Yanis SAADI DIT SAADA 21101885
Ines HARRAOUI 21204796

UE : LU2IN006
Groupe 3

Compte rendu

Projet : Réorganisation d'un réseau de fibres optiques

Sommaire

Ce projet consiste à reconstituer un réseau, tout d'abord grâce à une liste chaînée, puis un tableau de hachage, ensuite un arbre quaternaire tout en affichant les données de chaque chaîne du réseau dans un fichier texte. Suite à la manipulation de ces trois structures, une comparaison des temps de calcul pour chacune d'elles est effectuée. Enfin, sera construit un graphe de sommets représentant un réseau de nœuds…

Chaine.c, Chaine.h contiennent un ensemble de fonctions manipulant des chaînes, des fonctions de lecture et d'écriture visant au stockage des informations d'une structure de chaînes.

ChaineMain.c contient la récupération des données de la structure de chaînes contenue dans le fichier "00014_burma.cha" en appelant les fonctions de lecture et l'écriture.

Reseau.c, Reseau.h contiennent un ensemble de fonctions manipulant des listes chaînées, dans le but de reconstituer un réseau sous forme d'une liste chaînée.

Hachage.c, Hachage.h contiennent un ensemble de fonctions manipulant une table de hachage, dans le but de reconstituer un réseau composé d'un tableau de hachage.

ArbreQuat.c, ArbreQuat.h contiennent un ensemble de fonctions manipulant des arbres quaternaires, dans le but de reconstituer un réseau sous forme d'un arbre quaternaire..

ReconstitueReseau.c appelle les fonctions de reconstitution du réseau pour chacune des 3 structures en écrivant ses informations dans les fichiers "burmaliste.txt", "burmahachage.txt" et "burmaarbre.txt".

CompareTemps.c compare les temps d'exécution des fonctions de reconstitution du réseau pour chacune des 3 structures dans le fichier "temps_de_calcul.txt" mais, cette fois-ci, en reconstituant une structure de milliers de chaînes!

Graphe.c, Graphe.h contiennent un ensemble de fonctions manipulant des graphes, dans le but de reconstituer un réseau sous forme d'un graphe.

Voici les algorithmes des fonctions de recherche ou de création de noeuds, propres à chaque structure :

- **rechercheCreeNoeudListe**(*Reseau *R, double x,double y*) de complexité de *Θ(n)*, recherche un nœud dans le réseau à partir de ses coordonnées avec une liste chaînée et son algorithme :

1. Initialise un pointeur de cellule de nœud à la tête du réseau R (*noeudR = R->noeud*s).
2. Tant que *noeudR* n'est pas NULL, exécuter les étapes suivantes :
3. Si les coordonnées x et y du nœud courant sont égales aux coordonnées x et y fournies en paramètres, alors :
4. Retourner le nœud courant.
5. Passer au nœud suivant dans le réseau.
6. Fin du tant que
7. Si aucun nœud correspondant n'a été trouvé dans le réseau :
8. Ajouter un nouveau nœud en tête du réseau avec les coordonnées x et y fournies, en incrémentant le nombre de nœuds du réseau (R->nbNoeuds).
9. Retourner le nouveau nœud ajouté (R->noeuds->nd).

- **rechercheCreeNoeudHachage(Reseau* R, TableHachage* H, double x, double y)** de complexité de *Θ(n)* recherche un nœud dans le réseau avec une table de hachage et son algorithme :

1. Calculer la position du nœud en calculant la clé grâce aux coordonnées *x* et *y* en utilisant f, la fonction qui calcule la clé de hachage puis en calculant h, la fonction de hachage de la table donnée.
2. Récupérer la cellule de nœud à la position pos dans la table de hachage (*cour = H->T[pos]*).
3. Tant que la cellule de nœud cour n'est pas NULL, exécuter les étapes suivantes :
4. Si les coordonnées du nœud courant correspondent aux coordonnées du noeud recherché, alors :
5. Retourner le nœud courant.
6. Passer à la cellule de nœud suivante dans la table de hachage.
7. Fin du tant que.
8. Si aucun nœud correspondant n'a été trouvé dans la table de hachage :
9. Créer un nouveau nœud avec les coordonnées fournies, en utilisant la fonction *creerNoeud*.
10. Créer une nouvelle cellule de nœud pour le nœud créé.
11. Ajouter cette cellule de nœud en tête de la liste chaînée à la position pos dans la table de hachage.
12. Ajouter une nouvelle cellule de nœud pour le nœud créé en tête de la liste chaînée des nœuds du réseau.
13. Incrémenter le nombre de nœuds du réseau.
14. Retourner le nœud créé.

