SIMA Daniel 3800992 DM Info/EEA Groupe 10
SADYKOVA Karima 28610100 DM Info/EEA Groupe 10

# Projet : Réorganisation d'un réseau de fibres optiques

Dans notre projet nous avons utilisé les différents types de structures de données permettant d'effectuer la reconstitution et la réorganisation d'un réseau de fibres optiques d'une agglomération. Dans le cadre du cours LU2IN006 - Structures de données, nous avons étudié comme structures de données: les listes chaînées, la table de hachage, les arbres, les files, les piles et les graphes.

En effet, nous avons appliqué en premier lieu la structure listes chaînées pour manipuler une instance de ces listes et ensuite pour stocker le réseau de fibres optiques avec tous ses nœuds et voisins.

Par la suite, on a utilisé la structure table de hachage pour stocker à nouveau le réseau d'une différente manière, la table de hachage porte sur les coordonnées des nœuds et le nombre de cases.

La troisième manière de stocker le réseau a été sous la forme d'un arbre quaternaire, c'est-à-dire un arbre où chaque nœud possède 4 fils. Cet arbre peut représenter dans un espace bidimensionnel une cellule rectangulaire qui est divisée en 4 sous-cellules rectangulaires. Chaque sous-cellule est identifiée à travers le centre de la cellule rectangulaire "parent" qu'on compare avec les coordonnées de chaque nœud du réseau de fibres optiques. Ces 4 sous-cellules représentent les parties Nord-Ouest, Nord-Est, Sud-Est et Sud-Ouest.

Finalement, nous avons utilisé un graphe qui a été créé à partir d'un réseau. Les nœuds étaient représentés par des sommets dans un tableau de pointeurs sur les sommets et les voisins sous la forme de cellules_artes. Une autre structure importante a été utilisée dans cette partie du projet pour permettre de faire un parcours en largeur, il s'agit de la file, qui va nous aider à récupérer des données du graphe.
Les listes chaînées, la table de hachage et les arbres vont être analysés dans l'exercice 6, ou l'on compare les temps de calcul obtenus de ces structures lors de la recherche de l'existence d'un nœud dans le réseau.

* Les commandes peuvent être compilées avec Makefile qui est joint au dossier.

SIMA Daniel 3800992 DM Info/EEA Groupe 10
SADYKOVA Karima 28610100 DM Info/EEA Groupe 10

**Explication courte de notre travail sur chaque exercice**

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

Question 1.1)

- Voir fichier "Chaines.c" (code l. 263) pour le code de la fonction **Chaines* lectureChaine(FILE *f).** C'est une fonction permettant d'allouer, de remplir et de retourner une instance de la structure Chaines a partir d'une fichier .cha préalablement ouvert.
  Cette fonction utilise des fonctions définies auparavant comme **creer_Chaines()** (code l.61), **creer_CellChaine()** (code l.29), **inserer_en_tete_CellPoint()** (code l.76) et **inserer_en_tete_CellChaine()** (code l.117). Ces fonctions, comme leurs nom l'indique, créent ou insèrent en en tête.

Question 1.2)

- Voir fichier "Chaines.c" (code l. 309) pour le code de la fonction **void ecrireChaine(Chaines *C, FILE *f).** C'est une fonction permettant l'écriture de la structure Chaines C dans un fichier f préalablement ouvert.
  Cette fonction utilise une fonction définie auparavant qui est **nombre_points()** (code l.44) qui permet de retourner le nombre de points d'une CellChaine.

Jeux d'essais :

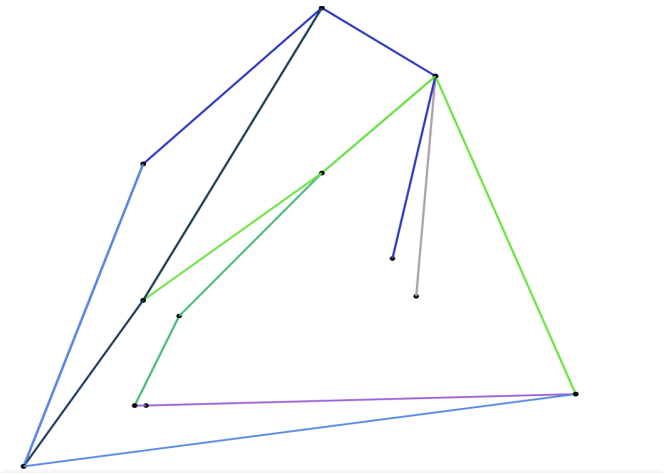
- Voir le fichier "ChaineMain.c". Lors de l'exécution on attend un deuxième argument (le nom du fichier à tester ".cha"), qui va être lu et écrit dans le dossier "Test_ChaineMain". Pour tester le bon fonctionnement des fonctions **lecture_chaine()** et **ecrire_Chaines()** ainsi que les fonctions qu'elles utilisent, vous pouvez comparer le fichier ".cha" que vous avez exécuté et le fichier avec le même nom mais avec l'extension ".txt" se trouvant dans le dossier "Test_ChaineMain" qui doivent être les mêmes en respectant les remarques de l'énoncé (a une exception près, on a ajouté des [] pour bien différencier chaque point).
  De plus, il existe une fonction **affichage_Chaines()** (code l.193) qui va afficher le fichier "00014_burma.cha" car c'est le seul à pouvoir être affiché dans le terminal grâce à sa taille plus petite.

Question 1.3)

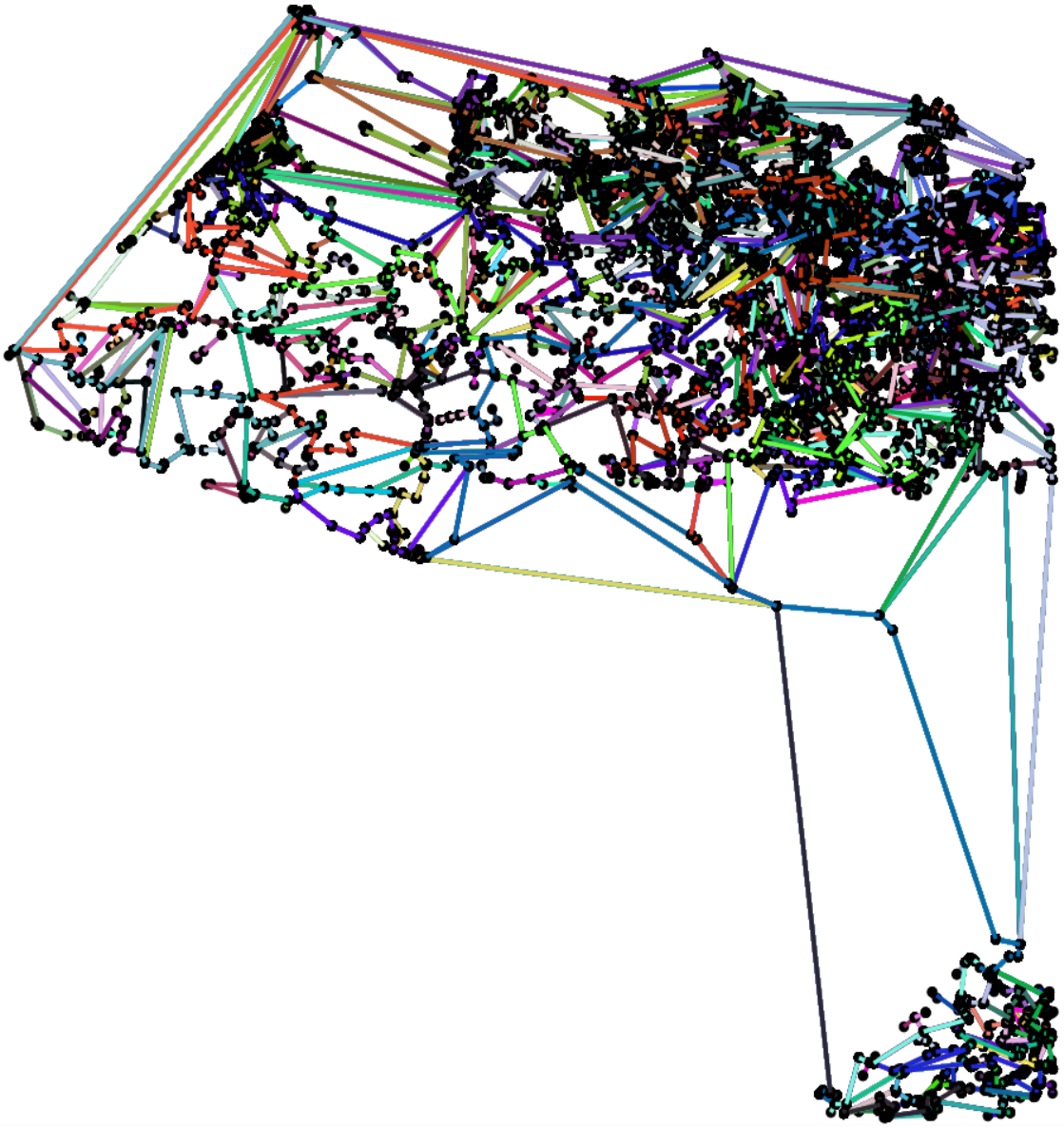
- Voir le fichier "ChaineMain.c". On pourrait également vérifier le bon fonctionnement des fonctions des questions précédentes en regardant l'affichage SVG.
  Ce dernier se trouve également dans le dossier "Test_ChaineMain" avec le même nom mais avec l'extension ".html".
- Voir ci-dessous l'affichage SVG des trois fichiers .cha:

