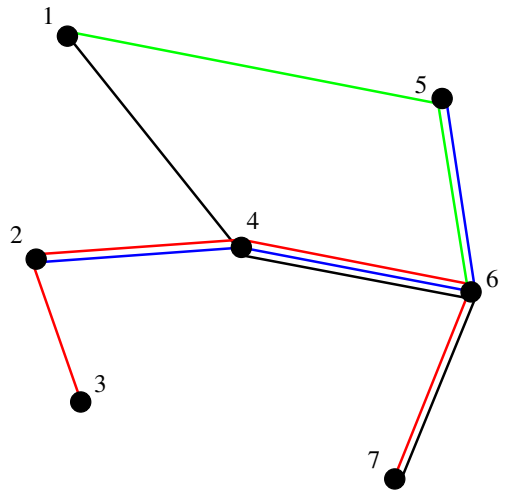

FIGURE 1 – Un exemple de réseau.

Elle décrit un réseau composé de 7 points et 4 chaînes représentées chacune par une couleur : une chaîne $(1, 4, 6, 7)$ reliant la commodité $(1, 7)$, une chaîne $(2, 4, 6, 5)$ reliant la commodité $(2, 5)$, une chaîne $(3, 2, 4, 6, 7)$ reliant la commodité $(3, 7)$ et la chaîne $(1, 5, 6)$ reliant la commodité $(1, 6)$. On peut remarquer que :

— les points 1, 3 et 7 sont uniquement clients car ils sont au bout d'une chaîne sans être un point intérieur d'une chaîne.

— le point 4 est uniquement un concentrateur car c'est toujours un point intérieur d'une chaîne.

— les autres points sont à la fois des clients et des concentrateurs.

Si l'on regarde non plus la liste des chaînes, mais le réseau dans sa globalité, on peut noter qu'il existe seulement 7 câbles dans ce réseau (ce sont les arêtes du graphe). Par exemple, dans le câble $(1, 4)$, une seule fibre est utilisée, alors que dans le câble $(4, 6)$, trois fibres sont utilisées. Notez que ce dessin ne

donne pas l'indication sur γ (le nombre maximal de fibres utilisables par câble), mais on peut déduire que $\gamma \geq 3$.

Instances des problèmes

Les instances que nous allons manipuler dans ce projet sont soit issues de la base TSPLib [1], soit issues de la base du 9ème challenge DIMACS [2]. Ces deux bases contiennent des instances de réseaux fréquemment utilisées pour tester l'efficacité d'algorithmes concernant les problèmes de réseaux. La plupart correspondent à des villes ou des pays (Burma=Birmanie, NY=New-York, d=Allemagne, att=USA, etc.), d'autres proviennent de réseaux non géographiques (réseaux électroniques, etc.). Ces instances ont été adaptées afin d'être utilisables pour ce projet. Sur le site, vous les trouverez sous la forme de fichiers-texte classés selon leur nombre de points. Il est demandé dans le projet d'utiliser ces instances pour tester vos algorithmes.

Lecture, stockage et affichage des données

Exercice 1 – Manipulation d'une instance de "Liste de Chaînes" (TME6)

Dans ce premier exercice, nous allons construire une bibliothèque de manipulation d'instances : lecture et écriture de fichier, affichage graphique de réseaux, calcul de la longueur totale des chaînes, et calcul du nombre de points.

Une instance de "Liste de Chaînes" est simplement donnée par un nombre de chaînes et par la liste des chaînes. Chaque chaîne est une liste de points du plan. Chaque point est repéré par ses coordonnées (abscisse x et ordonnée y). Chaque instance est donnée par un fichier texte d'extension `.cha` qui respecte le format donné par l'exemple `00014_burma.cha` suivant :

```
NbChain: 8
Gamma: 3
0 3 25.23 97.24 14.05 98.12 16.47 94.44
1 3 14.05 98.12 16.47 96.1 20.09 92.54
2 3 16.3 97.38 16.53 97.38 25.23 97.24
3 4 16.47 96.1 20.09 94.55 22.39 93.37 25.23 97.24
4 4 22.39 93.37 20.09 94.55 17.2 96.29 16.3 97.38
5 5 14.05 98.12 16.47 94.44 20.09 92.54 22.39 93.37 21.52 95.59
6 5 14.05 98.12 16.47 94.44 20.09 92.54 22.39 93.37 22 96.05
7 3 22.39 93.37 20.09 92.54 16.47 96.1
```

Les deux premières lignes donnent le nombre de chaînes et le nombre maximal γ de fibres optiques par câble. Les différentes chaînes du réseau sont ensuite données. Chaque ligne de chaîne comporte dans l'ordre, le numéro de la chaîne, le nombre de points de la chaîne et la liste des points. Chaque point est donné par ses coordonnées (abscisse et ordonnée). Chaque chaîne peut donc être vue comme une liste chaînée de points. On utilisera la structure de données suivante dans le fichier (fourni) `Chaine.h`.