- **rechercheCreeNoeudArbre(Reseau* R, ArbreQuat** a, ArbreQuat* p, double x, double y)** de complexité de *O(n)* et *Ω(log(n))* recherche un nœud dans le réseau avec un arbre quaternaire et son algorithme :

1. Si l'arbre est vide, alors :
2. Ajouter un nouveau nœud aux chaînes du réseau R avec les coordonnées fournies, en utilisant la fonction *ajout_teteCellNoeud*.
3. Incrémenter le nombre de nœuds du réseau.
4. Sinon, si l'arbre *a est une feuille (contient déjà un nœud), alors :
5. Si les coordonnées du nœud de l'arbre *a correspondent aux coordonnées fournies en paramètres, alors :
6. Retourner ce nœud.
7. Sinon, retourner NULL (pas de nœud correspondant).
8. Sinon (l'arbre **a* est un nœud interne et ne contient pas encore de nœud) :
9. Déterminer les coordonnées du centre de l'arbre.
10. Si les coordonnées fournies sont dans le quadrant sud-ouest *(x* < *xc* et *y* < *yc*), alors :
11. Appeler récursivement la fonction *rechercheCreeNoeudArbre* avec l'arbre sud-ouest comme nouvel arbre, et **a* comme parent.
12. Sinon, si les coordonnées sont dans le quadrant sud-est (*x* >= *xc* et *y* < *yc*), alors :
13. Appeler récursivement la fonction *rechercheCreeNoeudArbre* avec l'arbre sud-est *(*a)->se* comme nouvel arbre, et **a* comme parent.
14. Sinon, si les coordonnées sont dans le quadrant nord-ouest *(x* < *xc* et *y* >= *yc*), alors :
15. Appeler récursivement la fonction *rechercheCreeNoeudArbre* avec l'arbre nord-ouest *(*a)->no* comme nouvel arbre, et *a comme parent.
16. Sinon (les coordonnées sont dans le quadrant nord-est), alors :
17. Appeler récursivement la fonction *rechercheCreeNoeudArbre* avec l'arbre nord-est *(*a)->ne* comme nouvel arbre, et *a comme parent.
18. Fin du sinon.

reconstitueReseauListe(Chaines *C)

1. Créer un réseau vide en utilisant la fonction *creerReseau*.
2. Initialiser une liste de commodités à NULL.
3. Pour chaque chaîne :
    1. Initialiser *extrA* et *extrB* à NULL.
    2. Parcourir chaque point de la chaîne :
        1. Rechercher ou créer un nouveau nœud dans la liste de nœuds du réseau pour le point courant avec *rechercheCreeNoeudListe*.
        2. Si un nœud précédent existe (*V*), insérer des voisins entre le nœud précédent et le nœud courant avec de la fonction *insererVoisins*.
        3. Mettre à jour *V* avec le nœud courant.
        4. Si le point actuel est le dernier de la chaîne, mettre à jour *extrB* avec le nœud correspondant.
    3. Si une commodité entre *extrA* et *extrB* n'existe pas déjà, l'ajouter à la liste de commodités.
4. Assigner la liste de commodités au réseau.
5. Retourner le réseau créé.

reconstitueReseauHachage(Chaines *C, int M)