SIMA Daniel 3800992 DM Info/EEA Groupe 10
SADYKOVA Karima 28610100 DM Info/EEA Groupe 10

Affichage SVG du fichier 00014_burma.cha:

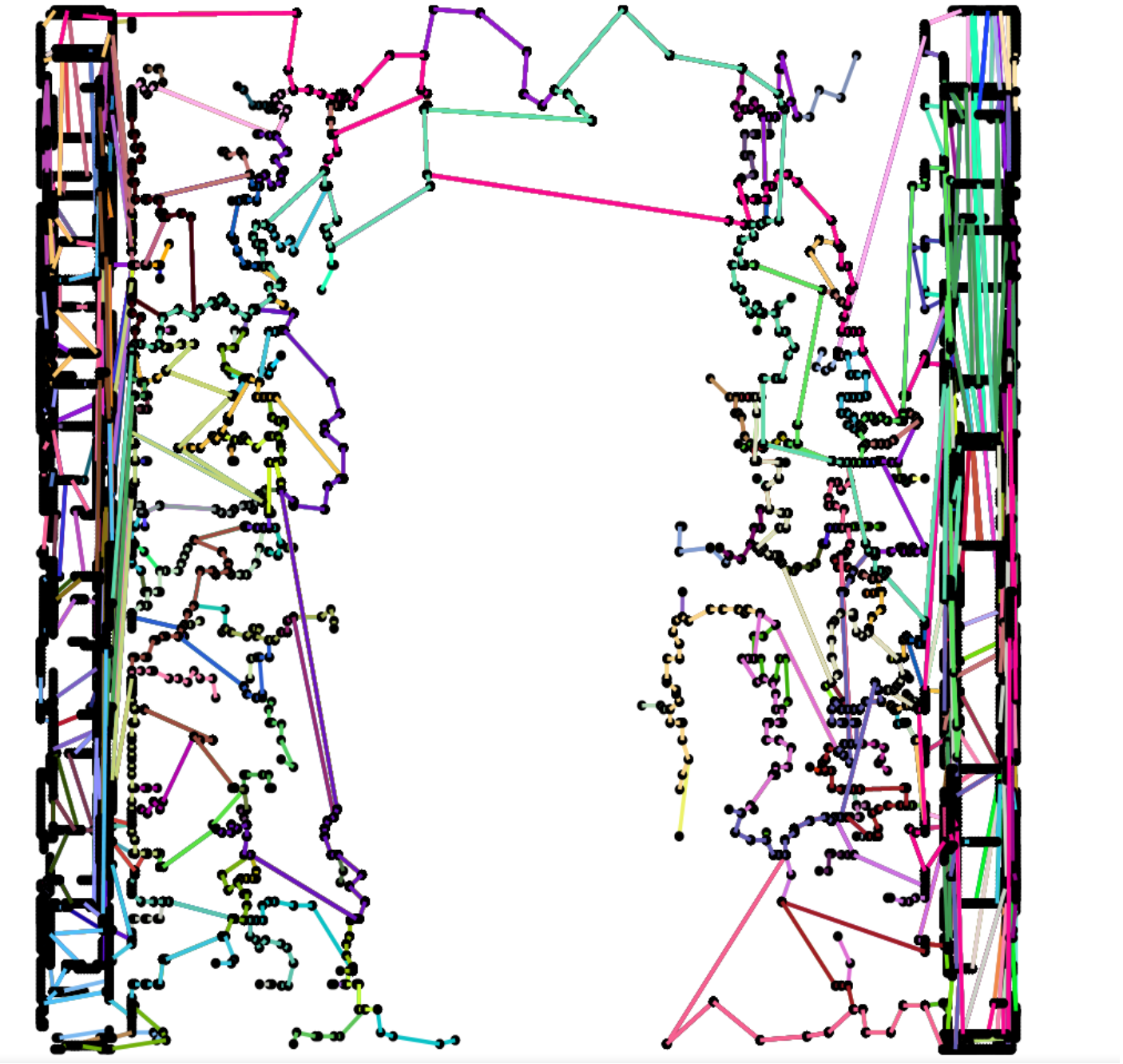


Affichage SVG du fichier 05000_USA-road-d-NY.cha:



SIMA Daniel 3800992 DM Info/EEA Groupe 10
SADYKOVA Karima 28610100 DM Info/EEA Groupe 10

Affichage SVG du fichier 07397_pla.cha:



Question 1.4)

- Voir fichier “Chaine.c” (code l.383 et l.408) pour les fonctions **double longueurChaine(CellChaine *c)** et **double longueurTotale(Chaines *C)**. Ces fonctions calculent la longueur physique d’une CellChaine, respectivement la longueur physique de toutes les CellChaines de la structure Chaines C. Dans la première fonction on a utilisé la fonction **pow()** originaire de la bibliothèque “math.h”.

Jeux d'essais :

- La fonction **double longueurChaine(CellChaine *c)** est utilisé dans la fonction **double longueurTotale(Chaines *C)** et cette dernière est testé dans le fichier "ChaineMain.c" en calculant la longueur physique totale du fichier .cha exécuté.

Question 1.5)

- Voir fichier "Chaine.c" (code l.426) pour la fonction **int comptePointsTotal(Chaines *C)**. Cette fonction compte le nombre total de points de la structure Chaines C. Elle utilise une autre fonction définie auparavant qui s'appelle **int nombre_points(CellChaine *cellChaine)** et qui renvoie le nombre de points pour une CellChaine.

Jeux d'essais:

La fonction **int comptePointsTotal(Chaines *C)** est testé dans le fichier "ChaineMain.c" en comptant le nombre total de points du fichier .cha exécuté.

**Exercice 2 – Premiere ḿethode : stockage par liste chaînée**

Question 2.1)

- Voir fichier "Reseau.c" (code l.154) pour la fonction **Noeud* rechercheCreeNoeudListe(Reseau *R, double x, double y)**. Cette fonction retourne un Noeud du Réseau correspondant au point (x, y) s'il existe ou le crée sinon et l'ajoute dans le Réseau. Elle utilise des fonction définies auparavant comme **Noeud *creer_Noeud(int num, double x, double y)** (code l.9) et **CellNoeud *creer_CellNoeud(Noeud *nd)** (code l.26) , qui comme leur nom l'indiquent créent des Noeuds et CellNoeuds.

Question 2.2)

- Voir fichier "Reseau.c" (code l.183) pour la fonction **Reseau* reconstitueReseauListe(Chaines *C).** Cette fonction retourne le réseau à partir de la structure Chaines C. Elle utilise des fonction définies auparavant comme **Reseau *creer_Reseau(int nbNoeuds, int gamma)** (code l. 58) qui permet de créer le Réseau, **Noeud* rechercheCreeNoeudListe(Reseau *R, double x, double y)** de la question précédente, ce qui permettra de justifier son bon fonctionnement ainsi que celui des fonctions qu'elle utilise lors des jeux d'essais, **void inserer_voisins(CellNoeud **CN, Noeud *n)** (code l.75) qui permet l'insertion du Noeud n dans la liste des voisins du CellNoeud CN (passage par référence) et la fonction **void inserer_commodite_fin(Reseau *R, CellCommodite *com)** (code l.98) qui insère une Cell Commodité à la fin du Réseau R.

Question 2.3)

- Voir le fichier “ReconstitueReseau.c” qui correspond au fichier de jeux d’essais de la fonction **Reseau* reconstitueReseauListe(Chaines *C)**, **Reseau* reconstitueReseauHachage(Chaines *C, int M)** et **Reseau* reconstitueReseauArbre(Chaines* C)**. Ceci permettra de montrer leur bon fonctionnement ainsi que celui des fonctions qu’elle utilise. Les résultats de ces fonctions se trouvent dans le dossier “Test_ReconstitueReseau” avec l'extension “.txt” et “.html” en fonction de la méthode utilisée et le fichier “.cha” utilisé.

- Dans ce main, nous avons utilisé l'instruction switch case qui fait appel aux fonctions selon le cas choisi. Pour manipuler le réseau nous avons 3 types de structures.
- Afin d'exécuter le main et choisir la méthode, l’utilisateur doit saisir pour le deuxième argument le nom du fichier de la lecture et pour le troisième argument le numéro de la méthode (1 - Listes, 2 - Hachage, 3 - Arbre)
- Pour chaque méthode choisie par l'utilisateur, le programme fait appel à la fonction spécifique de reconstitution du réseau qui dépend de la méthode:
    - **reconstitueReseauListe(C)**, **reconstitueReseauHachage(C, 15)**, ou **reconstitueReseauArbre(C)**

**Exercice 3 – Manipulation d’un réseau**

Question 3.1)

- Voir le fichier “Reseau.c” (code l.220 et l.237) pour les fonctions **int nbCommodites(Reseau *R)** et **int nbLiaisons(Reseau *R)**. Ces fonctions retournent le nombre de CellCommodites et respectivement de liaisons du Reseau.