1. http ://comopt.ifi.uni-heidelberg.de/software/TSPLIB95
2. http ://www.dis.uniroma1.it/challenge9/

```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);
void ecrireChaines(Chaines *C, FILE *f);
void afficheChainesSVG(Chaines *C, char* nomInstance);
double longueurTotale(Chaines *C);
int comptePointsTotal(Chaines *C);

#endif
```

On peut remarquer que :

- le struct `cellPoint` est un élément de la liste des points et contient les coordonnées d'un point.
- le struct `cellChaine` est un élément de la liste des chaînes et contient un numéro de chaîne et la liste des points.
- l'ensemble des chaînes est un struct contenant le nombre maximal de fibres par câble, le nombre de chaînes et la liste des chaînes.

Q 1.1 Dans un fichier `Chaine.c`, implémenter une fonction `Chaines* lectureChaine(FILE *f);` qui permet d'allouer, de remplir et de retourner une instance de notre structure à partir d'un fichier.

Q 1.2 Implémenter une fonction `void ecrireChaine(Chaines *C, FILE *f);` qui écrit dans un fichier le contenu d'une `Chaines` en respectant le même format que celui contenu dans le fichier d'origine. Créer un main `ChaineMain.c` permettant d'exécuter les fonctions de lecture et d'écriture que vous venez de définir (vous pouvez utiliser la ligne de commande pour passer le nom du fichier contenant l'instance).

Remarques :

— La fonction d'écriture permet de recréer le fichier de données, mais l'ordre des points et des chaînes sera inversé (à cause des insertions en tête de liste).

— Le but de cette fonction d'écriture est de tester le code de votre fonction de lecture sur plusieurs instances, avant d'attaquer la suite du projet.

Q 1.3 On désire donner une représentation graphique des instances. Pour cela, nous allons utiliser le format d'images SVG (Scalable Vector Graphics) qui est de plus en plus employé pour décrire des

graphiques simples et qui est très utilisé pour internet. Votre code va créer un fichier au format SVG pour html qui sera ainsi lu directement par votre explorateur internet préféré. Nous vous proposons sur moodle une petite librairie C très très simple qui crée un fichier SVG avec extension html. Il s'agit d'un struct `SVGwriter` qui est manipulé par des méthodes permettant de créer le fichier, ajouter des lignes et des points et changer de couleurs. Il y a également une génération aléatoire de couleur de segments (il faut initialiser la génération aléatoire pour obtenir des couleurs diverses). Dans votre fichier `Chaine.c`, ajouter la fonction `void afficheChaineSVG(Chaines *C, char* nomInstance);` qui permet de créer le fichier SVG en html à partir d'un struct `Chaines`. Cette fonction vous est donné dans le fichier "affichageSVG.txt" sur moodle. Tester cette fonction d'affichage dans votre fichier `ChaineMain.c`.

Q 1.4 Implémenter les fonctions :

- `double longueurChaine(CellChaine *c);` qui calcule la longueur physique d'une chaîne.
- `double longueurTotale(Chaines *C);` qui calcule la longueur physique totale des chaînes.

Pour calculer la longueur d'une chaîne, il faut sommer les distances entre les différents points qui composent la chaîne. Pour rappel, la distance entre entre deux points A et B de coordonnées (x_A, y_A) et (x_B, y_B) est donnée par $d(A, B) = \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}$.

Remarque : avec la bibliothèque `<math.h>`, pensez à ajouter l'option `-lm` lors de la création de votre exécutable.

Q 1.5 Écrire la fonction `int comptePointsTotal(Chaines *C);` qui donne le nombre total d'occurrences de points (les points qui apparaissent plusieurs fois sont comptés plusieurs fois).

