Benhaddou Chady
Kinane-Daouadji Isaac

Rapport Projet

Sujet :

Ce projet vise à améliorer et à optimiser le réseau de fibres optiques d'une agglomération en deux étapes : la reconstruction complète du plan du réseau, actuellement fragmenté entre plusieurs opérateurs, et la réorganisation des attributions de fibres pour équilibrer leur utilisation. Des algorithmes seront testés sur des instances de réseaux pour identifier les meilleures méthodes d'optimisation.

Description globale des structures et des fonctions du code :

Structures et fonctions:

Chaine.h :

Le fichier ‘Chaine.h’ définit des structures et fonctions pour gérer un système de chaînes composées de points dans un réseau de communication:

Structures
1. **CellPoint** : Point avec coordonnées ‘x, y’ et lien vers le point suivant.
2. **CellChaine** : Identifie une chaîne par son numéro et contient une liste de points.
3. **Chaines** : Regroupe plusieurs chaînes, spécifiant le nombre de chaînes et le nombre maximal de fibres par câble.

Fonctions principales
- **Création et ajout** : Fonctions pour créer et ajouter des points et des chaînes.
- **Affichage et écriture** : Fonctions pour afficher les détails des points et chaînes et pour écrire dans des fichiers.
- **Calculs** : Fonctions pour calculer la longueur des chaînes et compter les points.
- **Lecture et génération** : Fonctions pour lire les chaînes à partir de fichiers et générer des chaînes aléatoires.

Utilitaires
Inclut des bibliothèques standard pour la manipulation de fichiers, les opérations mathématiques, et plus.

Reseau.h :

Le fichier ‘Reseau.h’ définit des structures et fonctions pour la gestion d'un réseau de communication composé de noeuds et de commodités :

Structures
1. **Noeud** : Représente un point du réseau avec un numéro, des coordonnées et une liste de voisins.
2. **CellNoeud** : Cellule pour une liste chaînée contenant un pointeur vers un noeud et vers la cellule suivante.

3. **CellCommodite** : Représente un lien entre deux noeuds (commodité), avec des pointeurs vers les noeuds aux extrémités et vers la cellule suivante.
4. **Reseau** : Contient l'ensemble du réseau, y compris une liste de noeuds, une liste de commodités, le nombre de noeuds, et le nombre maximal de fibres par câble.

Fonctions principales
- **Création et gestion** : Fonctions pour créer des noeuds, des cellules de noeuds, et des cellules de commodités, ainsi que pour insérer des noeuds et des voisins dans le réseau.
- **Affichage et écriture** : Fonctions pour afficher le réseau en SVG et écrire les détails du réseau dans des fichiers.
- **Utilitaires** : Fonctions pour rechercher ou créer des noeuds, reconstituer un réseau à partir d'une liste de chaînes, et calculer le nombre de liaisons et de commodités.

Inclusions
- Inclut des bibliothèques standard pour la manipulation de fichiers et de données, ainsi que le fichier ‘Chaine.h’ pour la compatibilité avec les structures de chaînes.

Hachage.h :

Le fichier ‘hachage.h’ définit les structures et fonctions pour la gestion d'un système de hachage adapté à un réseau de communication. Ce système est conçu pour optimiser la recherche et l'insertion de noeuds dans le réseau :

Structure
- **TableHachage** : Structure pour stocker une table de hachage contenant des pointeurs vers des listes chaînées de noeuds (‘CellNoeud’).
  - ‘nbElement’ : Nombre d'éléments actuellement stockés dans la table (pas nécessaire pour certaines opérations).
  - ‘tailleMax’ : Taille maximale de la table de hachage.
  - ‘T’ : Tableau de pointeurs sur les têtes de listes chaînées de ‘CellNoeud’.

Fonctions principales
- **h** : Fonction de hachage basique pour convertir une clé entière en index de tableau basé sur la taille ‘M’ de la table.
- **f** : Fonction pour calculer un indice de hachage à partir des coordonnées ‘x, y’ d'un noeud.
- **reconstitueReseauHachage** : Reconstruit un réseau à partir d'une liste de chaînes (‘Chaines’) en utilisant une table de hachage pour optimiser les insertions.
- **rechercheCreeNoeudHachage** : Recherche un noeud dans la table de hachage par ses coordonnées; si le noeud n'existe pas, il est créé et ajouté à la table.

Utilisation
- Ce fichier est utilisé en conjonction avec ‘Reseau.h’ pour permettre une gestion efficace des noeuds dans un réseau en utilisant des techniques de hachage, réduisant ainsi le temps nécessaire pour les opérations de recherche et d'insertion.