Question 3.2)

- Voir le fichier “Reseau.c” (code l.257) pour la fonction **void ecrireReseau(Reseau *R, FILE *f)**. Cette fonction permet l’affichage du réseau dans un fichier au même format que le fichier “00014burma.res”. Pour écrire cette fonction, on utilise les deux fonctions définies dans la question précédente, ce qui permettra également de justifier leur bon fonctionnement lors des jeux d’essais.

Question 3.3)

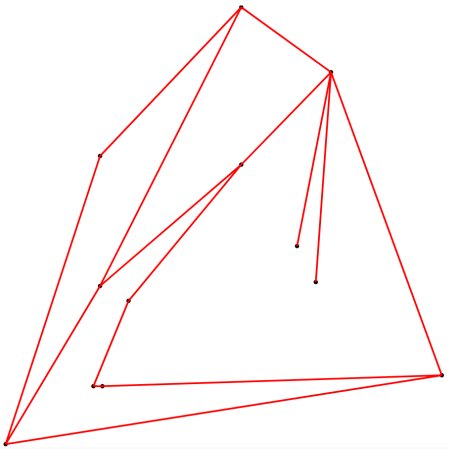
- Voir le fichier “Reseau.c” (code l.338) ou la fonction **void afficheReseauSVG(Reseau*R, char* nomInstance)** a été ajoutée.

Jeux d’essais:

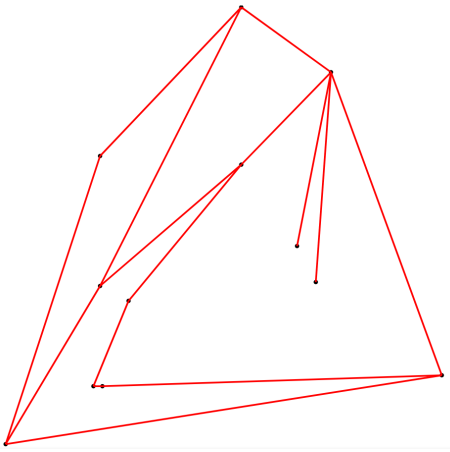
- La fonction **void ecrireReseau(Reseau *R, FILE *f)** est testé dans le fichier “ReconstitueReseau.c” lors de l'écriture des réseaux pour les 3 structures de données différentes. Cela peut se vérifier en

exécutant le fichier “ReconstitueReseau.c” avec pour deuxième argument un fichier .cha et pour troisième argument la méthode choisie (voir Question 2.3). Le fichier résultat est le fichier “AffichageListe.txt”, “AffichageHachage.txt” ou “AffichageArbre.txt” (selon la méthode choisie) se trouvant dans le dossier “Test_ReconstitueReseau”. Ceci permettra de montrer le bon fonctionnement de cette fonction ainsi que celui des fonctions qu’elle utilise. Les résultats de ces fonctions se trouvent dans le dossier “Test_ReconstitueReseau” avec l'extension “.txt” et “.html” en fonction de la méthode utilisée et le fichier “.cha” utilisé.

Affichage SVG du fichier 00014_bruma.cha pour trois structures de données différentes:



Liste Chainée



Arbre Quaternaire

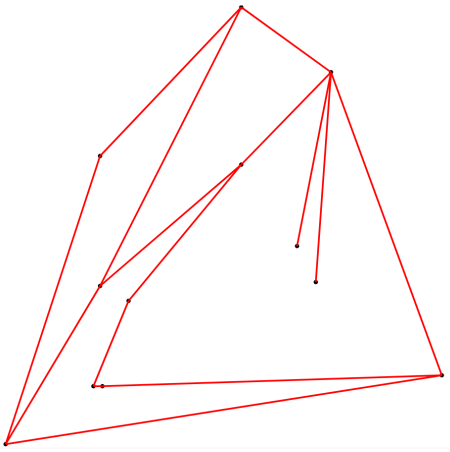
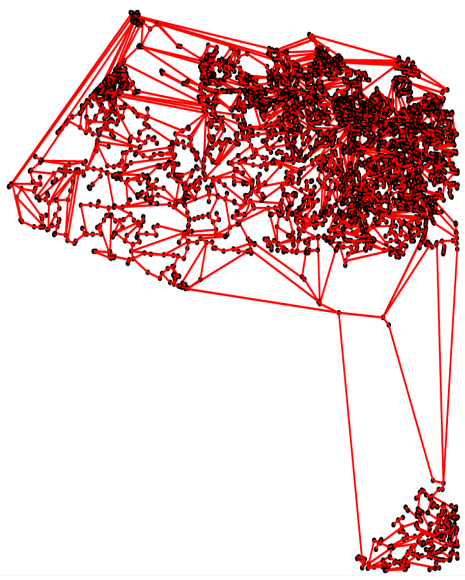
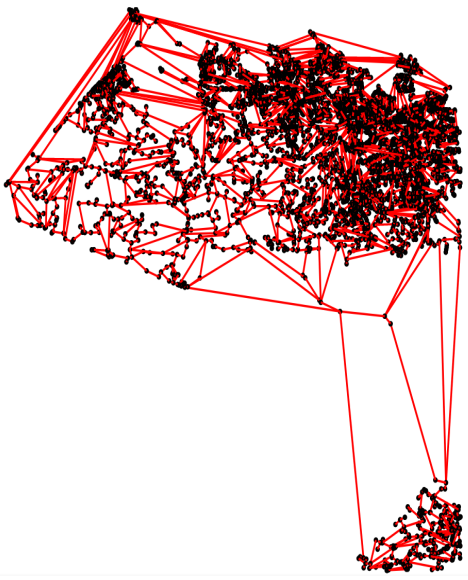


Table de Hachage

Affichage SVG du fichier 05000_USA-road-d-NY.cha pour trois structures de données différentes:



Liste Chainée



Arbre Quaternaire

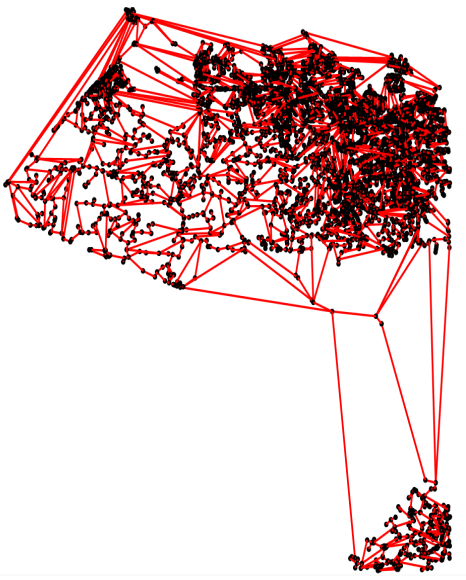
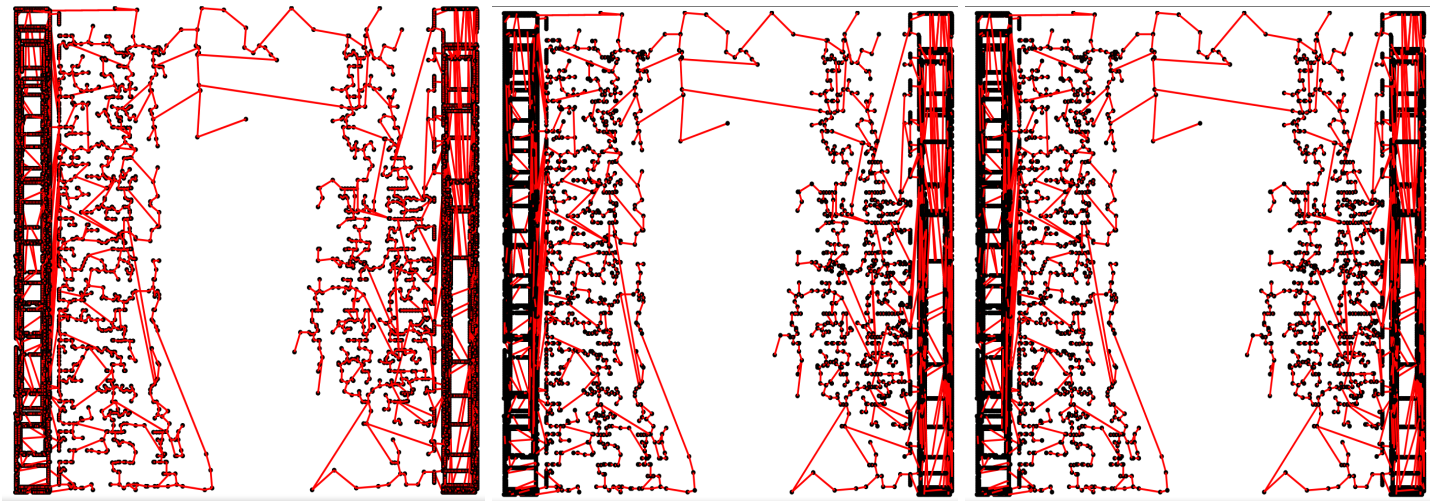


Table de Hachage

SIMA Daniel 3800992 DM Info/EEA Groupe 10
SADYKOVA Karima 28610100 DM Info/EEA Groupe 10

Affichage SVG du fichier 07397_pla.cha pour trois structures de données différentes:



Liste Chainée Arbre Quaternaire Table de Hachage

- On en déduit que le réseau est le même pour toutes les trois structures pour chaque fichier ".cha".
- Comme la manière de donner les numéros des noeuds pour la fonction qui utilise les listes chaînées est différente des fonctions qui utilisent la table de hachage et arbre quaternaire (même manière pour ces deux dernières), les chemins apparaissent dans un autre ordre pour la les affichages SVG de la liste chaînée, d'où une légère différence.