Reconstitution du réseau

Le but de cette partie est de reconstituer efficacement le réseau à partir des chaînes. À partir de la liste des chaînes, il s'agit de :

- trouver la liste des nœuds du réseau (on élimine les redondances de points). Ainsi, à une coordonnée donnée, il ne peut y avoir qu'un seul nœud.
- identifier tous les câbles qui sont issus d'un nœud. Ceci permet de conserver la liste des nœuds voisins à un nœud donné.
- de récupérer et conserver la liste des commodités du réseau.

L'algorithme de reconstitution est très simple. Le pseudo-code est le suivant :

```
On utilise un ensemble de nœuds V qui est initialisé vide: V ← ∅
On parcourt une à une chaque chaîne:
    Pour chaque point p de la chaîne:
        Si p ∉ V (on teste si le point n'a pas déjà été rencontré auparavant)
            On ajoute dans V un nœud correspond au point p.
        On met à jour la liste des voisins de p et celles de ses voisins.
    On conserve la commodité de la chaîne.
```

Dans cette partie, on s'intéresse à l'optimisation du test "$p \notin V$". Pour cela, on va étudier trois méthodes qui correspondent à trois structures de données pour implémenter l'ensemble V : une liste chaînée, une table de hachage et des arbres.

Exercice 2 – Première méthode : stockage par liste chaînée (TME6-TME7)

Dans cet exercice, on désire implémenter l'algorithme de reconstitution de réseau en codant l'ensemble des nœuds du réseau par une liste chaînée. Pour cela, nous avons besoin de définir une structure pour manipuler un réseau. Un réseau se présente comme un ensemble de nœuds, de câbles et de commodités. Dans cette structure :

- Chaque nœud v sera repéré par ses coordonnées, et on connaîtra la liste des pointeurs sur nœuds qui sont reliés à v par un câble. Lors de la reconstitution du réseau, on attribuera à chaque nœud un numéro entier unique qu'on lui choisira incrémentalement.
- Chaque câble est donné par des pointeurs sur ses deux nœuds extrémités.
- Chaque commodité est une paire de pointeurs sur les nœuds du réseau qui doivent être reliés par une chaîne.

Ainsi, pour stocker les données du réseau, on utilisera la structure de données suivante (fichier fourni), définie dans le fichier `Reseau.h` :

```
1  #ifndef __RESEAU_H__
2  #define __RESEAU_H__
3  #include "Chaine.h"
4
5  typedef struct noeud Noeud;
6
7  /* Liste chainee de noeuds (pour la liste des noeuds du reseau ET les listes des
       voisins de chaque noeud) */
8  typedef struct cellnoeud {
9      Noeud *nd;                /* Pointeur vers le noeud stock\'e */
10     struct cellnoeud *suiv;          /* Cellule suivante dans la liste */
11 } CellNoeud;
12
13 /* Noeud du reseau */
14 struct noeud{
15    int num;                          /* Numero du noeud */
16    double x, y;                      /* Coordonnees du noeud*/
17    CellNoeud *voisins;               /* Liste des voisins du noeud */
18 };
19
20 /* Liste chainee de commodites */
21 typedef struct cellCommodite {
22     Noeud *extrA, *extrB;        /* Noeuds aux extremites de la commodite */
23     struct cellCommodite *suiv;      /* Cellule suivante dans la liste */
24 } CellCommodite;
25
26 /* Un reseau */
27 typedef struct {
28     int nbNoeuds;                     /* Nombre de noeuds du reseau */
29     int gamma;                        /* Nombre maximal de fibres par cable */
30     CellNoeud *noeuds;                /* Liste des noeuds du reseau */
31     CellCommodite *commodites;        /* Liste des commodites a relier */
32 } Reseau;
33
34 Noeud* rechercheCreeNoeudListe(Reseau *R, double x, double y);
35 Reseau* reconstitueReseauListe(Chaines *C);
36 void ecrireReseau(Reseau *R, FILE *f);
37 int nbLiaisons(Reseau *R);
38 int nbCommodites(Reseau *R);
39 void afficheReseauSVG(Reseau *R, char* nomInstance);
40 #endif
```

Cette structure permet de stocker un `Reseau` comme une liste chaînée de `Noeud` et une liste chaînée de `Commodite`. Chaque `Noeud` v est donné par son numéro, ses coordonnées et la liste chaînée des nœuds voisins, c'est-à-dire les nœuds qui sont liés au noeud v par un câble. Une `Commodite` est simplement donnée par les deux nœuds qui seront à relier par une chaîne.

