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

  rendu de la partie 1 (code et solutionss) lors du TME 8 (semaine du 20/03/17)

- Partie 2 : Réorganisation du réseau

  l'ensemble des deux parties est à rendre lors du TME 11 (semaine du 26/04/17)

Les solutions de la partie 1 seront distribuées à tous lors de la séance 8 pour attaquer la partie 2 sur une même base pour tous.

Pour le rendu final, vous montrerez à votre chargé de TD votre code, son fonctionnement et ses performances (il y a un document sur le site du module précisant le contenu d'un rendu). Pour les étudiants qui ne peuvent être présents au TME11, vous devez contacter vos chargés de TD afin de prévoir une autre date au plus près de la semaine de rendu.

Chaque partie est divisée en exercices, qui vont vous permettre de concevoir progressivement le programme final. Chacun des exercices est le sujet des séances de TME de 3 à 11. Il est conseillé de suivre les étapes données par ces différents exercices, car chaque exercice est l'application directe de notions qui ont été introduites en cours et en TD en parallèle au projet.

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.

Remarque : Cet énoncé peut connaître des petites évolutions afin d'apporter des précisions ou des indications : venez en Cours/TD/TME pour les connaître et consulter fréquemment la page du module : `https://www.licence.info.upmc.fr/lmd/licence/2016/ue/2I006-2017fev/`

Cadre du projet

Dans ce travail, nous considérons une agglomération dont les services municipaux désirent améliorer le réseau de fibres optiques de ses administrés. On appellera ici *réseau* 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. 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 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. Nous allons donc tester plusieurs algorithmes pour chacune des parties.

• *Modélisation et notations*

On appelle *client* du réseau des points qui ont un rôle important : des entreprises clientes de l'opérateur, des locaux techniques de l'opérateur,... Un *concentrateur* est un point du réseau où se rejoignent plusieurs câbles. Un *câble* du réseau est un fourreau (ou une gaine) contenant exactement γ fibres optiques. Les câbles relient deux points du plan. Un point peut être un client, un concentrateur ou à la fois un client et un concentrateur. Les tronçons de fibres optiques de deux câbles qui arrivent à un même concentrateur peuvent alors 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 du plan : on appelle ce couple de points *une commodité*. Il existe plusieurs opérateurs dans l'agglomération et chaque opérateur possède plusieurs chaînes de fibres optiques.

• *Un exemple*

La figure 1 indique une instance de notre problème. Elle décrit un réseau de 7 points. Il y a 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 $(4, 1, 5, 6)$ reliant la commodité $(4, 6)$.

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 ; que le point 4 est uniquement un concentrateur ; et que les

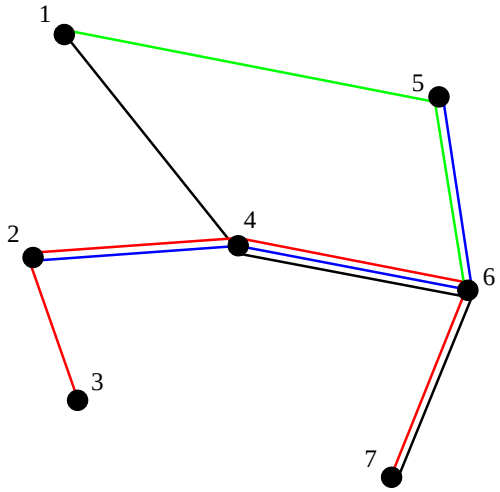

FIGURE 1 – Un exemple de réseau

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 sept câbles dans ce réseau. Par exemple, dans le câble $(1, 4)$, une seule fibre est utilisée. Dans le câble $(4, 6)$, trois fibres sont utilisées. Ce dessin ne donne pas l'indication du nombre γ du 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,...), d'autres proviennent de réseaux non géographiques (réseaux électroniques,...).
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.

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

Lecture, stockage et affichage des données

On veut pouvoir lire les données du réseau, les stocker dans des structures de données adéquates et les afficher proprement.