**Exercice 4 – Deuxième méthode : stockage par table de hachage**

Question 4.1)

- Voir le fichier "Hachage.h", ou la structure TableHachage possède deux attributs, le premier pour la taille de la table de hachage (nombre de cases de celle-ci) et un second correspondant à un tableau de listes de CellNoeuds.

Question 4.2)

- Voir le fichier "Hachage.c" (code l.10) pour la fonction **int fonctionClef(double x, double y)**. Cette fonction retourne la clef de la table de hachage qui porte sur les coordonnées (x, y) d'un point. Elle est donnée par la formule f(x,y) =y+ (x+y)(x+y+ 1)/2. Pour des valeurs de x et y allant de 1 à 10, la fonction clef est appropriée car elle semble donner des valeurs différentes pour toute combinaison de valeurs x et y.

Question 4.3)

- Voir le fichier "Hachage.c" (code l.14) pour la fonction **int fonctionHachage(int cle, int m).** Cette fonction donne la fonction de hachage, **h(k) = ⌊M(kA − √⌊kA⌋)⌋**, qui porte sur la clef k

donnée par les coordonnées, la taille M de la table de hachage et une constante A = $\frac{\sqrt{5}-1}{2}$.

On pourrait utiliser des fonctions de hachage portant sur la distance physique par rapport à un point ou portant sur le numéro du point…

Question 4.4)

- Voir fichier "Hachage.c" (code l.20) pour la fonction **Noeud* rechercheCreeNoeudHachage(Reseau* R, TableHachage*H, double x, double y).**
  Cette fonction retourne un Noeud du Réseau correspondant au point (x, y) s'il existe dans la Table de Hachage ou le créer sinon et l'ajoute dans celui-ci et dans le Réseau.
  Elle utilise des fonction définies auparavant comme **int fonctionClef(double x, double y)** et **int fonctionHachage(int cle, int m)** pour déterminer la case ou l'on doit chercher le Noeud ou l'introduire. Ceci permet également de justifier le bon fonctionnement de ces deux fonctions lors des jeux d'essais. En plus de ces fonctions on utilise des fonction définies dans le fichier "Reseau.c" et qui ont déjà été testées comme **Noeud *creer_Noeud(int num, double x, double y)** et **CellNoeud *creer_CellNoeud(Noeud *nd)**.

Question 4.4)

- Voir fichier "Hachage.c" (code l.80) pour la fonction **Reseau* reconstitueReseauHachage(Chaines *C, int M).**Cette fonction retourne le Réseau à partir de la structure Chaines C et la future taille de la Table de Hachage. Elle utilise des fonctions définies auparavant comme **Reseau *creer_Reseau(int nbNoeuds, int gamma), inserer_voisins(CellNoeud **CN, Noeud *n)** et **void inserer_commodite_fin(Reseau *R, CellCommodite *com)** qui ont été déjà définies et testées dans l'exercice précédent.
  On utilise également la fonction **Noeud* rechercheCreeNoeudHachage(Reseau* R, TableHachage*H, double x, double y)** de la question précédente, ce qui permettra de justifier son bon fonctionnement lors des jeux d'essais. Finalement on utilise une fonction de libération de la Table de Hachage, une fois le Réseau totalement reconstitué, et qui se nomme **void liberer_table_hachage(TableHachage *H)** (code l.52).

Jeux d'essais:

- Voir fichier "HachageMain.c", ou l'on teste la fonction **Reseau* reconstitueReseauHachage (Chaines *C, int M)** avec M=15 sur les fichiers ".cha" mis en deuxième argument lors de l'exécution. L'on peut vérifier le bon fonctionnement de cette fonction en comparant le fichier résultat qui se trouve dans le dossier "Test_HachageMain" et qui se nomme de la même manière mais avec l'extension ".txt" et le fichier avec extension ".res". L'affichage SVG sous format html se trouve également dans ce dossier sous le même nom que le fichier exécuté mais avec l'extension ".html". Cette fonction est également testé dans "ReconstitueReseau.c" en utilisant en deuxième argument un

fichier .cha à tester et en troisième argument le chiffre 2 qui désigne la méthode de la Table de Hachage.

**Exercice 5 – Troisieme methode : stockage par arbre quaternaire**

Question 5.1)

- Voir fichier “ArbreQuat.c” (code l.11) pour la fonction **void chaineCoordMinMax(Chaines* C, double* xmin, double*ymin, double* xmax, double* ymax)**. Cette fonction calcule les coordonnées minimales et maximales x et y des points de la structure Chaines C.

Question 5.2)

- Voir fichier “ArbreQuat.c” (code l.42) pour la fonction **ArbreQuat* creerArbreQuat(double xc, double yc, double coteX,double coteY)**. Cette fonction crée un Arbre Quaternaire.

Question 5.3)

- Voir fichier “ArbreQuat.c” (code l.66) pour la fonction **void insererNoeudArbre(Noeud* n, ArbreQuat** a, ArbreQuat*parent)**. Cette fonction insère un Noeud dans l’Arbre Quaternaire, l’insertion se fait récursivement selon les cas qui sont divisés en trois cas: Cas de l’Arbre Vide, Cas Feuille et Cas Cellule Interne.
  Pour réaliser cette fonction, on utilise la fonction **ArbreQuat* creerArbreQuat(double xc, double yc, double coteX,double coteY)** qui permettra de créer un Arbre Quaternaire dans le Cas de l’Arbre Vide. Cela permettra également de montrer le bon fonctionnement de cette fonction lors des jeux d’essais.

Question 5.4)

- Voir fichier “ArbreQuat.c” (code l.150) pour la fonction **Noeud* rechercheCreeNoeudArbre(Reseau* R, ArbreQuat** a, ArbreQuat*parent, double x, double y)**. Cette fonction retourne un Noeud du Réseau correspondant au point (x, y) s’il existe dans l’Arbre Quaternaire ou le crée sinon et l’ajoute dans celui-ci et dans le Réseau. Elle se déroule selon trois cas qui sont: Cas de l’Arbre Vide, Cas Feuille et Cas Cellule Interne (ou l’on utilise la récursivité de la fonction).
  Cette fonction utilise des fonctions définies auparavant, deux d’entre elles étant déjà écrites et testées dans le fichier “Reseau.c”, ce sont: **Noeud *creer_Noeud(int num, double x, double y)** et **CellNoeud *creer_CellNoeud(Noeud *nd)**. La fonction **void insererNoeudArbre(Noeud* n, ArbreQuat** a, ArbreQuat*parent)** de la question précédente est également utilisé ici pour insérer le noeud dans l’Arbre si ce dernier n’est pas présent dans l’Arbre. Ceci permettra de démonter son bon fonctionnement lors des jeux d’essais.

SIMA Daniel 3800992 DM Info/EEA Groupe 10
SADYKOVA Karima 28610100 DM Info/EEA Groupe 10

---

Question 5.5)

- Voir fichier “ArbreQuat.c” (code l.237) pour la fonction **Reseau* reconstitueReseauArbre(Chaines* C)**. Cette fonction retourne le Reseau à partir de la structure Chaines C. Elle utilise des fonctions définies auparavant, dont **Reseau *creer_Reseau(int nbNoeuds, int gamma), inserer_voisins(CellNoeud **CN, Noeud *n)** et **void inserer_commodite_fin(Reseau *R, CellCommodite *com)** qui ont été déjà définies et testées dans l’exercice les fichiers “Reseau.c” et “ReconstitueReseau.c”. D’autres fonctions définies dans le fichier courant ont été également utilisées, comme **void chaineCoordMinMax(Chaines* C, double* xmin, double*ymin, double* xmax, double* ymax)**, **ArbreQuat* creerArbreQuat(double xc, double yc, double coteX,double coteY)** et **Noeud* rechercheCreeNoeudArbre(Reseau* R, ArbreQuat** a, ArbreQuat*parent, double x, double y)** des questions précédentes de cet exercice. Ceci permettra également de justifier leur bon fonctionnement lors des jeux d'essais.
  Une fonction de libération de l’Arbre Quaternaire, **void liberer_arbre_quat(ArbreQuat* A)** (code l.222), est également utilisée à la fin une fois le Reseau reconstitué.