ArbreQuat.h :

Le fichier ‘ArbreQuat.h’ définit les structures et fonctions pour la gestion d'un arbre quaternaire adapté à la structuration spatiale des noeuds d'un réseau :

Structure
- **ArbreQuat** : Structure pour un arbre quaternaire, utilisé pour stocker et organiser spatialement les noeuds du réseau.
  - ‘xc, yc’ : Coordonnées du centre de la cellule de l'arbre.
  - ‘coteX, coteY’ : Dimensions de la cellule (longueur et hauteur).
  - ‘noeud’ : Pointeur vers le noeud du réseau stocké dans cette cellule.
  - ‘so, se, no, ne’ : Pointeurs vers les sous-arbres correspondant aux quadrants sud-ouest, sud-est, nord-ouest, et nord-est.

Fonctions principales
- **creerArbreQuat** : Crée un nouvel arbre quaternaire avec des dimensions et un centre spécifiés.
- **insererNoeudArbre** : Insère un noeud dans l'arbre quaternaire à la position appropriée en fonction de ses coordonnées.
- **rechercheCreeNoeudArbre** : Recherche un noeud par ses coordonnées dans l'arbre; si absent, un nouveau noeud est créé et inséré.
- **reconstitueReseauArbre** : Reconstruit un réseau à partir d'une liste de chaînes (‘Chaines’), en utilisant un arbre quaternaire pour optimiser le placement et la recherche des noeuds.
- **afficherArbre** : Affiche la structure de l'arbre quaternaire à des fins de débogage ou de visualisation, avec indication du niveau de profondeur.

Utilisation
- Le fichier est utilisé en conjonction avec ‘Reseau.h’ et ‘Chaine.h’ pour une gestion avancée des noeuds dans un réseau, permettant une recherche et une insertion efficaces grâce à la structuration spatiale qu'offre l'arbre quaternaire.

Graphe.h :

Le fichier ‘Graphe.h’ définit les structures et les fonctions pour la création et la manipulation de graphes basés sur un réseau, permettant notamment la gestion des sommets, arêtes, et chemins entre points :

Structures
- **Arete** : Représente une connexion entre deux sommets avec leurs numéros d'identité.
- **Cellule_arete** : Élément d'une liste chaînée contenant une arête et un pointeur vers la cellule suivante.
- **Sommet** : Décrit un sommet du graphe avec ses coordonnées et une liste de ses arêtes voisines.
- **Commod** : Identifie une commodité comme un lien entre deux sommets spécifiques.
- **Graphe** : Structure globale d'un graphe incluant un tableau de sommets, le nombre de sommets, de commodités, et des détails sur les capacités du réseau.
- **Liste** : Liste chaînée simple pour stocker des valeurs entières, utilisée pour représenter les chemins ou autres séquences de sommets.

Fonctions principales
- **creeArete**, **creeCellule_arete**, **creeSommet** : Fonctions pour créer des arêtes, des cellules d'arêtes, et des sommets.
- **insererAreteVoisin** : Insère une arête dans la liste des voisins d'un sommet si elle n'existe pas déjà.
- **affiche_graphe** : Affiche le graphe ou le sauvegarde dans un fichier spécifié.
- **creerGraphe** : Crée un graphe à partir d'un réseau existant.
- **cheminPlusCourt** : Calcule le chemin le plus court entre deux sommets dans le graphe.

- **ajouterEnTete**, **afficheListe** : Fonctions pour manipuler et afficher des listes chaînées d'entiers.
- **retrouverChemin** : Retrouve et retourne la liste représentant le chemin le plus court entre deux sommets.
- **reorganiseReseau** : Tente de réorganiser les connexions dans un réseau pour optimiser les routes entre les sommets.

Utilisation
Ce fichier est utilisé pour modéliser des réseaux sous forme de graphes, permettant des analyses complexes comme la recherche de chemins les plus courts et la réorganisation de réseau pour une meilleure efficacité. Il est essentiel dans les applications nécessitant une gestion détaillée et dynamique des connexions entre divers points dans des réseaux de transport, de télécommunication ou autres infrastructures similaires.

ReconstitueReseau.c :

Ce code C fournit une interface utilisateur pour reconstruire des réseaux à partir de données de chaînes géométriques stockées dans un fichier, en utilisant trois méthodes différentes : une liste, une table de hachage, et un arbre. L'utilisateur est invité à saisir le nom d'un fichier et à choisir la méthode de reconstruction. Selon le choix de l'utilisateur, le programme lit les données du fichier, les affiche, puis reconstruit et visualise le réseau en utilisant la méthode spécifiée, produisant une sortie au format SVG pour visualisation dans un navigateur. En cas de choix invalide, un message d'erreur est affiché.