Remarque 1 : Dans cet exercice et dans la suite du projet, vous allez plusieurs fois devoir manipuler des listes chaînées d'éléments. Comme les opérations à réaliser sur les listes seront identiques quels que soient les éléments, n'hésitez pas réutiliser les fonctions créées pour un élément donné, en les adaptant à d'autres types d'éléments. Les opérations de base à réaliser sont typiquement (ici pour le cas d'un élément composé de trois champs : un entier et deux doubles) :

- `Element *creerElement(int var1, double var2, double var3)` : création de l'élément
- `CellElement *creerCellElement(Element *cour, CellElement *suiv)` : création d'une cellule de la liste des éléments
- `void detruireLE(CellElement *L)` : libération de la mémoire occupée par la liste des éléments

Avec :

```c
/* Element */
typedef struct element{
    int var1;
    double var2, var3;
} Element;

/* Liste chainee d'elements */
typedef struct cellElement {
    struct element *cour;           /* Pointeur vers l'element courant */
    struct cellElement *suiv;       /* Cellule suivante dans la liste */
} CellElement;
```

Remarque 2 : N'oubliez pas de désallouer les zones mémoires qui ne sont plus utilisées ! (notamment en fin de programme)

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 étant 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.

```
#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* lectureChaine(FILE *f);
void ecrireChaineTxt(Chaines *C, FILE *f);
void afficheChaineSVG(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.

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

Dans ce premier exercice, nous allons construire une bibliothèque de manipulations d'instances de type `Chaine` : lecture du fichier et affichage graphique.

Q 1.1 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 d'entrée (vous pouvez utiliser la bibliothèque d'Entrée/Sortie qui vous a été fournie au TME2-3).

Q 1.2 Dans le but de valider votre code de lecture d'instance, construire une fonction `void ecrireChaineTxt(Chaines *C, FILE *f);` qui écrit dans un fichier le contenu d'une `Chaine` en respectant le même format que celui contenu dans le fichier d'origine (cela revient à recréer le fichier d'entrée). Le but de cette fonction est de tester votre code sur plusieurs instances afin de le tester avant d'attaquer la suite. Pour cela, créer un main `ChainMain` permettant d'exécuter les deux

fonctions (vous pouvez utiliser la ligne de commande pour passer le nom du fichier d'instance).

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 le site du module 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 (pensez à initialiser la génération aléatoire si vous désirez obtenir des couleurs diverses).
Implémenter une fonction `void afficheChaineSVG(Chaines *C, char* nomInstance);` qui permet de créer le fichier SVG en html à partir d'un struct `Chaines`.

Q 1.4 Implémenter les fonctions `double longueurChaine(CellChaine *c);`
puis `double longueurTotale(Chaines *C);` qui permettent de calculer la longueur physique d'une chaîne et la longueur physique totale des chaînes de votre instance. 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 $d(A, B) = \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}$.

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

PARTIE 1 : Reconstitution du réseau (TME4-7)

Le but de cette partie est de reconstituer le réseau à partir des chaînes : c'est-à-dire, à partir de l'ensemble des chaînes, trouver la liste des *nœuds du réseau* : pour différencier entre les points de départ des chaînes, on appelera plutôt nœuds les points dans le réseau. A une coordonnée donnée, il y a un unique nœud pour lequel on connaît tous les câbles qui en sont issus : et ainsi les nœuds qui sont au bout de ces câbles : on appelera ces nœuds des *nœuds voisins*. On conserve aussi la liste des commodités.

L'*algorithme de reconstitution* est très simple :
On utilise un ensemble des points du réseau V qui est initialisé vide : $V \leftarrow \emptyset$
On parcourt une à une chaque chaîne :

Pour chaque point p de la chaîne : on teste si $p \in V$
(c'est-à-dire qu'on teste si le point a déjà été rencontré auparavant)
S'il n'a pas encore été stocké, on l'ajoute dans V.
On met à jour en même temps la liste des voisins de p en ajoutant le (ou les deux) points qui sont reliés à p par des câbles.
On conserve la commodité de la chaîne.

Et on passe à la chaîne suivante.

Dans cette première partie, on ne tiendra pas compte de la valeur `Gamma` γ qui sera utilisée dans la partie 2 : on le conserve par contre d'une instance `Chaine` vers l'instance réseau que l'on construit.

Trois méthodes vont être comparées qui correspondent en fait chacune à trois structures de données pour implémenter V : une simple liste chaînée, une table de hachage ou des arbres.

Exercice 2 – Première méthode : stockage par liste chaînée (TME4)

Dans cet exercice, on désire implémenter l'algorithme de reconstitution de réseau en codant l'ensemble des points par une liste chaînée.

Pour cela, nous avons besoin d'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 sera repéré par ses coordonnées. De plus, pour chaque nœud u, on connaît la liste des pointeurs sur nœuds qui sont reliés à u par un câble.
Chaque câble est donné par des pointeurs sur ses deux nœuds extrémités.
Une commodité est une paire de pointeurs sur les nœuds du réseau qui doivent être reliés par une chaîne.
Pour permettre la sauvegarde sur disque des instances, lors de la reconstitution du réseau, on attribuera également un numéro entier unique qu'on lui choisira incrémentalement.

Pour stocker les données du réseau, on utilisera la structure de données suivante (fichier fourni) dans `Reseau.h`

