Projet :Réorganisation d’un réseau de fibres optiques

Realisé par:

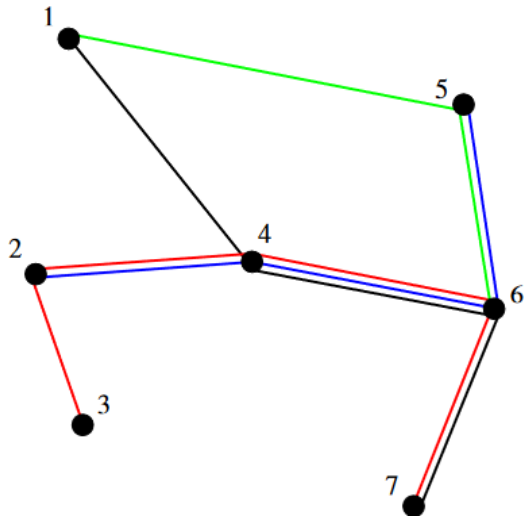
GRIM Tassadit **CHIKBOUNI Melissa**

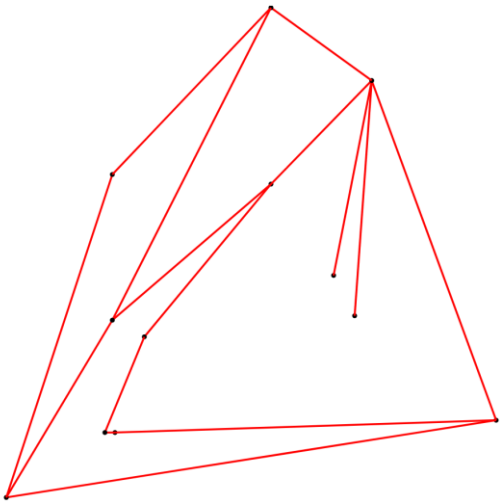
GROUPE 10

Introduction:

Ce rapport présente un projet visant à améliorer le réseau de fibres optiques d'une agglomération donnée. L'objectif principal de ce projet est de proposer des méthodes efficaces pour la reconstruction du plan du réseau et la réorganisation des attributions de fibres entre les différents opérateurs en utilisant des structures de données différentes : Liste chaînées, table de hachage, arbres quaternaires et graphes.

Un réseau est constitué d'un ensemble de câbles, chacun abritant plusieurs fibres optiques et servant à connecter différents clients entre eux.





Dans un premier temps notre réseau est représenté par la structure suivante :

```
#ifndef __RESEAU_H__
#define __RESEAU_H__
#include "Chaine.h"

typedef struct noeud Noeud;

/* Liste chainee de noeuds (pour la liste des noeuds du reseau ET les listes des
   voisins de chaque noeud) */
typedef struct cellnoeud {
    Noeud *nd;                /* Pointeur vers le noeud stock\'e */
    struct cellnoeud *suiv;          /* Cellule suivante dans la liste */
} CellNoeud;

/* Noeud du reseau */
struct noeud{
   int num;                          /* Numero du noeud */
   double x, y;                      /* Coordonnees du noeud*/
   CellNoeud *voisins;               /* Liste des voisins du noeud */
};

/* Liste chainee de commodites */
typedef struct cellCommodite {
    Noeud *extrA, *extrB;        /* Noeuds aux extremites de la commodite */
    struct cellCommodite *suiv;      /* Cellule suivante dans la liste */
} CellCommodite;

/* Un reseau */
typedef struct {
    int nbNoeuds;                     /* Nombre de noeuds du reseau */
    int gamma;                        /* Nombre maximal de fibres par cable */
    CellNoeud *noeuds;                /* Liste des noeuds du reseau */
    CellCommodite *commodites;        /* Liste des commodites a relier */
} Reseau;
```