SVGwriter.c :

Ce code en C est spécialement conçu pour créer des représentations visuelles de réseaux à l'aide de fichiers SVG, qui peuvent ensuite être visualisés dans un navigateur web. Les fonctions fournies permettent de manipuler des éléments graphiques comme des points et des lignes, ce qui est idéal pour représenter les nœuds et les connexions d'un réseau. Voici comment chaque fonction contribue à la représentation d'un réseau :

1. **SVGinit** : Prépare un fichier SVG pour y dessiner en spécifiant les dimensions et en initialisant les couleurs de base pour les points (nœuds) et les lignes (connexions).

2. **SVGlineColor** et **SVGpointColor** : Ces fonctions permettent de personnaliser les couleurs des lignes et des points, respectivement. Cela peut être utilisé pour distinguer différents types de connexions ou de nœuds dans un réseau.

3. **SVGlineRandColor** : Introduit une variabilité visuelle en attribuant des couleurs aléatoires aux lignes, ce qui peut être utile pour visualiser des caractéristiques ou des classifications au sein du réseau de manière distincte.

4. **SVGpoint** : Place un point (nœud) sur le SVG à des coordonnées spécifiques, permettant de visualiser la position des nœuds dans le réseau.

5. **SVGline** : Dessine une ligne entre deux points, représentant une connexion entre les nœuds dans le réseau.

6. **SVGfinalize** : Clôt le fichier SVG, garantissant que toutes les balises sont correctement fermées et que le fichier est prêt à être visualisé.

En combinant ces fonctions, le code est capable de générer des représentations graphiques détaillées d'un réseau, où les nœuds et les liens peuvent être clairement distingués grâce aux couleurs et à la disposition spatiale, facilitant ainsi l'analyse et la compréhension de la structure du réseau.

Struct_File.c :

Ce code en C définit des fonctions pour manipuler une file d'attente à l'aide d'une structure de données chaînée. Voici une explication détaillée des fonctions et leur utilité :

1. **Init_file(S_file *f)** : Cette fonction initialise une file d'attente vide. Elle configure les pointeurs ‘tete’ et ‘dernier’ de la structure ‘S_file’ sur ‘NULL’, indiquant que la file est vide.

2. **estFileVide(S_file *f)** : Cette fonction vérifie si la file est vide en retournant ‘1’ (vrai) si le pointeur ‘tete’ est ‘NULL’, et ‘0’ (faux) dans le cas contraire. Cette vérification est utile pour éviter des erreurs d'exécution lors de l'accès à des éléments d'une file vide.

3. **enfile(S_file *f, int donnee)** : Cette fonction ajoute un élément à la fin de la file. Elle crée une nouvelle `Cellule_file`, lui assigne une valeur, et ajuste les pointeurs pour maintenir l'ordre de la file. Si la file est initialement vide (‘tete’ est ‘NULL’), le nouvel élément devient à la fois la tête et le dernier élément de la file. Sinon, il est ajouté à la fin et le pointeur `dernier` est mis à jour.

4. **defile(S_file *f)** : Cette fonction retire l'élément en tête de file et retourne sa valeur. Elle stocke temporairement le pointeur vers l'élément en tête, met à jour le pointeur ‘tete’ pour pointer vers le prochain élément, et libère la mémoire de l'élément retiré. Si l'élément retiré est le dernier de la file, le pointeur `dernier` est également mis à jour à ‘NULL’.

Chacune de ces fonctions manipule les éléments de la file de manière à respecter la propriété fondamentale des files d'attente : le premier élément ajouté sera le premier à être retiré (FIFO, First In First Out). Ces opérations sont essentielles pour gérer de manière efficace les données dans des scénarios tels que la gestion des processus dans les systèmes d'exploitation, le traitement des tâches en attente, ou les algorithmes de planification.

Description des tests et de mesures de performance :

MesurePerformance.c :

Le code ‘MesurePerformance.c’ est utilisé pour mesurer les performances de différentes méthodes de reconstruction de réseaux à partir de chaînes de points générées aléatoirement. Voici un résumé de ce que chaque fonction fait et comment elle mesure les performances :

Fonction ‘genererDonneesPerformanceReconstitueReseauListe’
Cette fonction mesure le temps nécessaire pour reconstruire un réseau en utilisant des listes chaînées. Elle procède comme suit :
- Crée un fichier texte pour stocker les résultats.
- Génère des chaînes avec un nombre croissant de points (de 500 à 5000, par paliers de 500) dans une zone définie (‘xmax’ et ‘ymax’).
- Mesure le temps de calcul pour la reconstruction du réseau à partir de ces chaînes en utilisant la méthode ‘reconstitueReseauListe’.
- Écrit le nombre de chaînes et le temps de calcul dans le fichier.