Jeux d’essais:

- Voir fichier “ArbreQuatMain.c”, ou l’on teste la fonction **Reseau* reconstitueReseauArbre(Chaines* C)**. L’on peut vérifier le bon fonctionnement de cette fonction en exécutant ce fichier avec un fichier “.cha” en deuxième argument et en comparant le résultat des affichages .txt et .html se trouvant dans le dossier “Test_ArbreQuatMain” avec le fichier .res du fichier .cha exécuté en deuxième argument et en observant le SVG.

**Exercice 6 – Comparaison des trois structures**

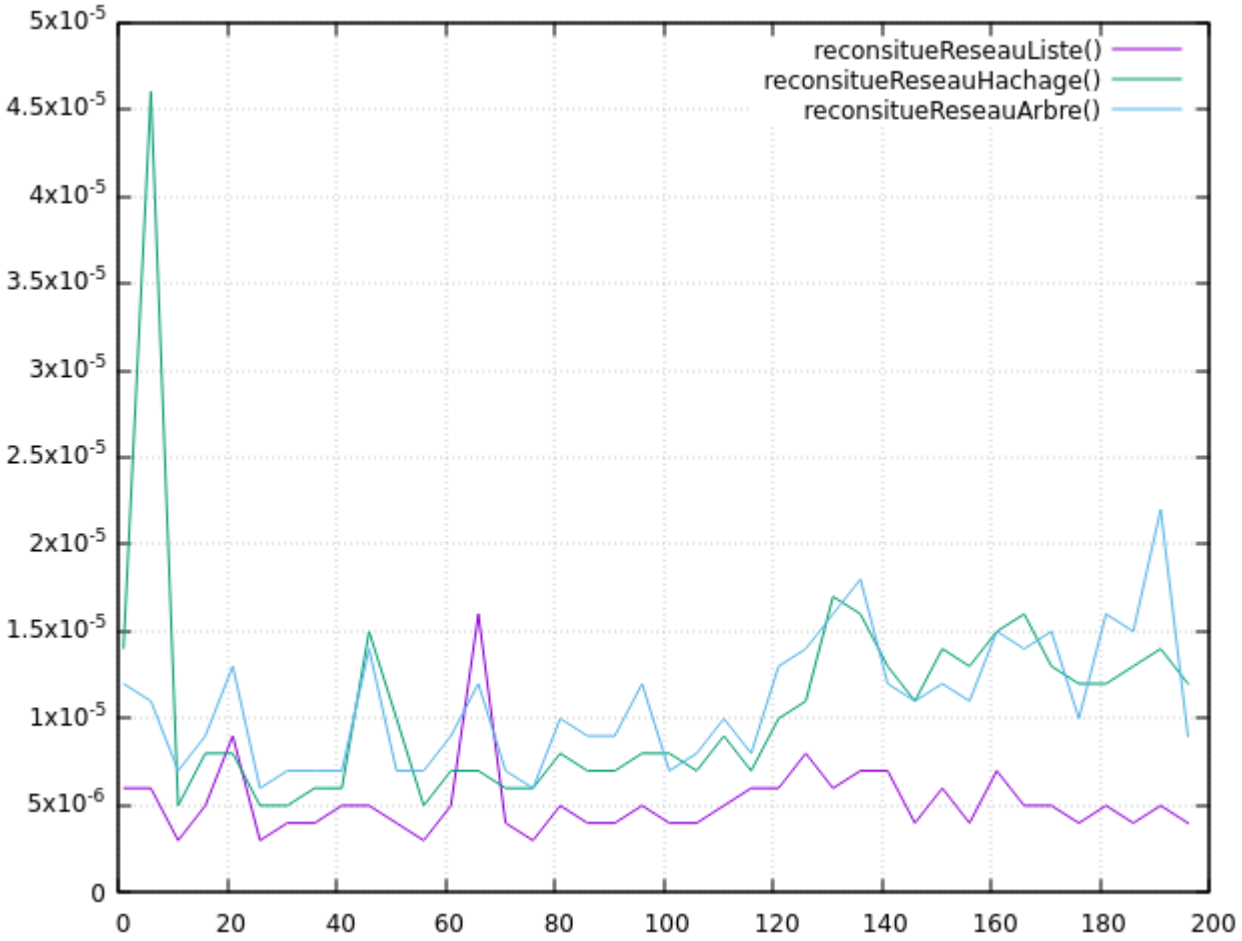
Question 6.1) et Question 6.4)

- Voir la première partie du fichier “MainTemps.c”, où l'on exécute les 3 fonctions de reconstitution du Réseau avec le fichier “.cha” mis en deuxième argument lors de l'exécution. Les fichiers contenant les temps se trouvent dans le dossier “Sortie_vitesses”. A noter que dans cette partie on fait la moyenne sur 2 pour chaque exécution pour avoir des valeurs plus appropriées de la réalité.

  PS: Si vous voulez exécuter cette première partie du fichier, arrêtez le terminal quand ce message apparaîtra: “"=== Passage à la deuxième partie ===” car sinon cela risque de prendre énormément de temps (c’est déjà légèrement le cas).

SIMA Daniel 3800992 DM Info/EEA Groupe 10
SADYKOVA Karima 28610100 DM Info/EEA Groupe 10

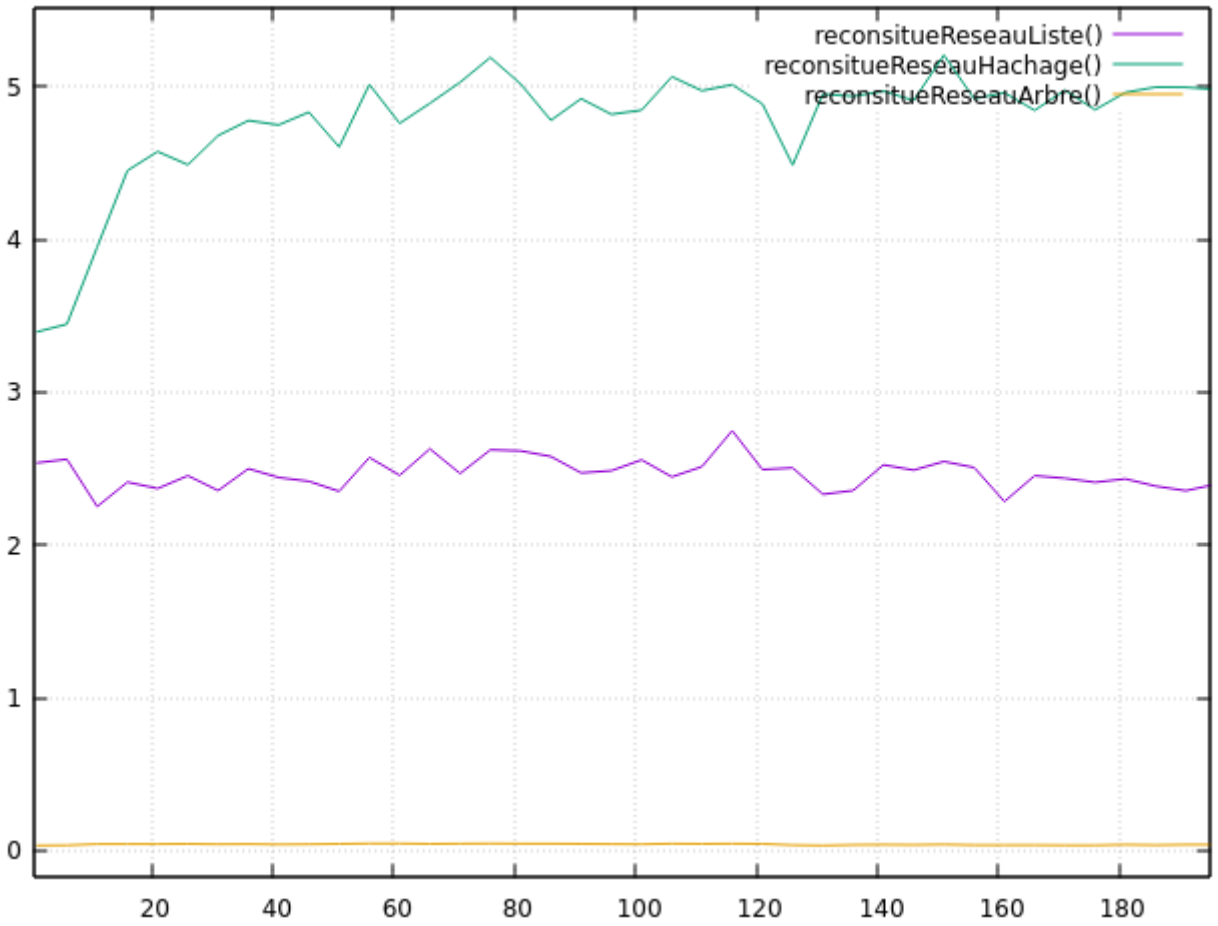
Graphe montrant l'évolution du temps mis par le CPU en fonction de la taille de la Table de Hachage pour les 3 fonctions de reconstitutions du Réseau pour le fichier 00014_burma.cha:




- Ce graphique présente des temps très courts (de l'ordre de des microsecondes) pour toutes les trois structures de données utilisées. Ceci est parfaitement normal et s'explique notamment car le fichier "00014_burma.cha" ne possède que 12 nœuds. L'axe des abscisses correspond au nombre de cases dans la Table de Hachage (paramètre M), ceci est nécessaire seulement pour la fonction qui utilise la Table de Hachage et n'est pas pertinent pour les Listes Chainées et l'Arbre Quaternaire d'ou leurs temps plutôt constant.
  A ce niveau, les variations de temps (quelques microsecondes) sont très petites et ne permettent pas d'aboutir à une réelle conclusion.