Fonctions utilisées:

```c
/* ~~~~~~ Fonctions ajoutées pour un code modulaire / libération  ~~~~~~ */
void libererChaines(Chaines *C);
double distance(CellPoint *p1, CellPoint *p2);
int comptePoints(CellPoint *points) ;


/* ~~~~~~ Fonctions principales ~~~~~~ */

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

Exécution et résultats:

```
tassadit@tassadit-Latitude-5480:~/Téléchargements/LastVersion$ ./ChaineMain 00014_burma.cha testChaineMain.txt

Longueur totale des chaînes : 85.97
Nombre total de points : 30
```

```
NbChain: 8
Gamma: 3
7 3 16.47 96.10 20.09 92.54 22.39 93.37
6 5 22.00 96.05 22.39 93.37 20.09 92.54 16.47 94.44 14.05 98.12
5 5 21.52 95.59 22.39 93.37 20.09 92.54 16.47 94.44 14.05 98.12
4 4 16.30 97.38 17.20 96.29 20.09 94.55 22.39 93.37
3 4 25.23 97.24 22.39 93.37 20.09 94.55 16.47 96.10
2 3 25.23 97.24 16.53 97.38 16.30 97.38
1 3 20.09 92.54 16.47 96.10 14.05 98.12
0 3 16.47 94.44 14.05 98.12 25.23 97.24
```

Première partie : Reconstitution

Dans la première partie du projet, nous nous concentrons sur la reconstitution du plan complet du réseau à partir des chaînes de fibres optiques selon l'algorithme suivant :

- On utilise un ensemble de nœuds V initialisé vide.
- Pour chaque chaîne de fibres optiques :
    - Pour chaque point p de la chaîne :
        - Si le point p n'appartient pas à V (c'est-à-dire s'il n'a pas déjà été rencontré auparavant) :
            - Ajouter un nœud correspondant au point p dans l'ensemble V.
            - Mettre à jour la liste des voisins de p ainsi que celle de ses voisins.
            - Conserver la commodité de la chaîne.

Dans cette partie du projet, nous nous concentrons sur l'optimisation du test "le point p n'appartient pas à V". Pour cela, nous étudions trois méthodes différentes qui correspondent à trois structures de données :

- Une liste chaînée,
- Une table de hachage
- Des arbres.

Chacune de ces méthodes présente des avantages et des inconvénients en termes de complexité temporelle et spatiale, et nous les évaluons pour déterminer laquelle est la plus adaptée à notre problème de reconstitution du réseau de fibres optiques.

Première méthode : stockage par liste chaînée

Le réseau dans cette partie est stocké selon la structure suivante :

```c
#ifndef __RESEAU_H__
#define __RESEAU_H__
#include "Chaine.h"

typedef struct noeud Noeud;

/* Liste chainee de noeuds (pour la liste des noeuds du reseau ET les listes des
    voisins de chaque noeud) */
typedef struct cellnoeud {
    Noeud *nd;                /* Pointeur vers le noeud stock\'e */
    struct cellnoeud *suiv;         /* Cellule suivante dans la liste */
} CellNoeud;

/* Noeud du reseau */
struct noeud{
   int num;                         /* Numero du noeud */
   double x, y;                     /* Coordonnees du noeud*/
   CellNoeud *voisins;              /* Liste des voisins du noeud */
};

/* Liste chainee de commodites */
typedef struct cellCommodite {
    Noeud *extrA, *extrB;       /* Noeuds aux extremites de la commodite */
    struct cellCommodite *suiv;     /* Cellule suivante dans la liste */
} CellCommodite;

/* Un reseau */
typedef struct {
    int nbNoeuds;                    /* Nombre de noeuds du reseau */
    int gamma;                       /* Nombre maximal de fibres par cable */
    CellNoeud *noeuds;               /* Liste des noeuds du reseau */
    CellCommodite *commodites;       /* Liste des commodites a relier */
} Reseau;
```

Les fonctions implémentées :