Fonction ‘genererDonneesPerformance’
Cette fonction évalue les performances de plusieurs méthodes de reconstruction de réseaux, y compris l'utilisation d'un arbre quaternaire et de tables de hachage avec différentes tailles. Les étapes comprennent :
- Création d'un fichier texte pour les résultats.
- Génération de chaînes comme dans la première fonction.
- Mesure et comparaison des temps de reconstruction en utilisant :
  - ‘reconstitueReseauArbre’ : reconstruction utilisant un arbre quaternaire.
  - ‘reconstitueReseauHachage’ avec différentes tailles de table (10, 50, 100, 200) : mesure l'effet de la taille de la table de hachage sur les performances.
- Stocke les résultats pour chaque technique dans le fichier.

Points Clés
- **Méthodologie de Mesure** : Utilise ‘clock()’ pour obtenir le temps CPU utilisé par chaque méthode, ce qui est une manière commune de mesurer les performances dans des applications où le temps d'exécution est critique.
- **Comparaison des Performances** : Permet d'évaluer l'efficacité des différentes structures de données (listes chaînées, arbres quaternaires, tables de hachage) dans le contexte de la reconstruction de réseaux complexes.
- **Écriture des Résultats** : Les résultats sont écrits dans des fichiers texte pour une analyse ultérieure, permettant de visualiser les performances en fonction de l'augmentation de la complexité des données d'entrée.

En conclusion, ces fonctions fournissent un cadre pratique pour évaluer et comparer les performances de différentes stratégies de reconstruction de réseaux, facilitant l'optimisation des algorithmes pour des applications spécifiques.

Les fichiers de test suivants sont essentiels pour assurer la fiabilité des systèmes de gestion de réseaux, chaînes et graphes développés en C. Chaque fichier se concentre sur des aspects spécifiques pour tester rigoureusement les structures de données et les algorithmes sous-jacents. Voici un aperçu consolidé :

TestChaine.c
Ce fichier contient des tests unitaires pour les fonctions de gestion des listes de points et de chaînes, validant la création correcte des structures, l'ajout d'éléments, et les opérations sur les fichiers.

TestGraphe.c
Il se concentre sur l'intégration des fonctionnalités de graphes avec les chaînes et le hachage pour manipuler les données de réseau. Les tests vérifient la construction du graphe, l'affichage, et la recherche de chemins.

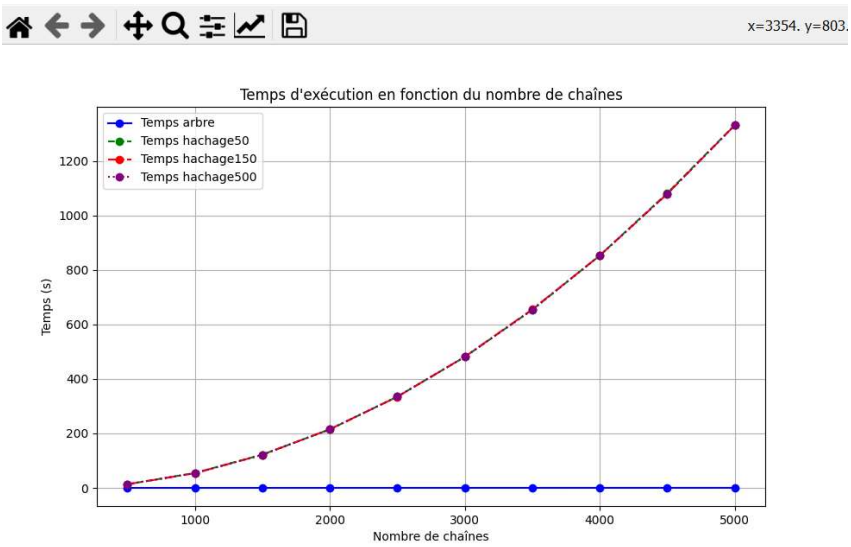
TestReseau.c
Ce fichier cible les fonctionnalités de base de la création et de la manipulation de réseaux, testant la création de réseaux et de nœuds, l'insertion de voisins, et la création de cellules pour les nœuds et les commodités.