SIMA Daniel 3800992 DM Info/EEA Groupe 10
SADYKOVA Karima 28610100 DM Info/EEA Groupe 10

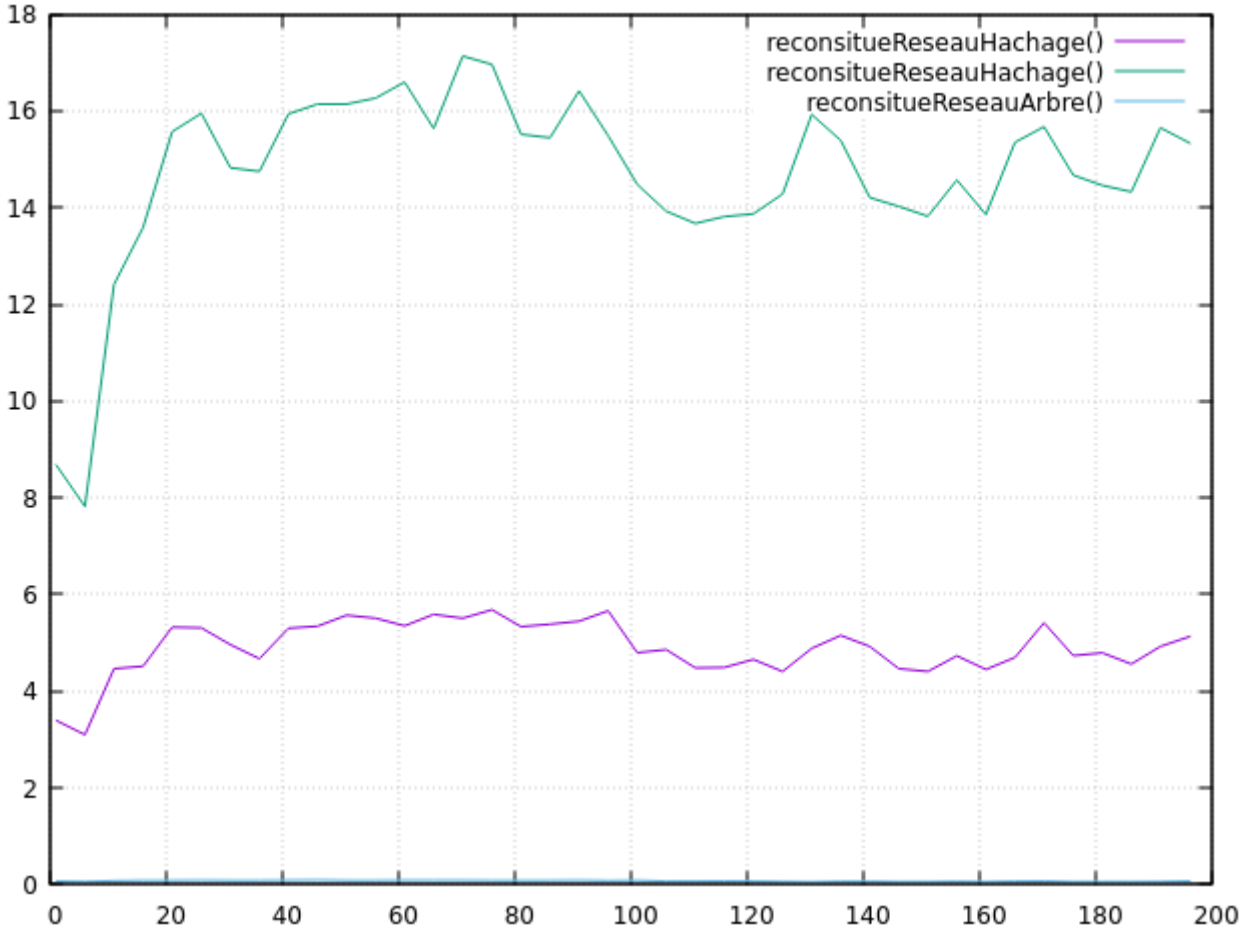
Graphe montrant l'évolution du temps mis par le CPU en fonction de la taille de la Table de Hachage pour les 3 fonctions de reconstitutions du Réseau pour le fichier 05000_USA-road-d-NY.cha:




- Ce graphique présente des temps plus grands que le précédent (de l'ordre des secondes) car cette fois le nombre de Noeuds du fichier "05000_USA-road-d-NY.cha" est bien plus important (4504 Nœuds au total). L'axe des abscisses correspond au nombre de cases dans la Table de Hachage (paramètre M), ceci est nécessaire seulement pour la fonction qui utilise la Table de Hachage et n'est pas pertinent pour les Listes Chainées et l'Arbre Quaternaire d'où leurs temps plutôt constant.
  Le temps de calcul pour l'Arbre Quaternaire est bien inférieur (de l'ordre des millisecondes) aux deux autres structures (de l'ordre des secondes), ce qui est normal et s'explique à travers les comparaisons effectuées dans la fonction **rechercheCreeNoeudArbre()** (elle permet d'aller bien plus rapidement à l'endroit qui nous intéresse et ne présente pas de boucle). La fonction qui utilise les Listes Chaînées possède des temps plus petits que celle qui utilise la Table de Hachage car le nombre de Nœuds est encore assez petit pour montrer un réel contraste entre l'efficacité de ces deux structures de données.
  Les temps sont plus grands pour la fonction qui utilise la Table de Hachage (différence de l'ordre de 2 secondes) que les temps pour la fonction qui utilise les Liste Chainées. Cependant au début les fonctions qui utilisent ces deux structures de données semblent avoir des temps proches, ce qui se

justifie à travers le petit nombre de cases de la Table de Hachage au début et donc une ressemblance avec la Liste Chaînée.

Graphe montrant l'évolution du temps mis par le CPU en fonction de la taille de la Table de Hachage pour les 3 fonctions de reconstitutions du Réseau pour le fichier 07397_pla.cha:




- Ce graphique ressemble fortement au graphique obtenu grâce au fichier “05000_USA-road-d-NY.cha”. Le nombre de Noeuds est de 6429 pour ce fichier (une augmentation d’environ 2000 Noeuds comparé au fichier précédent). Cette différence se traduit sur le graphique avec une augmentation des temps de calcul pour toutes les trois structures de données, l'augmentation la plus importante est celle de la Table de Hachage suivie de la Liste Chaînée et enfin avec une légère augmentation visible sur le fichier “07397_pla_temps.txt” vient l’Arbre Quaternaire.
  Les remarques du graphique précédent s’appliquent également pour ce graphique.

SIMA Daniel 3800992 DM Info/EEA Groupe 10
SADYKOVA Karima 28610100 DM Info/EEA Groupe 10

---

Question 6.2)

- Voir le fichier “Chaine.c” (code l.444), ou la fonction **Chaines* generationAleatoire(int nbChaines,int nbPointsChaine,int xmax,int ymax)** a été écrite. Cette fonction retourne une structure Chaines C généré avec des données aléatoires selon les paramètres de la fonction.
  Elle utilise des fonctions définies auparavant, comme **Chaines *creer_Chaines(int gamma, int nbChaines)**, **CellChaine *creer_CellChaine(int numero)**, **void inserer_en_tete_CellPoint(CellChaine *cellChaine, double x, double y)** et **void inserer_en_tete_CellChaine(Chaines *chaines, CellChaine *cellChaine)**. Ces fonctions ont déjà été écrites et testées dans “Chaine.c”.
  De plus, on a utilisé la fonction **rand()** définie dans bibliothèque **<time.h>** pour générer des nombres aléatoires selon la formule int i = rand()%(MAX-MIN)+MIN.