```
1 #ifndef __RESEAU_H__
```

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

Cette structure permet de stocker un `Reseau` comme une liste chaînée de `Noeud` et une liste chaînée de `Commodite`. Chaque `Noeud` 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 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 `noeud`, la fonction retourne un point existant dans `noeud` et que, dans le cas contraire, la fonction crée un nœud et l'ajoute dans `noeud` ainsi que 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 le mettre à jour !).

Q 2.2 Implémenter une fonction `Reseau* reconstitueReseauListe(Chaines *C);` qui reconstruit le réseau R à partir de la liste des chaînes C et en utilisant pour structure directement la liste chaînée `noeud`.

Q 2.3 Créer un programme main `ReconstitueReseau` qui utilise la ligne de commande pour prendre

un fichier .cha en paramètre et un nombre entier indiquant quelle méthode (parmi les 3 de cette partie) l'on désire utiliser.

Exercice 3 – Manipulation d'un réseau (TME4-5)

On veut à présent construire des méthodes pour manipuler et afficher un struct `Reseau`.
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
NbLiaison: 15
NbCommodite: 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
```

Les deux premières lignes donnent le nombre de nœuds et le nombre maximal γ de fibres optiques par câble (que l'on a recopié depuis l'instance des chaines de départ). Les différents nœuds du réseau sont ensuite donnés.
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 Implémenter les fonctions `nbCommodites(Reseau *R);` et `int nbLiaison(Reseau *R);` qui comptent le nombre de commodités et des liaisons d'un réseau.

Q 3.2 Implémenter une fonction `void ecrireReseauTxt(Reseau *R, FILE *f);` qui écrit dans un fichier le contenu d'un `Reseau` en respectant le même format que celui contenu dans le fichier d'origine (cela revient à recréer le fichier d'entrée). Trouvez-vous bien le même fichier que l'exemple **00014_burma** ?

Q 3.3 Implémenter une 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 et de valider (en partie) vos fonctions.

Exercice 4 – Deuxième méthode : stockage par table de hachage (TME 5)

Pour cet exercice, nous allons utiliser une table de hachage par chaînage.

La table de hachage va contenir un tableau de pointeurs vers une liste de nœuds `CellNoeud*`. Lors du parcours de la liste des nœuds constituant les chaînes, la table de hachage va nous permettre de déterminer rapidement si le 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 l'insertion et de la recherche d'un point dans la structure du réseau.

Q 4.1 Donner une structure `Hachage` qui permet d'implémenter une structure de liste de hachage par chaînage. 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 ? 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. Vous testerez expérimentalement plusieurs valeurs de M afin de déterminer la valeur la plus appropriée.
Il est également possible d'utiliser des fonctions de hachage bien différentes : vous pouvez proposer d'autres fonctions.

Q 4.2 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 de R. 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 4.3 Implémenter une fonction `Reseau* recreeReseauHachage(Chaines *C);` qui reconstruit le réseau R à partir de la liste des chaînes C et en utilisant la table de hachage H.

Exercice 5 – Reconstruction du réseau 2 : Arbre quaternaire (TME 6)

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).