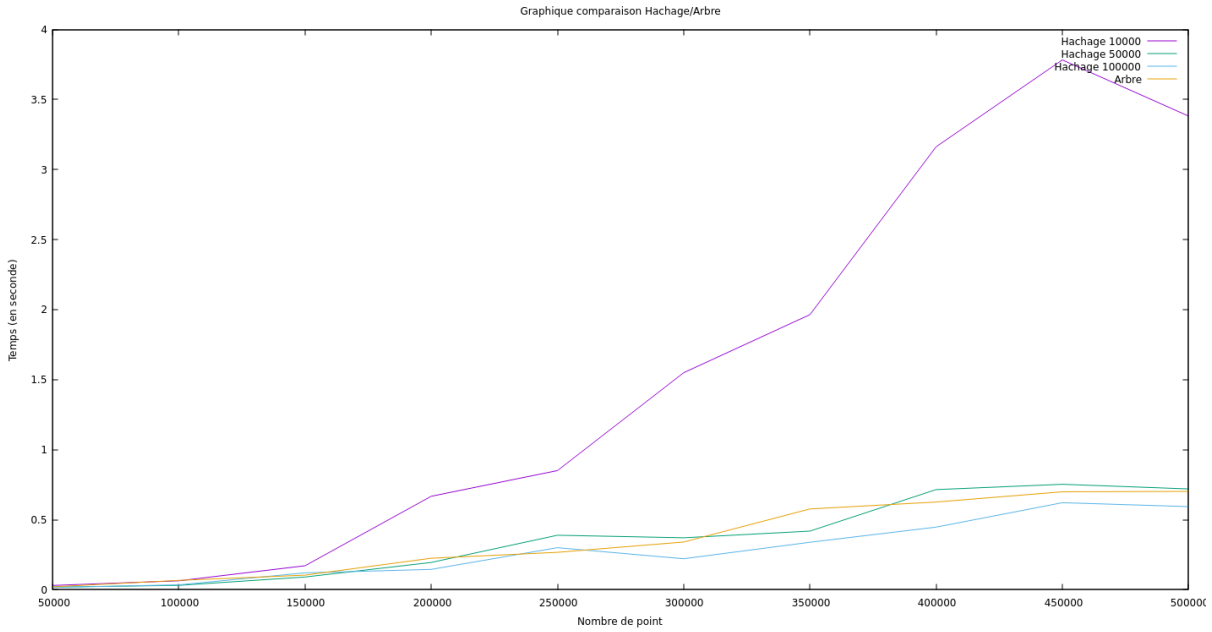
1. Créer un réseau vide en utilisant la fonction *creerReseau*.
2. Initialiser une table de hachage H avec une taille M.
3. Initialiser une liste de commodités à NULL.
4. Pour chaque chaîne :
    1. Initialiser *extrA* et *extrB* à NULL.
    2. Parcourir chaque point de la chaîne :
        1. Rechercher ou créer un nouveau nœud dans la table de hachage pour le point courant avec *rechercheCreeNoeudHachage*.
        2. Si un nœud précédent existe (*V*), insérer des voisins entre le nœud précédent et le nœud courant avec de la fonction *insererVoisins*.
        3. Mettre à jour *V* avec le nœud courant.
        4. Si le point actuel est le dernier de la chaîne, mettre à jour *extrB* avec le nœud correspondant.
    3. Si une commodité entre *extrA* et *extrB* n'existe pas déjà, l'ajouter à la liste de commodités.
5. Assigner la liste de commodités au réseau.
6. Retourner le réseau créé.

reconstitueReseauArbre(Chaines* C)