```c
void construireLiaison(Noeud *n1, Noeud *n2);
void ajouterVoisin(Noeud *n, Noeud *voisin);
void detruireReseau(Reseau* res);
void ecrireReseau(Reseau *R, FILE *f);
int nbLiaisons(Reseau *R);
int nbCommodites(Reseau *R);
void afficheReseauSVG(Reseau *R, char* nomInstance);
```

Deuxième méthode : stockage par table de hachage (résolution de collision par chaînage)

On utilise la structure de table de hachage suivante pour le stockage des nœuds du réseau :

```
typedef struct{
  int tailleMax;
  CellNoeud** T;
} TableHachage ;
```

Les fonctions implémentées :

```
/* ~~~~~~ Fonctions ajoutees pour un code modulaire / liberation  ~~~~~~ */
void libererTableHachage(TableHachage *H);
Noeud * creeNoeud (Reseau * R , double x , double y );



/* ~~~~~~ Fonctions principales ~~~~~~ */

int fonctionHachage(int M, double k);
double Clef(double x, double y);
Noeud* rechercheCreeNoeudHachage(Reseau* R, TableHachage*H, double x, double y);
TableHachage* initialiseTablehachage(int m);
Reseau* reconstitueReseauHachage(Chaines* C, int M);
```

Troisième méthode : stockage par arbre quaternaire

On utilisera pour reconstituer le réseau la structure suivante :

```
#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
```

Les fonctions utilisées :

```
/* ~~~~~~~ Fonctions ajoutées pour un code modulaire / libération  ~~~~~~~ */

Noeud * creeNoeudH (Reseau *R ,double x , double y );
void liberer_ArbreQuat(ArbreQuat* aq);


/* ~~~~~~~ Fonctions principales ~~~~~~~ */

ArbreQuat* creerArbreQuat(double xc, double yc, double coteX,
double coteY);
void chaineCoordMinMax(Chaines* C, double* xmin, double*
ymin, double* xmax, double* ymax);
Noeud* rechercheCreeNoeudArbre(Reseau* R, ArbreQuat** a, ArbreQuat* parent, double x, double y) ;
Reseau* reconstitueReseauArbre(Chaines *C) ;
void insererNoeudArbre(Noeud* n, ArbreQuat** a, ArbreQuat* parent);
```

Comparaison des trois structures :

Comparaison des temps de calcul obtenus avec les trois structures de données utilisées pour tester l'existence d'un nœud dans un réseau.

Fonctions utilisées :

```
/* ~~~~~~~ Fonctions auxiliaires ~~~~~~~ */
double time_cpu(int start, int end);
double time_hachage_diff_sizes(Chaines* C);
double moy_struct(Chaines *c, int Meth, int M);