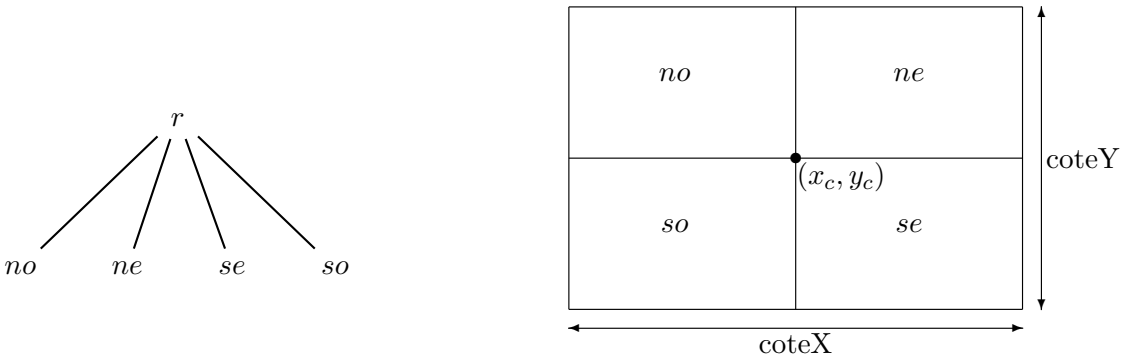

FIGURE 2 – Représentation d'un arbre quaternaire dans un espace à deux dimensions.

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 un pointeur vers un `Noeud n` du réseau. Par exemple, pour quatre nœuds n_1, n_2, n_3 et n_4 du réseau, nous pourrons avoir l'arbre et la représentation de la Figure 3 :

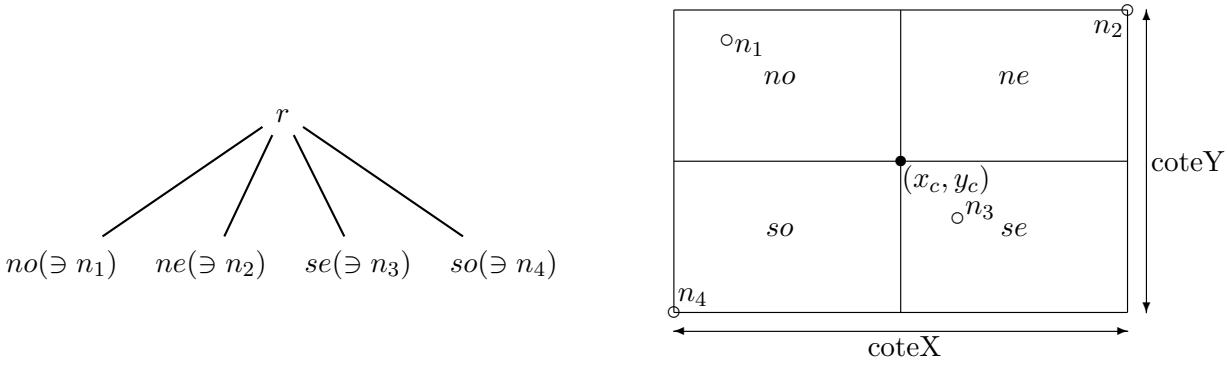

FIGURE 3 – Représentation d'un arbre quaternaire contenant quatre nœuds du réseau.

On peut donc utiliser la structure de données suivante :

```
1 #ifndef __ARBRE_QUAT_H__
2 #define __ARBRE_QUAT_H__
3
```

```
4  /* Arbre quaternaire contenant les noeuds du reseau */
5  typedef struct arbreQuat{
6      double xc, yc;          /* Coordonnees du centre de la cellule */
7      double coteX;           /* Longueur de la cellule */
8      double coteY;           /* Hauteur de la cellule */
9      Noeud* noeud;           /* Pointeur vers le noeud du reseau */
10     struct arbreQuat *so;   /* Sous-arbre sud-ouest, pour x < xc et y < yc */
11     struct arbreQuat *se;   /* Sous-arbre sud-est, pour x >= xc et y < yc */
12     struct arbreQuat *no;   /* Sous-arbre nord-ouest, pour x < xc et y >= yc */
13     struct arbreQuat *ne;   /* Sous-arbre nord-est, pour x >= xc et y >= yc */
14 } ArbreQuat;
15
16 #endif
```

Pour pouvoir créer le nœud racine de l'arbre quaternaire, 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 Implémenter une 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 `ArbreQuat* insererNoeudArbre(Noeud* n, ArbreQuat* a, ArbreQuat* parent);` permettant d'insérer un `Noeud` du réseau dans l'arbre quaternaire. La cellule parent est utilisée dans cette fonction pour déterminer les dimensions de la nouvelle cellule si celle-ci doit être créée.
Trois cas sont à considérer (arbre vide, feuille et cellule interne) :