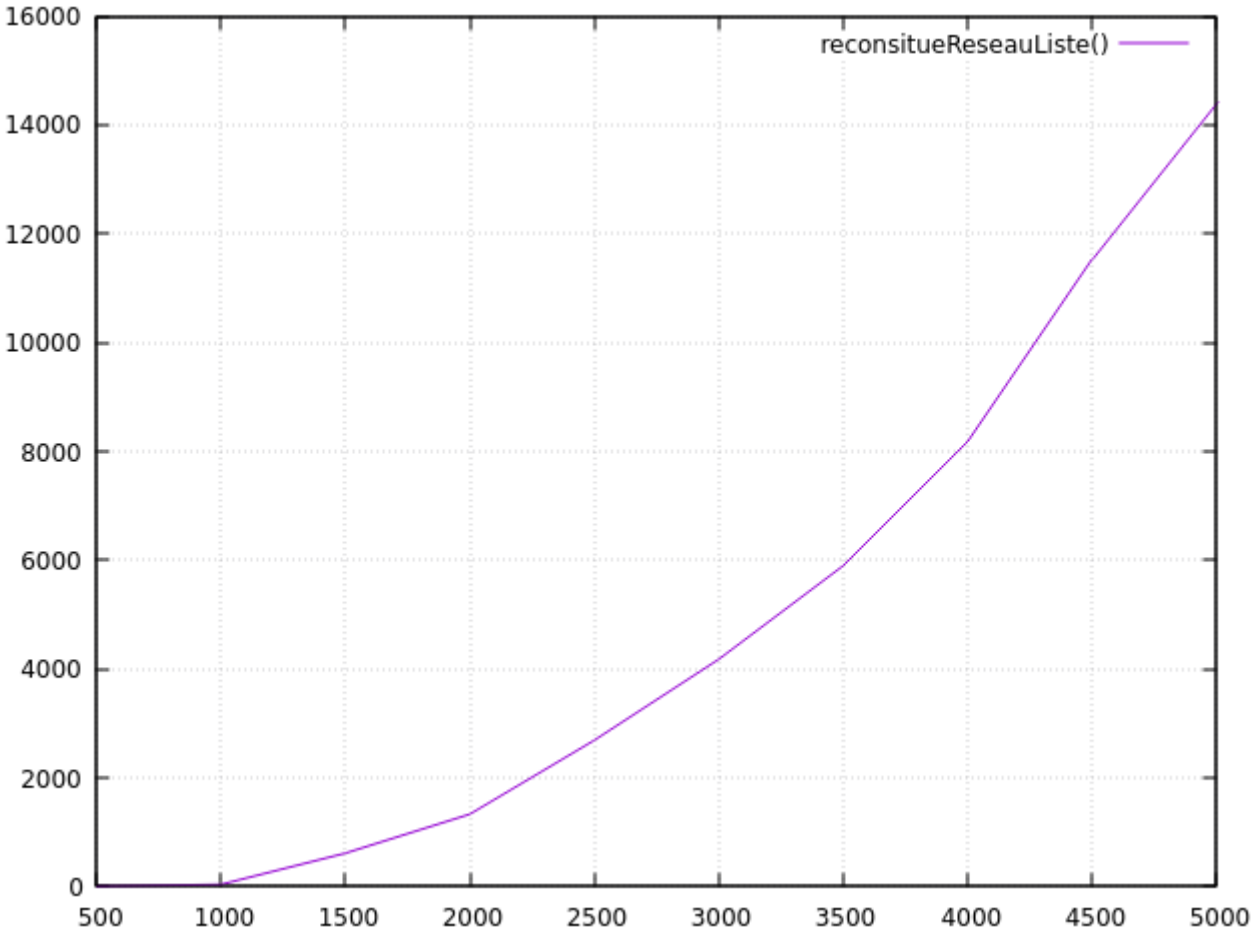
Question 6.3) et Question 6.4)

- Voir la deuxième partie du fichier “MainTemps.c”, où l'on exécute les trois fonctions de reconstitution du Réseau avec la structure Chaines C retourné par la fonction **Chaines* generationAleatoire(int nbChaines,int nbPointsChaine,int xmax,int ymax)** de la question précédente pour un nombre de chaînes allant de 500 à 5000 avec 100 points par chaîne et avec des coordonnées maximales pour x et y définies dans les directives préprocesseur. Cela prouve le bon fonctionnement de cette fonction.
  Les fichiers résultats avec les temps ainsi que les deux graphes se trouvent dans le fichier “Sortie_vitesses” sous le nom de “GraphiqueListeChainees.txt”, “GraphiqueArbreHachage.txt”, “GraphiqueListeChainees.png” et “GraphiqueArbreHachage.png”

  PS: L'exécution risque de prendre énormément de temps, comme vous pouvez le voir sur les graphes ci-dessous. Cependant l'évolution de la boucle est observable sur le terminal a travers des prints.

SIMA Daniel 3800992 DM Info/EEA Groupe 10
SADYKOVA Karima 28610100 DM Info/EEA Groupe 10

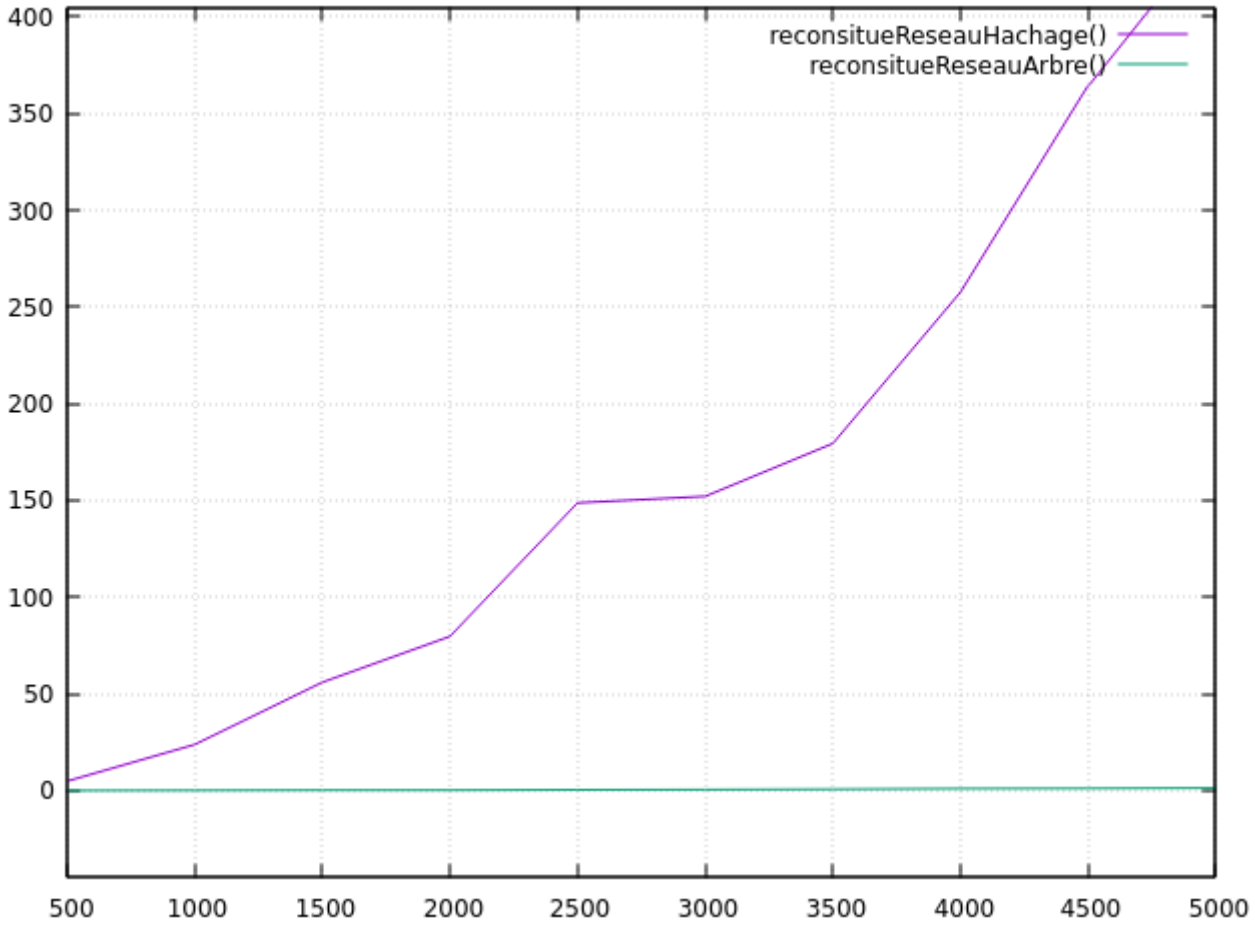
Graphe montrant l'évolution du temps mis par le CPU en fonction du nombre de chaînes pour la fonction reconsititueReseauListe() qui utilise les Listes Chaînées:




- Ce graphique montre bien que la fonction **reconstitueReseauListe()**, qui utilise pour structure de données les Listes Chaînées possède un temps de calcul exponentiel à partir de 1000 chaînes (c'est-à-dire 1000*100=100 000 Noeuds). On arrive aux alentours de 4h de temps de calcul pour 5000 chaînes (c'est-à-dire 5000*100= 500 000 Noeuds).
  Donc cette structure de données n’est pas adaptée à partir du seuil de 100 000 Noeuds. Avant ce seuil, les temps de calcul sont aux alentours de quelques dizaines de secondes.