/* ~~~~~~~ Fonctions principales ~~~~~~ */
void time_diff_structures();
double time_HT_file(int M);
void comparaison_taille();
void time_LC_file();
void time_AQ_file();
```

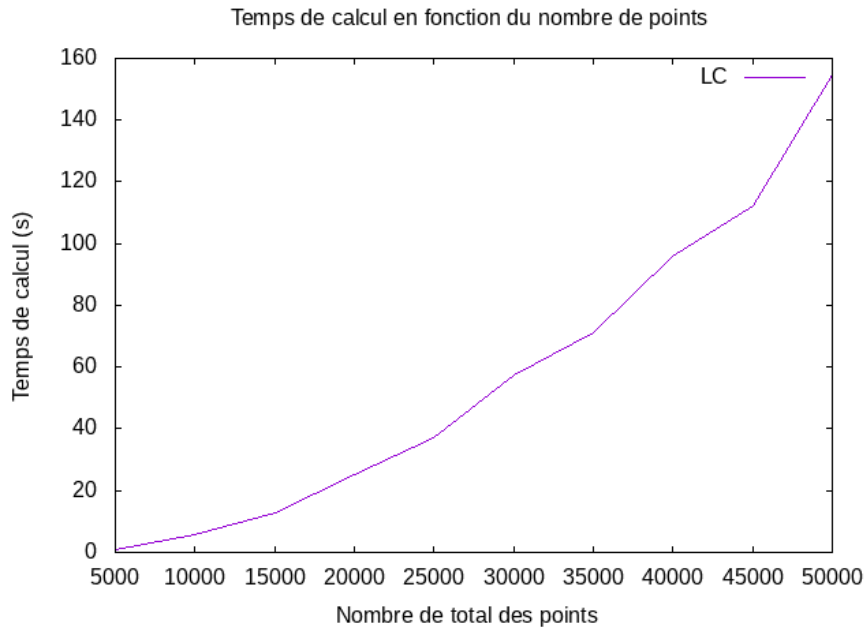
<u>Temps de calcul en fonction du nombre de points total des chaînes pour les trois structures</u> :

Remarque:

Les résultats sont obtenus en utilisant la fonction struct_moy qui calcule la moyenne sur plusieurs appels de reconstitueReseau(struct) pour avoir des résultats plus fiables.

En effectuant plusieurs appels et en prenant la moyenne des résultats, on obtient une estimation plus précise et plus fiable de la performance.

1. Liste chaînée :



- Le temps de calcul augmente rapidement avec le nombre total de points des chaînes, dépassant 150 secondes pour 50 000 points.

- Cette augmentation est bien plus que linéaire, suggérant une complexité supralinéaire de l'algorithme utilisant les listes chaînées.

- La complexité supralinéaire est due à la nécessité de parcourir linéairement les listes pour la recherche et insertion (rechercheCreeNoeud), ce qui devient de plus en plus coûteux à mesure que le nombre de points augmente.

2. Table de hachage et arbre quaternaire:

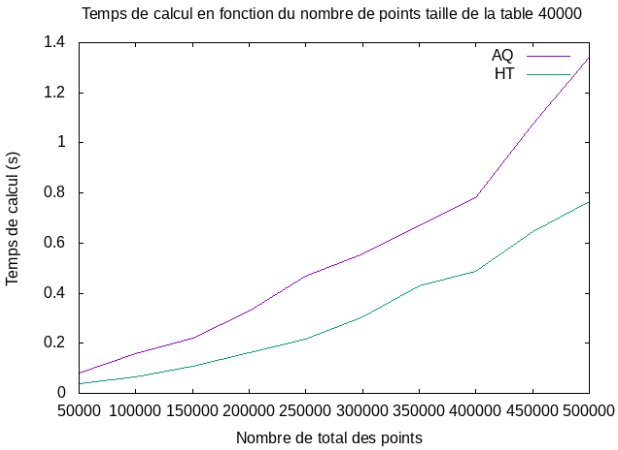


Photo 1: pour une table de taille 40000

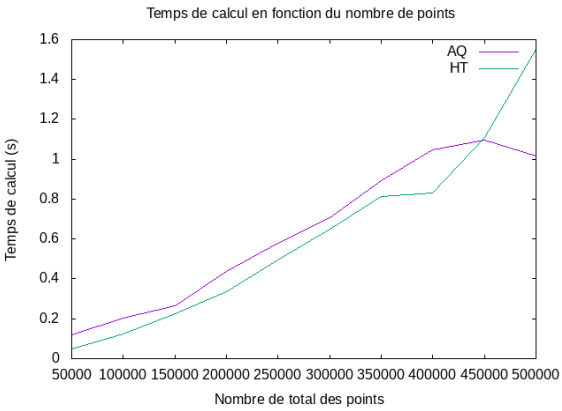


Photo 2: pour une table de taille 20000

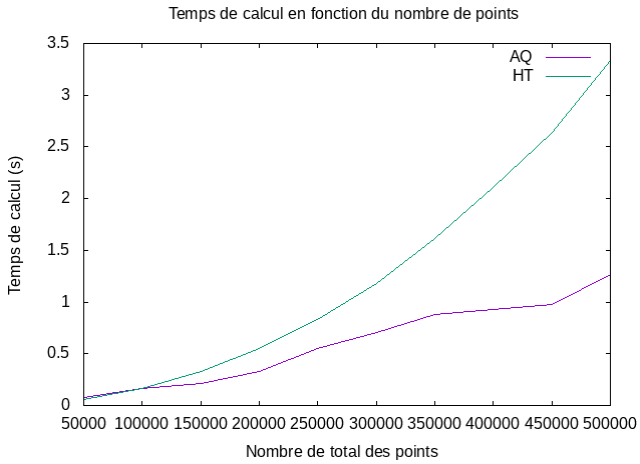
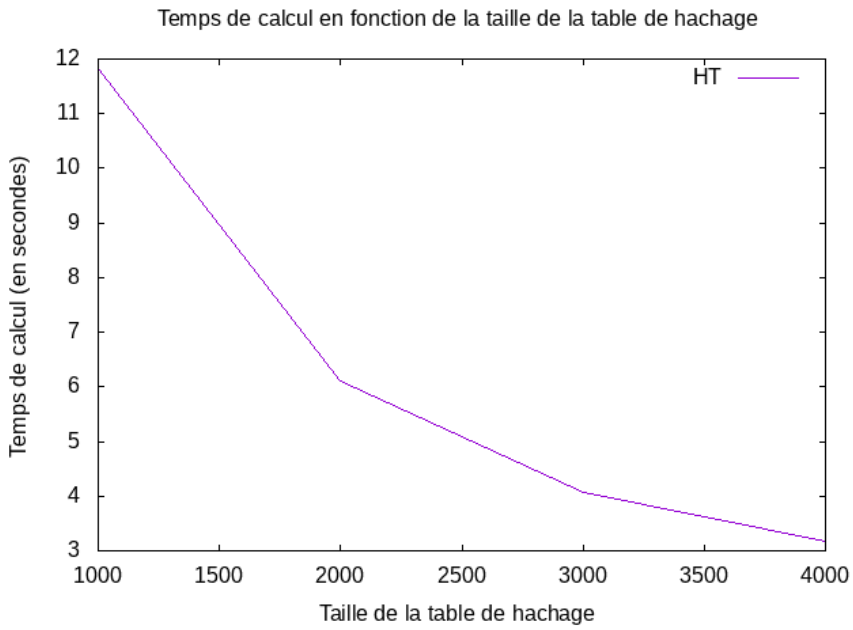


Photo 3: pour une table de taille 5000

a. Table de hachage:

Les temps de calcul varient en fonction de la taille de la table de hachage et du nombre de points.

- **<u>Variation du temps d'exécution ReconstitueReseauHachage en fonction de la taille de la table de hachage:</u>**



Le graphe montrent que le temps diminue à mesure que la taille de la table de hachage augmente.Une table de hachage plus grande réduit les collisions, diminue la charge de la table et maintient une complexité de recherche moyenne constante, ce qui accélère la recherche des éléments.

Analyse:

Pour une taille de table de hachage de 20000, les temps de calcul sont relativement faibles et augmentent de manière stable avec le nombre de points.La table de hachage dans ce cas est plus performante que l'arbre pour un nombre de points inférieur ou égal approximativement 450000 points et au-delà la l'arbre est plus performant que la table.(photo 2)

Explication:

La performance de la table de hachage peut diminuer par rapport à un arbre binaire lorsque le nombre de points devient très grand en raison de plusieurs facteurs. Les collisions dans la table de hachage peuvent devenir plus fréquentes et la complexité dans le pire cas sera en o(n) . De plus, la taille de la table de hachage doit être augmentée pour éviter les collisions, et dans le meilleur des cas chaque point prendra une case dans la table ce qui nous donne une complexité temporelle de o(1) cependant cela peut entraîner une utilisation inefficace de la mémoire. En revanche, un arbre quaternaire peut maintenir une structure équilibrée même avec un grand nombre de points, ce qui lui permet de conserver des performances stables.

Pour une taille de 40000, les temps de calcul sont encore plus bas, ce qui indique une amélioration des performances grace à la diminution des collision (photo 1). En revanche, pour une taille de table de hachage de 5000, les temps de calcul sont nettement plus élevés et augmentent de manière significative avec le nombre de points(photo3). Cela suggère que la

taille de la table de hachage est insuffisante pour gérer efficacement les données, entraînant davantage de collisions et des temps de calcul plus longs.

b.Arbres Quaternaires :

Les temps de calcul pour l'arbre quaternaire semblent relativement stables à mesure que le nombre de points augmente. Bien qu'ils soient généralement légèrement supérieurs à ceux de la table de hachage avec une taille de table de hachage de 20000, ils restent constants sur une large gamme de données. Cela indique que l'arbre quaternaire maintient des performances robustes indépendamment de la taille des données.

En résumé, pour de petits réseaux ou lorsque la taille du réseau est limitée, les listes chaînées peuvent offrir des temps de calcul optimaux. Cependant, lorsque le réseau devient plus important, la table de hachage peut offrir de meilleures performances lorsque sa taille est grande(on aura moins de collisions), tandis que les arbres quaternaires peuvent également être une option viable avec des temps de calcul relativement faibles et une meilleure gestion de la mémoire que les listes chaînées. La taille de la table de hachage joue également un rôle important dans les performances globales de la structure de données.

Deuxième partie :Optimisation du réseau

Le but de cette partie est d'optimiser l'utilisation des fibres optiques du réseau en minimisant la somme totale des longueurs des chaînes.

Le réseau dans cette partie est représenté par un graphe.