- **Arbre vide :** Le cas où l'arbre a est vide (`(a == NULL)`). Dans ce cas, il faut créer l'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.
- **Cellule interne :** Le dernier cas est le cas le cas où on tombe sur une cellule interne de l'arbre (`(a != NULL)` et `(a->noeud == NULL)`). Dans ce cas, 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 Implémenter une fonction `Noeud* chercherNoeudArbre(CellPoint* pt, Reseau* R, ArbreQuat**`

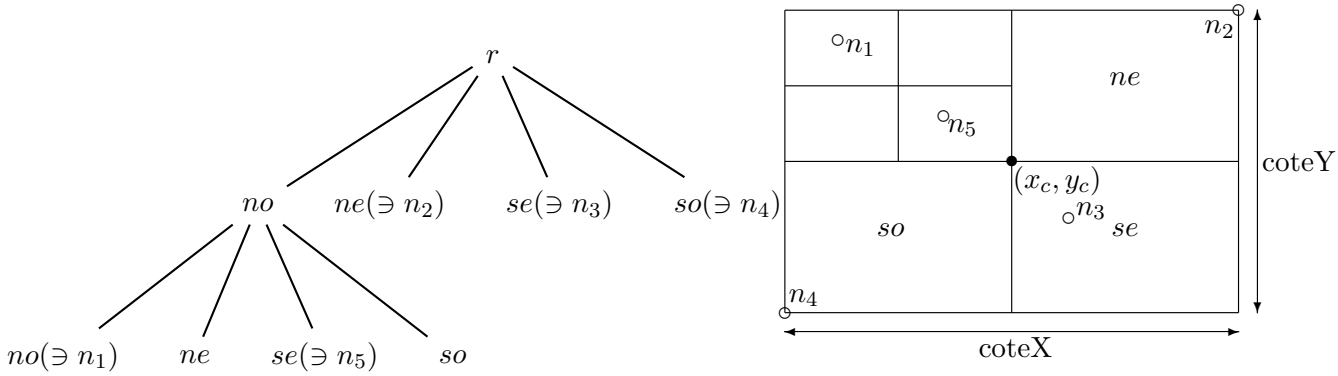

FIGURE 4 – Représentation d'un arbre quaternaire contenant cinq nœuds du réseau.

`aptr, ArbreQuat* parent);` qui retourne un `Noeud` du réseau R correspondant au point de coordonnées (pt→x,pt→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.
Trois cas sont à considérer (arbre vide, feuille et cellule interne) :

- **Arbre vide :** Le cas où l'arbre a est vide (`(a == NULL)`). Dans ce cas, il faut créer le nœud, l'insérer dans le réseau et dans l'arbre.
- **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 ce n'est pas le cas, il faut créer le nœud, l'insérer dans le réseau et dans l'arbre.
- **Cellule interne :** Le dernier cas est le cas où on tombe sur une cellule interne de l'arbre (`(a != NULL)` et `(a->noeud == NULL)`). Dans ce cas, il faut déterminer, de manière récursive, dans quelle cellule de l'arbre chercher le nœud du réseau (grâce aux coordonnées (pt→x,pt→y) du nœud du réseau).

Q 5.5 Implémenter une fonction `Reseau* recreeReseauArbre(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 deux structures (TME 7)

Dans cet exercice, vous allez comparer en terme de temps de calcul les deux structures de données utilisées pour la reconstruction du réseau : la table de hachage et l'arbre quaternaire.

Q 6.1 Créer un programme main ou un script qui exéctute automatiquement les trois fonctions de reconstruction et qui calcule uniquement les temps de calcul des trois fonctions de reconstructions. Sauvegarder dans un fichier les temps de calcul obtenus pour chacune des trois fonctions pour les différentes instances fournies.

Q 6.2 Construire un graphique comprenant en abscisse le nombre de points total des chaînes et en ordonnée le temps de calcul pour la reconstruction du réseau pour les deux structures de données considérées. Concluez sur vos résultats.

RENDU TME 8 : Vous devez rendre le travail réalisé pour les parties 1 et 2 au TME 8. Pour cela,

vous enverrez à votre chargé de TD un fichier .tar ou .zip contenant les fichiers sources, les fichiers réseaux que vous avez réussi à reconstruire à partir des chaînes, ainsi que leurs images générées avec Gnuplot. Le fichier contiendra également les courbes que vous avez obtenues lors de l'exercice 5. Votre code doit pouvoir être testé facilement (menu en console).
Nous vous incitons également à commencer le rapport final qui sera à rendre en fin de projet au TME 11.