Q 2.1 Créer une fonction `Noeud* rechercheCreeNoeudListe(Reseau *R, double x, double y);` qui retourne un `Noeud` du réseau R correspondant au point (x, y) dans la liste chaînée `noeuds` de R. Noter que si ce point existe dans `noeuds`, la fonction retourne un nœud existant dans `noeuds` et que, dans le cas contraire, la fonction crée un nœud et l'ajoute dans la liste des nœuds du réseau de R. Le numéro d'un nouveau nœud est simplement choisi en prenant le nombre `nbNoeuds+1` (just'avant de mettre à jour à la valeur `nbNoeuds`).

Q 2.2 Implémenter une fonction `Reseau* reconstitueReseauListe(Chaines *C);` qui reconstruit le réseau R à partir de la liste des chaînes C comme indiqué dans le pseudo-code donné au début de cette partie. Utiliser directement la liste chaînée `noeuds` du réseau pour effectuer les tests de type "$p \notin V$", en exploitant la fonction de question précédente.

Q 2.3 Créer un programme main `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 (liste, table de hachage, ou arbre).

Exercice 3 – Manipulation d'un réseau (TME7)

On veut à présent construire des méthodes pour manipuler et afficher un struct `Reseau`. Pour cela, on va stocker sur disque un `Reseau` en utilisant le format illustré par l'instance `00014_burma` qui est donné par le fichier suivant `00014_burma.res` (obtenu par l'exercice précédent).

```
NbNoeuds: 12
NbLiaisons: 15
NbCommodites: 8
Gamma: 3

v 12 16.530000 97.380000
v 11 25.230000 97.240000
v 10 20.090000 94.550000
v 9 17.200000 96.290000
v 8 16.300000 97.380000
v 7 21.520000 95.590000
v 6 14.050000 98.120000
v 5 16.470000 94.440000
v 4 22.000000 96.050000
v 3 22.390000 93.370000
v 2 20.090000 92.540000
v 1 16.470000 96.100000

l 8 12
l 11 12
l 6 11
l 3 11
l 1 10
l 3 10
l 9 10
l 8 9
l 3 7
l 1 6
```

```
l 5 6
l 2 5
l 3 4
l 2 3
l 1 2

k 5 11
k 2 6
k 11 8
k 11 1
k 8 3
k 7 6
k 4 6
k 1 3
```

Dans ce fichier :

- Les quatre premières lignes donnent le nombre de nœuds du réseau, le nombre de câbles (liaisons), le nombre de commodités, et le nombre maximal γ de fibres optiques par câble.
- Ensuite, les lignes commençant par "v" donnent les nœuds du réseau. Les nœuds sont repérés par leur numéro et leurs deux coordonnées.
- Les lignes commençant par un "l" contiennent une liaison (un câble) donnée par les numéros de ses deux extrémités.
- Les lignes commençant par un "k" correspondent à une commodité, c'est-à-dire une paire de numéros de nœuds qui devront être reliés par une chaîne.

Q 3.1 Pour écrire un tel fichier, commencer par implémenter les fonctions `nbCommodites(Reseau *R);` et `int nbLiaisons(Reseau *R);` qui comptent le nombre de commodités et de liaisons du réseau `R`.

Q 3.2 Implémenter une fonction `void ecrireReseau(Reseau *R, FILE *f);` qui écrit dans un fichier le contenu d'un `Reseau` en respectant le même format du fichier `00014_burma.res`.

Q 3.3 Dans le fichier `affichageReseau.txt`, récupérer la fonction `void afficheReseauSVG(Reseau *R, char* nomInstance);` qui permet de créer un fichier SVG en html pour visualiser un réseau. Tester votre code sur plusieurs instances en le comparant avec l'affichage des chaînes pour valider (en partie) vos fonctions.

Exercice 4 – Deuxième méthode : stockage par table de hachage (TME8)

Pour cet exercice, nous allons utiliser une table de hachage avec gestion des collisions par chaînage. La table de hachage va donc contenir un tableau de pointeurs vers une liste de nœuds. Lors du parcours de la liste des points constituant une chaîne, la table de hachage va nous permettre de déterminer rapidement si un nœud a déjà été stocké dans le réseau.

Attention : s'il n'a pas encore été stocké, il faudra le stocker **à la fois** dans la table de hachage et dans le réseau.

On espère que l'utilisation d'une table de hachage va accélérer la reconstruction du réseau.

Q 4.1 Donner une structure `TableHachage` qui permet d'implémenter une table de hachage avec gestion des collisions par chaînage. La définir dans un fichier nommé `Hachage.h`.