SIMA Daniel 3800992 DM Info/EEA Groupe 10
SADYKOVA Karima 28610100 DM Info/EEA Groupe 10

Graphe montrant l'évolution du temps mis par le CPU en fonction du nombre de chaînes pour les fonctions de reconstitution de Réseau utilisant la Table de Hachage et l'Arbre Quaternaire:




- Ce graphique nous montre que le temps de calcul mis par la fonction **reconsitueReseauArbre()** est bien plus avantageux que celui mis par la fonction **reconsisuteReseaHachage()**. Ce dernier croît au fur et à mesure que le nombre de chaînes augmente (donc le nombre de Noeuds) pour atteindre 400 secondes (environ 7 minutes) pour 5000 chaines (500 000 Noeuds) alors qu'au début il était aux alentours de quelques secondes.
  De l'autre côté, la fonction qui utilise l'Arbre Quaternaire met un temps constant proche de 0 secondes. Il était d'environ quelques millisecondes au début pour atteindre environ 1 seconde pour 5000 chaines (500 000 Noeuds).

Conclusion:

- Ces deux graphes mettent en évidence deux choses importantes. La première chose étant que la fonction qui utilise la structure de données de l'Arbre Quaternaire est de loin la plus efficace. Deuxièmement, la fonction qui utilise la Table de Hachage est réellement plus efficace seulement à partir de 1000 chaînes (100 000 Noeuds), c'est à partir de ce moment que la fonction qui utilise les Liste Chainees voit son temps de calcul croître exponentiellement.

On préférera donc l'Arbre Quaternaire dans ce contexte.

*Les détails des temps de calcul de tous ces graphes sont présents dans le dossier "Sortie_vitesses" sous format ".txt".

**Exercice 7 – Parcours en largeur**

Question 7.1)

- Voir le fichier "Graphe.c" (code l.162), pour la fonction **Graphe* creerGraphe(Reseau* r)**. Cette fonction crée un Graphe à partir d'un réseau passé en paramètres. Elle utilise des fonctions définies auparavant, comme **Commod creer_Commod(int e1, int e2)** (code l.60), **Sommet *creer_Sommet(int num, double x, double y)** (code l.43) et **void creer_L_voisin(Cellule_arete **tabC, Sommet *som, CellNoeud *cn)** (code l.71), qui créent un Commod, un Sommet et la dernière permet de remplir la liste des voisins (L_voisins) d'un Sommet. Cette dernière utilise un tableau de listes de Cellules_aretes pour éviter d'allouer plusieurs fois une même Arête.
  Le bon fonctionnement de ces trois fonctions sera justifié lors des jeux d'essais, ou l'on a besoin de créer un Graphe.

Question 7.2) et Question 7.3)

- Voir le fichier "Graphe.c" (code l.247), pour la fonction **ListeEntier nb_min_arretes_chaine(Graphe *G, int s1, int s2)**. Cette fonction retourne le chemin (ListeEntier) entre les sommets s1 et s2. Elle utilise des fonctions fournies dans les bibliothèques "Struct_File.h" et "Struct_Liste.h", comme **void Init_file(S_file *f)** et **void Init_Liste(ListeEntier *L)** qui initialise une S_file et une ListeEntier, **void ajoute_en_tete(ListeEntier* L, int u)** qui ajoute un int en tete de la ListeEntier, **void enfile(S_file * f, int donnee)** qui ajoute un int en fin de S_file, **int defile(S_file *f)** qui supprime le premier élément de la S_file et le retourne. Enfin on utilise une fonction de libération pour la structure ListeEntier, qui est **void desalloue(ListeEntier *L)**.

Question 7.4)

- Voir le fichier "Graphe.c" (code l.371), pour la fonction **int reorganiseReseau(Reseau* r)**. Cette fonction retourne 1 (vrai) si le nombre de chaînes qui passe par chaque arête du graphe $< \gamma$, retourne 0 (faux) sinon. Elle utilise des fonctions définies auparavant ou fournies dans la bibliothèque "Struct_Liste.h", comme **Graphe* creerGraphe(Reseau* r)** et **ListeEntier nb_min_arretes_chaine(Graphe *G, int s1, int s2)** définies dans les questions précédentes. Enfin on utilise également des fonctions de libération, **void desalloue(ListeEntier *L)** et **void liberer_Graphe(Graphe *G)** (code l.143) qui libère la structure Graphe en utilisant à son tour une fonction définie dans ce fichier, qui est **void liberer_Sommet(Sommet *som)** (code l.134) et qui libère un Sommet en passant par la fonction **void liberer_Cell_arete(Cellule_arete *cellarete)** (code

l.119) qui libère des Cellules_aretes. Ceci va permettre de monter leur bon fonctionnement lors des jeux d'essais.

Question 7.5)

- Voir fichier "GrapheMain.c", dans lequel on exécute avec en deuxième argument le fichier ".cha". On teste ici la fonction **int reorganiseReseau(Reseau* r)** ce qui permettra de justifier son bon fonctionnement ainsi que le bon fonctionnement des fonctions qu'elle utilise a travers les affichages sur le terminal.

  On utilisant donc cette fonction pour les trois instances fournies, on trouve:
  - Pour "00014_burma.cha" => La fonction dit qu'il existe au moins une arête ayant un nombre de chaînes supérieur ou égal à gamma. Ceci est vrai après affichage de du Graphe (possible que pour ce fichier, car nombre petit de Noeuds). Pour voir l'affichage de ce fichier décommenter la l.435 du fichier "Graphe.c".

  - Pour "05000_USA-road-d-NY.cha" => La fonction dit qu'il existe au moins une arête ayant un nombre de chaînes supérieur ou égal à gamma.

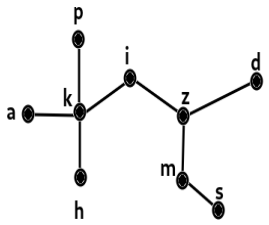
  - Pour "07397_pla.cha" => La fonction dit qu'il existe au moins une arête ayant un nombre de chaînes supérieur ou égal à gamma.

- Pour améliorer cette fonction nous proposons d'afficher la matrice sommet-sommet, pour les graphes avec un nombre pas très grand de Noeuds. De même, on pourrait retourner la/les arêtes qui ont un nombre de chaînes supérieur ou égal à gamma, pour pouvoir les modifier plus tard.

* Les fichiers .txt qui sont créés suite aux fonctions ecriture_Chaines() et ecrireReseau() permettent de comparer le bon fonctionnement de ces fonctions, malgré un numéro différent pour les Nœuds et un ordre différent pour les liaisons et/ou commodités. L'énoncé nous a contraint à faire ainsi…
** Dans tous les fichiers source il existe des fonctions en plus de l'énoncé qu'on a laissé parce qu'elles fonctionnent et peuvent être adaptées dans ce contexte.

SIMA Daniel 3800992 DM Info/EEA Groupe 10
SADYKOVA Karima 28610100 DM Info/EEA Groupe 10

# Explication stockage arborescence




/* Initialisaion */

arete[k]=k
arete[p]=p
arete[a]=a
arete[h]=h
arete[i]=i
arete[z]=z
arete[m]=m
arete[s]=s
arete[d]=d

arborescence[k]=k
arborescence[p]=p
arborescence[a]=a
arborescence[h]=h
arborescence[i]=i
arborescence[z]=z
arborescence[m]=m
arborescence[s]=s
arborescence[d]=d

/* Quand la S_file est vide*/

arete[k]=k
arete[i]=k->i,
arete[p]=k->p
arete[a]=k->a
arete[h]=k->h
arete[z]=i->z
arete[m]=z->m
arete[s]=m->s
arete[d]=z->d

arborescence[k]=k
arborescence[p]=p
arborescence[a]=a
arborescence[h]=h
arborescence[i]=i
arborescence[z]=z
arborescence[m]=m
arborescence[s]=s
arborescence[d]=d

/* A l'entrée du parcorus de tous les Sommets du Graphe*/

arborescence[k]=k, deja ok
arborescence[p]=p, traitement =>arborescence[p]=k->p ok
arborescence[a]=a, traitement =>arborescence[a]=k->a ok
arborescence[h]=h, traitement =>arborescence[h]=k->h ok
arborescence[i]=i, traitement =>arborescence[i]=i->p ok
arborescence[z]=z, traitement =>arborescence[z]=i->p pas encore ok
arborescence[m]=m, traitement =>arborescence[m]=z->m pas encore ok
arborescence[s]=s, traitement =>arborescence[s]=m->p pas encore ok
arborescence[d]=d, traitement =>arborescence[d]=z->p pas encore ok

Traitement des cas pas ok

arborescence[z]=i->p, traitement => arborescence[z]=k->i->p ok
arborescence[m]=z->m, traitement => arborescence[m]=i->z->m pas encore ok
arborescence[s]=m->p, traitement => arborescence[s]=z->m->p pas encore ok
arborescence[d]=z->p, traitement => arborescence[d]=i->z->p pas encore ok

Donc arborescence[i] donne le chemin allant de la racine (ici k) à i.

arborescence[s]=i->z->m->p, traitement => arborescence[s]=k->i->z->m->p ok

Traitement des cas pas ok

Traitement des cas pas ok

arborescence[m]=i->z->m, traitement => arborescence[m]=k->i->z->m ok
arborescence[s]=z->m->p, traitement => arborescence[s]=i->z->m->p pas encore ok
arborescence[d]=i->z->p, traitement => arborescence[d]=k->i->z->p ok

- Les sommets du graphe (les lettres) sont en réalité des numéros.