PARTIE 2 : Optimisation du réseau

Le but de cette partie est d'optimiser l'utilisation des fibres optiques du réseau. L'objectif est de relier les deux extrémités de chacune des extrémités par une chaîne dans le réseau. Pour évaluer les chaînes que vous obtiendrez, on se basera sur deux critères : minimiser la somme totale des longueurs des chaînes et minimiser le nombre maximal γ de fibres utilisées par câble.

Pour cette partie 2, les instances sont les fichiers décrivant les réseaux obtenus dans la partie1. Il est NECESSAIRE d'utiliser les fichiers .res fournis sur le site pour la partie 2 afin d'avoir tous la même numération des sommets.

Exercice 7 – Parcours en largeur (TME8)

Q 7.1 Récupérer et tester tout d'abord les fichiers de code.
- des sructures de `listes` et de `files` d'entiers qui vous seront utiles plusieurs fois dans cette partie.
- une structure de `Graphe` permettant de lire un fichier .res au format Graphe (similaire à celui vu en cours)
- une fonction `evaluation_Nchaine`
- un programme `MainGraphe` qui prend en argument le nom d'un fichier .res (sans extension), qui le lit et le stocke, puis qui crée sur disque une visualisation.

Q 7.2 Coder une fonction retournant le plus petit nombre d'arêtes d'un chemin entre entre deux sommets u et v d'un graphe (il s'agit d'un parcours en largeur vu en cours).

Q 7.3 Proposer une façon de stocker l'arborescence des chemins issus de u dans la fonction précédente. Déduire de cette arborescence un chemin de u à v. Transformer votre fonction pour qu'elle retourne une liste d'entiers correspondant à ce chemin.

Q 7.4 Utiliser la fonction précédente pour retourner des chaînes reliant les extrémités de chaque commodité du graphe.

Q 7.5 Ecrire votre solution dans un fichier texte d'extension .ncha (pour NOUVELLE chaînes) tel que :
- chaque ligne correspond à une commodité (u, v) du fichier .res en respectant cet ordre,
- chaque ligne donne la liste des sommets d'une chaîne du graphe de u à v dans cet ordre,
- chaque ligne se termine par -1.

Par exemple on peut obtenir le fichier `00014_burma.ncha` suivant

```
5 6 11 -1
2 1 6 -1
11 12 8 -1
11 6 1 -1
8 9 10 3 -1
7 3 11 6 -1
4 3 11 6 -1
1 2 3 -1
```

Q 7.6 On veut évaluer une solution NCHAINE, pour cela, calculer :

- `longueur:` la longueur totale des chemins obtenus EN DISTANCE EUCLIDIENNE des arêtes (vous pouvez remarquer que la structure graphe contient cette valeur pour chacune des arêtes).
- `gamma:` le nombre maximal de chaînes qui passe par la même arête.

Pour le calcul de `gamma`, vous pouvez ajouter un champ à la structure `Graphe` afin de pouvoir tenir le compte du nombre de fois où une arête est utiliser pour les chaînes de la solution.

Q 7.7 Pour évaluer la partie 2 de ce projet, on va uniquement observer trois instances : "00783_rat", "05934_rl" et "07397_pla".
La fonction d'évaluation contenue dans `evaluation_NChaines` est
`double evaluation(int gamma, double longueur, char* nom);`.
Elle prend en paramètre les évaluations `longueur` et `gamma`, ainsi que le nom de l'instance (uniquement parmi "00783_rat", "05934_rl" et "07397_pla").
Cette fonction tient compte de la distance euclidienne normalisée entre le point présentant en abscisse la valeur de gamma et en ordonnée la longueur des chaînes obtenus par votre algorithme et un point "idéal" de faible gamma et de faible longueur (donné dans la fonction). Plus la distance entre votre point et le point "idéal" est faible, plus la qualité de votre solution est bonne. La fonction renvoie une valeur égale à 100 si vous avez réussi à atteindre le point "idéal".

Le but de la suite de cette partie 2 est de maximiser cette évaluation pour les rois instances.