```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
```

Fonctions utilisées :

```c
//Fonctions pour la création d'un graphe
Sommet* creer_sommet(int num, double x, double y);
Cellule_arete* creer_L_voisins(int u, int v);
void ajouter_voisins(Sommet* s, Arete* a);
Graphe* creerGraphe(Reseau* r) ;


// Fonction d'affichage
void afficher_commod(Commod commod);
void afficher_arete(Arete* a);
void afficher_sommet(Sommet* s);
void afficher_L_voisins(Sommet* s);
void afficher_graphe(Graphe* G);
void afficherListeChainee(ListeChaine* liste) ;


//Fonctions pour gérer les listes

void ajouterEnTete(ListeChaine* liste, int valeur) ;
void libererListe(ListeChaine* liste) ;
void libererGraphe(Graphe* g) ;
int trouve_voisin(Cellule_arete* C,int num);



//Fonctions principales

int plusPetitNombreAretes(Graphe* G, int u, int v) ;

ListeChaine* cheminEntreDeuxSommets(Graphe* G, int u, int v) ;

int reorganiseReseau(Reseau* r) ;
```

Calcul du plus petit nombre d'arêtes d'une chaîne entre deux sommets et Stockage de l'arborescence des chemins et déduction de la chaîne de u à v en utilisant un parcours en largeur et un tableau:

```
plusPetitNombreAretes entre (4,5) =3 : Liste d'entiers (chemin) : 4 3 2 5
```

L'algorithme utilisé pour trouver le plus court chemin en nombre d'arcs (ou arêtes) entre deux sommets dans un graphe est un parcours en largeur à partir d'un sommet de départ. Ce parcours en largeur permet de connaître le nombre

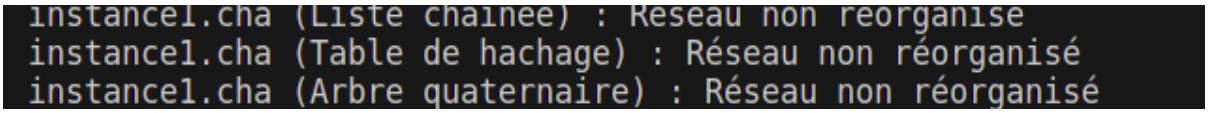
minimal d'arcs nécessaires pour atteindre tout sommet à partir du sommet de départ.

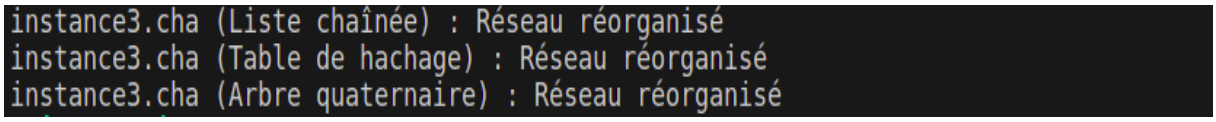
Le principe de cet algorithme est le suivant :

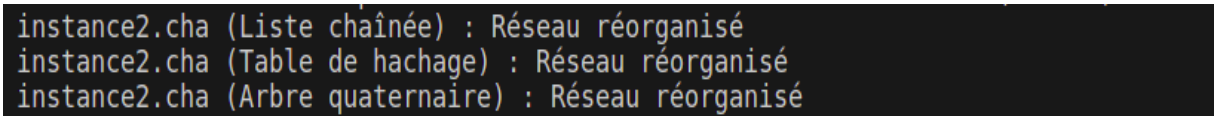
1. On initialise un tableau de distances D et un tableau des "pères" P.
    - D[r] est initialisé à 0 (où r est le sommet de départ) pour marquer la distance de r à lui-même.
    - P[r] est initialisé à -1 pour indiquer qu'il n'y a pas de prédécesseur de r.
2. Pour chaque nouveau sommet v ajouté à la file depuis son prédécesseur u, on met à jour D[v] = D[u] + 1 pour indiquer la distance entre r et v, et P[v] = u pour enregistrer le prédécesseur de v dans le chemin le plus court.
3. Pour retrouver le plus court chemin C pour atteindre un sommet v à partir de r, on remonte le tableau P en commençant par v.
    - On initialise u à v.
    - Tant que u n'est pas égal à -1 (c'est-à-dire qu'on n'a pas atteint le sommet de départ r), on ajoute u à l'ensemble C et on met à jour u à son prédécesseur P[u].
    - On répète ce processus jusqu'à ce qu'on atteigne le sommet de départ r.

Résultats de la fonction ReorganiseReseau

Enfin, dans la fonction `reorganiseReseau`, nous avons combiné plusieurs étapes, y compris la création du graphe à partir du réseau, le calcul des chemins les plus courts pour chaque commodité, et la vérification du nombre de chemins passant par chaque arête ce qui nous permet de déduire si un réseau est réorganiser ou pas . Cette fonctionnalité globale repose également sur l'algorithme BFS pour calculer les chemins les plus courts







Idées d'amélioration de la fonction ReorganiseReseau :

1) Unification des fonctionnalités : Au lieu d'avoir deux fonctions distinctes pour calculer le nombre minimal d'arêtes entre deux sommets et pour stocker le chemin correspondant, une approche unifiée pourrait être mise en place, permettant à une seule fonction d'exécuter ces deux tâches de manière efficace.

2)Consolidation des opérations : Au lieu d'avoir deux fonctions distinctes pour calculer le nombre minimal d'arêtes entre deux sommets et pour stocker le chemin correspondant, une approche unifiée serait plus efficace, permettant à une seule fonction de réaliser ces deux tâches de manière intégrée.

Combinaison de fonctionnalités : Actuellement, la fonction reorganiseReseau se limite à vérifier si le réseau est réorganisé, mais elle pourrait être étendue pour inclure la réorganisation effective du réseau en cas de besoin.

voici l'algormithme de reoganisation de réseau :

```
Fonction reorganiseReseau(reseau):
    Tant que le reseau n'est pas entierement connecte:
        Pour chaque paire de sommets non connectes dans le reseau:
            Calculer le nombre minimal d'aretes pour les connecteravec notre fonction
        Selectionner la paire de sommets avec le nombre minimal d'aretes
        Ajouter ces aretes au reseau
    Retourner le reseau reorganise
```