Q 4.2 La valeur à stocker est donnée par les coordonnées (x, y) d'un point. 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 ?

Q 4.3 Pour une table de hachage de M cases, on utilisera la fonction de hachage $h(k) = \lfloor M(kA - \lfloor kA \rfloor) \rfloor$ où $A = \frac{\sqrt{5}-1}{2}$ pour toute clef k. Plus tard, vous testerez expérimentalement plusieurs valeurs de M afin de déterminer la valeur la plus appropriée (cf Exercice 6). Il est également possible d'utiliser des fonctions de hachage bien différentes : vous pourrez proposer d'autres fonctions.

Q 4.4 Implémenter une fonction `Noeud* rechercheCreeNoeudHachage(Reseau* R, TableHachage* H, double x, double y);` qui retourne un `Noeud` du réseau R correspondant au point (x, y) dans la table de hachage H. Noter que si ce point existe dans H, la fonction retourne un point existant dans H et que, dans le cas contraire, la fonction crée un nœud et l'ajoute dans H ainsi que dans la liste des nœuds du réseau de R.

Q 4.5 Implémenter une fonction `Reseau* reconstitueReseauHachage(Chaines *C, int M);` qui reconstruit le réseau R à partir de la liste des chaînes C et en utilisant une table de hachage H de taille M.

Exercice 5 – Troisième méthode : stockage par arbre quaternaire (TME9)

Pour cet exercice, un arbre quaternaire sera utilisé pour la reconstitution du réseau. Comme pour la table de hachage, l'arbre quaternaire va nous permettre de déterminer rapidement si un nœud a déjà été stocké dans le réseau.

Un arbre quaternaire est un arbre où chaque nœud possède quatre fils. Dans un espace à deux dimensions, un arbre quaternaire représente une cellule rectangulaire. Son centre permet d'identifier les fils, qui représentent les parties nord-ouest, nord-est, sud-est et sud-ouest de l'espace par rapport à ce centre (voir Figure 2).

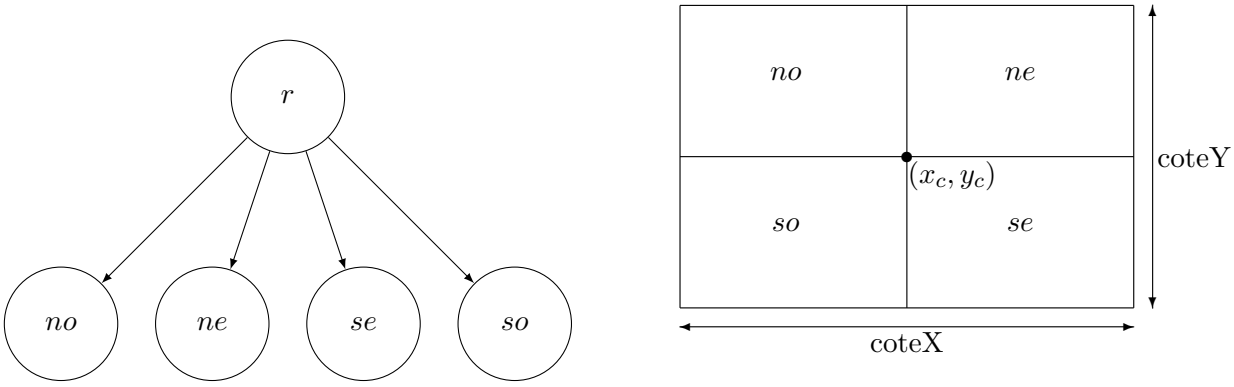

FIGURE 2 – Représentation d'un arbre quaternaire dans un espace à deux dimensions.

Les nœuds de notre réseau peuvent être stockés au niveau des feuilles de l'arbre quaternaire. En effet, on peut associer une donnée à chaque feuille de l'arbre, identifiée grâce à ses coordonnées x et y. Dans le cadre de ce projet, la donnée associée à chaque feuille sera donc un pointeur vers un `Noeud` du réseau. Par exemple, pour un réseau avec quatre nœuds n_1, n_2, n_3 et n_4, nous pourrions avoir l'arbre et la représentation de la Figure 3.