Exercice 8 – Plus courts chemin : algorithme de Dijkstra (TME 8-9)

Dans le but de pouvoir déterminer efficacement des chaînes de longueur minimale, nous allons implémenter l'algorithme de Dijkstra. Dans cet algorithme, il est nécessaire de retrouver facilement un élément de coût minimum parmi un ensemble d'éléments. La structure de données de type tas est donc parfaitement adaptée et nous allons construire un tas spécifique à l'algorithme de Dijsktra.

Q 8.1 Le tas que l'on considère contient un élément composé d'un entier i et d'un réel c. L'entier i s'appelle le numéro (qui correspondra au numéro du sommet dans le cas de Dijkstra) et le réel c est la clef (qui correspondra à une distance dans le cas de Dijkstra). Le tas permet de retrouver rapidement l'élément de plus petite clef.
Dans cette structure de tas, nous allons uniquement insérer des éléments dont le numéro i est un entier entre 0 et $n-1$ où n est un entier connu dès le départ. De plus chaque élément contiendra au plus un seul élément par numéro (donc le tas contient au plus n éléments).
A partir de l'exercice 2 du TD 7, implémenter une structure de tas correspondant à cette description possédant les fonctions :

- un accès en $O(1)$ au couple (f, i) ayant la plus petite valeur c.
- insertion en $O(log(n))$ d'un couple (i, c).
- suppression en $O(log(n))$ de l'élément de plus petite valeur c.
- test en $O(1)$ de la présence ou non d'un élément (i, c) pour une valeur i donnée.
- suppression, s'il existe, en $O(log(n))$, de l'élément (i, c) pour une valeur i donnée.

Q 8.2 Implémenter l'algorithme de Dijkstra comme vu en cours et en TD. Vous utiliserez la structure de tas implémentée dans la question précédente pour à la fois déterminer le sommet le plus proche des sommets visités et pour déterminer s'il y a encore des sommets à visiter.

Q 8.3 Appliquer l'algorithme de Dijkstra sur l'ensemble des commodités du réseau. Calculer les évaluations pour les chaînes obtenues.

Exercice 9 – A vous de jouer ! (TME 10)

Dans les deux exercices précédents, vous avez minimisé les longueurs des chaînes reliant les différentes commodités sans vous préoccuper de la valeur γ du réseau (c'est-à-dire le nombre de fibres optiques maximum par câble). Dans cet exercice, vous allez devoir tenir compte de ces deux critères : minimiser la longueur des chaînes et minimiser la valeur maximale d'utilisation des câbles.
La recherche "indépendante" de chemins ne permet pas de tenir compte de l'utilisation des câbles. Vous devez soit adapter les méthodes précentes, soit proposer une toute nouvelle méthode : à vous de jouer !

Q 9.1 Proposer au moins une nouvelle méthode !
Vos algorithmes doivent afficher les résultats pour chacune des instances. Un exemple d'affichage est le suivant : `Instance = 00783_rat, Evaluation = 95.40, gamma = 25, distance = 96282.54`

Q 9.2 Au cours de votre recherche de solutions, vous pouvez envoyer vos fichier .ncha aux responsables de TD afin de valider vos solutions. ATTENTION : il doit bien s'agir de solutions valides et meilleure que celles obtenues à l'exercice précédent.

BONUS : Les binôme ayant obtenu les meilleurs résultats se verront attribuer des points bonus !

RENDU FINAL SEMAINE 11 :
Vous devez rendre l'intégralité du travail (y compris ce qui a déjà été rendu pour la partie 1) en semaine 11.
Pour cela, vous enverrez à votre chargé de TD un fichier .tar ou .zip contenant les fichiers sources ; un rapport les résultats obtenus sur les exercices des DEUX PARTIES ; et les résultats obtenus avec vos algorithmes d'optimisation de la partie 2. Le rapport contiendra également les courbes que vous avez obtenues lors des exercices d'évaluation de la partie 1.
Veuillez vous reporter au document concernant les rendus sur le site.
Votre code doit pouvoir être testé facilement (arguments en ligne de commande ou menu en console).
Vous devez envoyer vos fichiers **le jour précédent la 11ème séance de TME**, avant minuit. A l'occasion de la 11ème séance de TME, vous présenterez votre travail à votre chargé de TD/TME lors d'une mini-soutenance de quelques minutes. A cette occasion, vous lui présenterez notamment les résultats obtenus pour l'exercice 8.

Remarque : En attendant votre tour de passage, vous pouvez mettre en œuvre les exercices du TD11.