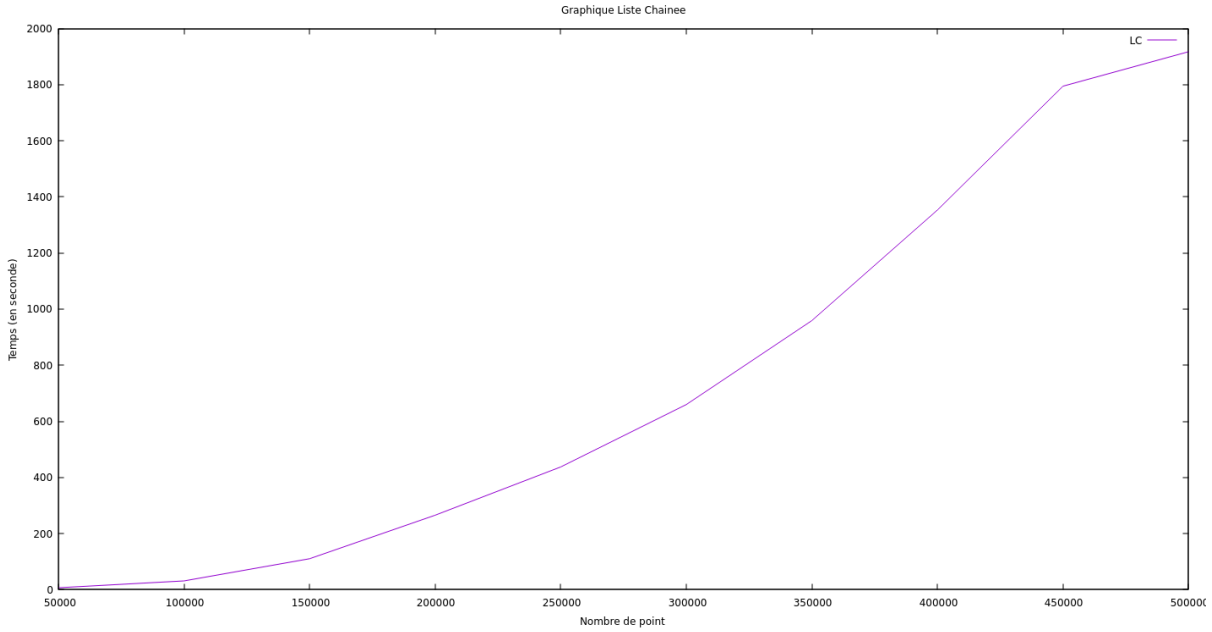
1. Créer un réseau vide en utilisant la fonction *creerReseau*.
2. Initialiser une liste de commodités à NULL.
3. Calculer les coordonnées minimales et maximales des chaînes avec *chaineCoordMinMax*.
4. Créer un arbre ArbreQuat avec les coordonnées du centre et les dimensions calculées.
5. Pour chaque chaîne :
    1. Initialiser *extrA* et *extrB* à NULL.
    2. Parcourir chaque point de la chaîne :
        1. Rechercher ou créer un nouveau nœud dans l'arbre pour le point courant avec *rechercheCreeNoeudArbre*.
        2. Si un nœud précédent existe (V), insérer des voisins entre le nœud précédent et le nœud courant avec de la fonction *insererVoisins*.
        3. Mettre à jour *V* avec le nœud courant.
        4. Si le point actuel est le dernier de la chaîne, mettre à jour *extrB* avec le nœud correspondant.
    3. Si une commodité entre *extrA* et *extrB* n'existe pas déjà, l'ajouter à la liste de commodités.
6. Assigner la liste de commodités au réseau.
7. Retourner le réseau créé.

Afin de procéder à l'analyse des résultats de chacune des structures, procédons d'abord à une observation des temps de calcul. Hachage 1, Hachage 2 et Hachage 3 possèdent respectivement un tableau de hachage de taille M = 10000, M = 50000 et M = 100000.

On observe dans le graphique ci-dessous que plus on augmente la taille de la table de hachage, plus elle est efficace. Elle surpasse totalement l'efficacité de la structure en arbre lorsque la taille du tableau égale 100000.



Le graphique ci-dessous, concernant la structure en liste chaînée, présente un temps de calcul considérablement grand, frôlant les 20 minutes pour un réseau de 500000 points !



Cette différence de temps ne s’explique pas uniquement par les fonctions de reconstitution de réseau de chaque structure car elles sont assez similaires. C’est en fait la fonction de recherche ou de création de nœud qui fait toute la différence.
On peut alors déjà deviner la complexité de cette fonction pour chacune des structures, grâce aux algorithmes.
La complexité de la fonction de reconstitution de chaque structure dépend alors de la complexité de la fonction de recherche d’un nœud.