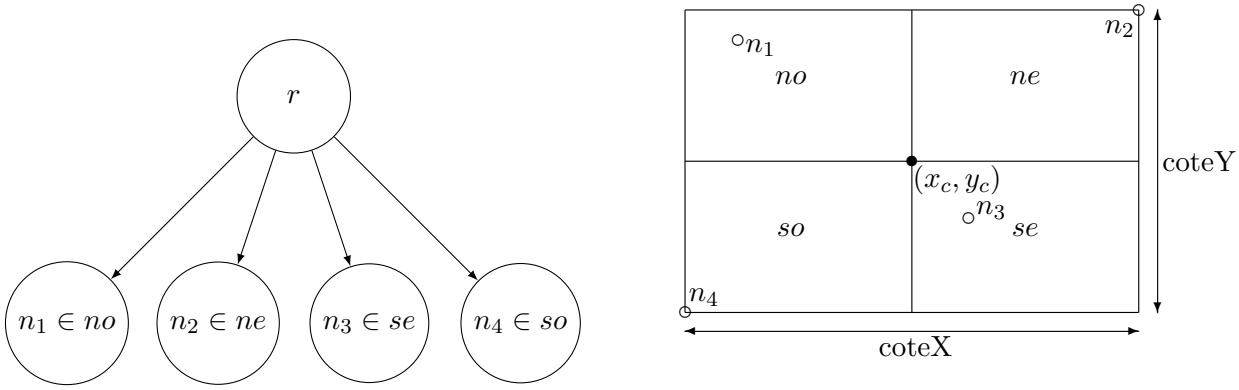

FIGURE 3 – Représentation d'un arbre quaternaire contenant quatre nœuds.

On va utiliser la structure de données suivante (fichier joint) :

```
#ifndef __ARBRE_QUAT_H__
#define __ARBRE_QUAT_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;

#endif
```

Pour pouvoir créer le nœud racine de l'arbre quaternaire qui contiendra tous les nœuds du réseau, il est nécessaire d'identifier la longueur (coteX) et la hauteur (coteY) de la cellule. Pour cela, on peut utiliser les coordonnées minimales et maximales des points à stocker dans la structure.

Q 5.1 Implémenter une fonction `void chaineCoordMinMax(Chaines* C, double* xmin, double* ymin, double* xmax, double* ymax);` qui détermine les coordonnées minimales et maximales des points constituant les différentes chaînes du réseau.

Q 5.2 Écrire la fonction `ArbreQuat* creerArbreQuat(double xc, double yc, double coteX, double coteY);` qui permet de créer une cellule de l'arbre quaternaire, de centre (x_c, y_c), de longueur coteX et de hauteur coteY. Cette fonction initialisera le nœud du réseau, les arbres nord-ouest, nord-est, sud-ouest et sud-est à NULL.

Q 5.3 Implémenter une fonction `void insererNoeudArbre(Noeud* n, ArbreQuat** a, ArbreQuat* parent);` permettant d'insérer un `Noeud` du réseau dans un arbre quaternaire. L'argument `parent` est utilisée dans cette fonction pour déterminer les dimensions de la nouvelle cellule si celle-ci doit être créée (et stockée à l'adresse indiquée par `a`). En effet, lors de l'insertion d'un nœud, trois cas sont à considérer (arbre vide, feuille et cellule interne) :

— **Arbre vide :** Dans le cas où l'arbre est vide (`(*a == NULL)`), il faut créer un arbre en utilisant la fonction `creerArbreQuat`. Les coordonnées du centre du nouvel arbre, ainsi que sa longueur

et hauteur, seront identifiées grâce à la position du point à insérer (il faut déterminer si on insère dans la partie nord-ouest, nord-est, sud-est ou sud-ouest par rapport à l'arbre parent) et grâce à la longueur et hauteur de l'arbre parent.

— **Feuille :** Si le nœud doit être inséré au niveau d'une feuille de l'arbre, c'est-à-dire si un nœud a déjà été stocké dans la cellule correspondant à l'arbre (`((*a)->noeud != NULL)`), il faut à la fois insérer le nœud n mais également l'ancien nœud de la feuille (`(*a)->noeud`), en utilisant la fonction `insererNoeudArbre` de manière récursive. Par exemple, si on reprend le cas de la Figure 3 et que l'on veut insérer un nœud n_5 dont les coordonnées figurent dans l'arbre nord-ouest de la racine, on voit que cet arbre contient déjà le nœud n_1. Il est donc nécessaire de diviser la cellule nord-ouest en quatre pour que n_1 et n_5 se trouvent dans deux feuilles différentes de l'arbre. Le résultat que l'on obtient est représenté à la Figure 4.

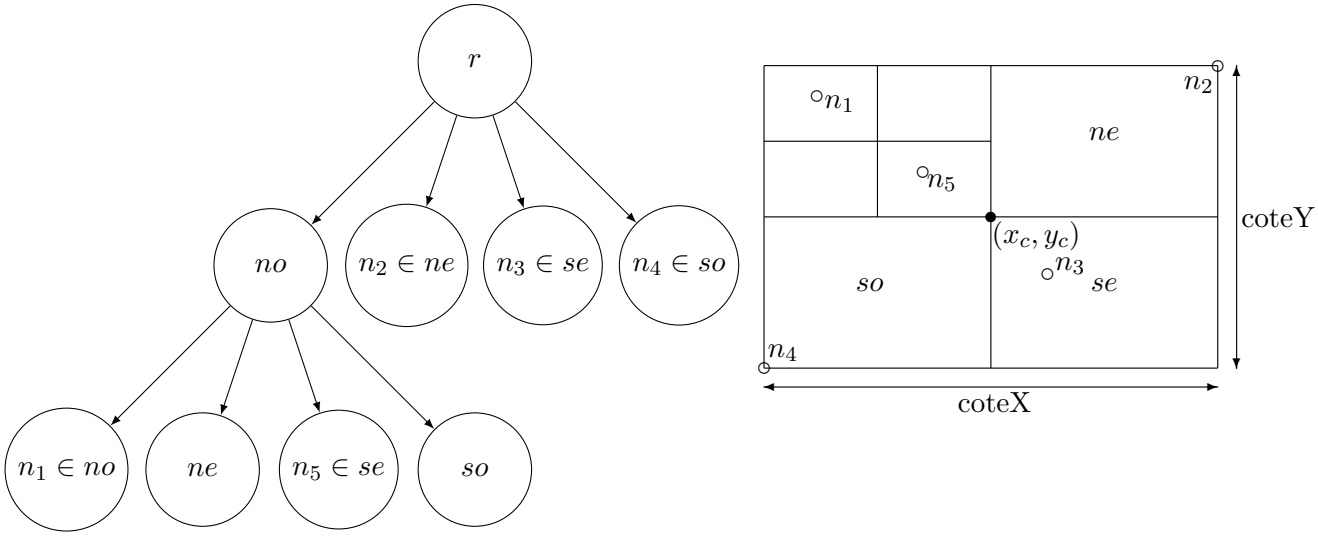

FIGURE 4 – Représentation d'un arbre quaternaire contenant cinq nœuds du réseau.

— **Cellule interne :** Dans le cas où on est sur une cellule interne de l'arbre (`(*a != NULL) && ((*a)->noeud == NULL)`), il faut déterminer, de manière récursive, dans quelle cellule de l'arbre placer le nœud du réseau (on utilise les coordonnées (x, y) du nœud du réseau que l'on compare aux coordonnées du centre de l'arbre).

Q 5.4 Écrire la fonction `Noeud* rechercheCreeNoeudArbre(Reseau* R, ArbreQuat** a, ArbreQuat* parent, double x, double y);` qui retourne un `Noeud` du réseau R correspondant au point de coordonnées (x, y) dans l'arbre quaternaire. Noter que si ce nœud existe dans l'arbre quaternaire, la fonction retourne un nœud existant dans l'arbre et que, dans le cas contraire, la fonction crée un nœud et l'ajoute dans l'arbre ainsi que dans la liste des nœuds du réseau de R. On rappelle que trois cas sont à distinguer (arbre vide, feuille et cellule interne) :

— **Arbre vide :** Dans le cas où l'arbre est vide (`(*a == NULL)`), il faut créer le nœud correspondant au point, puis l'insérer dans le réseau et dans l'arbre (avec la fonction `insererNoeudArbre`).

— **Feuille :** Si on est sur une feuille de l'arbre (`(a->noeud != NULL)`), on regarde si le nœud que l'on cherche correspond à celui de la feuille. Si c'est le cas, on retourne le nœud. Si ce n'est pas le cas, il faut créer le nœud correspondant au point, puis l'insérer dans le réseau et dans l'arbre (avec la fonction `insererNoeudArbre`).

— **Cellule interne :** Dans le cas où on tombe sur une cellule interne de l'arbre (`(*a != NULL)`

`&& ((*a)->noeud == NULL)`), il faut déterminer, de manière récursive, dans quelle cellule de l'arbre chercher le nœud du réseau (grâce aux coordonnées (x, y)).

Q 5.5 Implémenter une fonction `Reseau* reconstitueReseauArbre(Chaines* C);` qui reconstruit le réseau R à partir de la liste des chaînes C et en utilisant l'arbre quaternaire.

Exercice 6 – Comparaison des trois structures (TME10)

Dans cet exercice, vous allez comparer les temps de calcul obtenus avec les trois structures de données utilisées pour tester l'existence d'un nœud dans le réseau : la liste chaînée, la table de hachage et l'arbre quaternaire.

Q 6.1 Créer un programme `main` ou un script qui exécute automatiquement les trois fonctions de reconstruction et qui calcule uniquement leur temps de calcul. Sauvegarder dans un fichier les temps de calcul obtenus pour chacune des trois fonctions, pour les différentes instances fournies. Qu'observez-vous ?

Remarque : Pour la table de hachage, il faut aussi faire varier la taille de la table.

Q 6.2 On va maintenant comparer les temps de calcul pour de nouvelles données générées aléatoirement. Créer une fonction `Chaines* generationAleatoire(int nbChaines,int nbPointsChaine,int xmax, int ymax)` qui permet de créer des chaînes de points. Cette fonction prend en paramètre le nombre de chaînes à créer, le nombre de points par chaîne et les coordonnées maximales des points. Pour chacune des chaînes, cette fonction doit créer aléatoirement `nbPointsChaine` points situés entre les points (0,0) et (xmax,ymax).

Q 6.3 On veut maintenant construire des graphiques prenant en abscisse le nombre de points total des chaînes et en ordonnée le temps de calcul selon la structure de données utilisée. On peut par exemple utiliser les données suivantes : `nbPointsChaine = 100`, `xmax = 5000` et `ymax = 5000`, en faisant varier le nombre de chaînes de 500 à 5000 par pas de 500. Faire deux graphiques différents :

- Un graphique donnant les temps de calcul avec la liste chaînée.
- Un graphique comprenant les résultats obtenus avec la table de hachage et l'arbre quaternaire.

Remarque : Pour la table de hachage, il faudra aussi faire varier la taille de la table.

Q 6.4 Analyser vos résultats.

Optimisation du réseau

Le but de cette partie est d'optimiser l'utilisation des fibres optiques du réseau. L'objectif est de relier les commodités par une chaîne dans le réseau. Pour évaluer les chaînes que vous obtiendrez, on se basera sur le critère suivant : minimiser la somme totale des longueurs des chaînes.

Exercice 7 – Parcours en largeur (TME11)

Dans cette partie, on considère la structure de graphe suivante :

```c
#ifndef __GRAPHE_H__
#define __GRAPHE_H__
#include<stdlib.h>
#include<stdio.h>
#include "Struct_Liste.h"

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

typedef struct{
  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;

#endif
```

Q 7.1 Écrire une fonction `Graphe* creerGraphe(Reseau* r);` qui crée un graphe à partir d'un réseau. Attention :

- Le numéro des nœuds dans le graphe doit correspondre à leur position dans le tableau de sommets.
- Chaque arête $\{u, v\}$ du graphe est allouée une seule fois, mais apparaît dans la liste des voisins du sommet u et du sommet v dans le graphe.

Q 7.2 Coder une fonction retournant le plus petit nombre d'arêtes d'une chaîne entre deux sommets u et v d'un graphe (il s'agit d'un parcours en largeur vu en cours). Utiliser la bibliothèque `File` fournie, pour coder la bordure de manière efficace.

Q 7.3 Proposer une façon de stocker l'arborescence des chemins issus de u dans la fonction précédente. Déduire de cette arborescence une chaîne de u à v. Transformer votre fonction pour qu'elle retourne une liste d'entiers correspondant à cette chaîne (le code permettant de manipuler une liste d'entiers est fourni sur le site).

Q 7.4 Proposer une fonction `int reorganiseReseau(Reseau* r)` :

- qui crée le graphe correspondant au réseau,
- qui calcule la plus courte chaîne pour chaque commodité,
- et qui retourne vrai si pour toute arête du graphe, le nombre de chaînes qui passe par cette arête est inférieur à γ, et faux sinon. On créera une matrice permettant de compter le nombre

de chaînes passant par chaque arête $\{u, v\}$.

Q 7.5 Tester votre fonction `reorganiseReseau` sur plusieurs instances, et analyser les résultats. Que proposez-vous pour améliorer la fonction (on ne vous demande pas d'implémenter votre solution).