TestArbreQuat.c
Il teste les fonctionnalités des arbres quaternaires pour la gestion spatiale des nœuds dans un réseau. Les tests incluent l'insertion de nœuds, la recherche et la vérification de la structure de l'arbre.

TestHachage.c
Ce code C teste des fonctions qui manipulent des réseaux et des tables de hachage, en vérifiant la création correcte de nœuds et la reconstruction de réseaux à partir de fichiers. Les résultats sont validés par des assertions pour assurer la fiabilité des opérations.

Thèmes et pratiques communs
Les tests utilisent des assertions pour vérifier le comportement correct des fonctions, la gestion correcte des cas limites, et la bonne gestion de la mémoire. Ces fichiers sont cruciaux pour maintenir une haute qualité de code et la fiabilité dans le développement d'applications basées sur des réseaux et des graphes. Ils garantissent que chaque composant fonctionne correctement avant le déploiement.

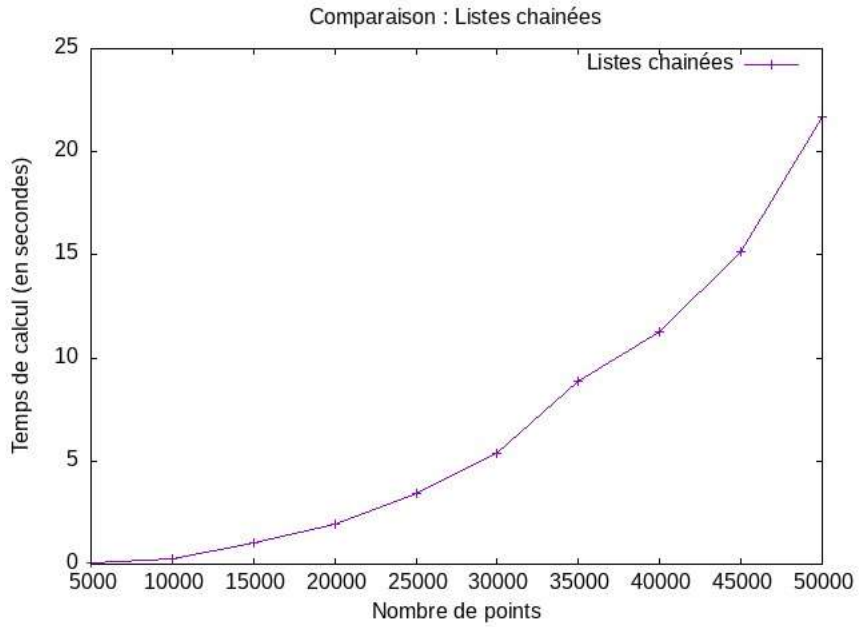
Analyse des performances :



1	500	0.016954	13.172514	13.109254	13.550263
2	1000	0.039576	54.685341	53.822720	53.747919
3	1500	0.056304	121.928899	121.482988	121.454679
4	2000	0.078093	214.771387	215.168718	214.782254
5	2500	0.102614	334.729472	334.362483	334.979965
6	3000	0.133056	480.882199	481.303055	481.046705
7	3500	0.163023	654.016109	655.517742	653.608433
8	4000	0.191243	853.124968	853.334783	852.897433
9	4500	0.210573	1082.438777	1079.546003	1079.552419
10	5000	0.242338	1332.478684	1331.450634	1331.824990

Analyse des Résultats
Les différences entre les temps pour différentes tailles de table de hachage indiquent l'impact de la taille de la table sur les performances. Une table plus grande pourrait réduire les collisions lors du hachage, améliorant ainsi les performances, mais cela nécessite également plus de mémoire.

Conclusions Pratiques
- **Efficacité des Méthodes** : Les résultats indiquent la viabilité et l'efficacité relative des différentes structures de données pour la gestion des réseaux. Un arbre quaternaire est préférable pour les scénarios où la structure spatiale est importante, tandis que les tables de hachage pourraient être optimales pour des accès rapides.
- **Optimisation des Performances** : L'analyse pourrait aider à optimiser les choix algorithmiques et de structure de données en fonction des exigences spécifiques de performance et d'échelle du problème.
- **Validation de l'Implémentation** : Tester avec des chaînes de tailles variées aide à identifier des problèmes potentiels de scalabilité et de performance avant le déploiement dans un environnement de production.



Le graphique précédent nous permet de conclure que la liste chaînée n'est pas la structure la plus appropriée en raison de sa complexité moyenne quadratique.

Réponses aux questions :

Exercice 4 :
Question 2 : Après avoir testé la fonction, on remarque qu'il n'y a pas de collisions, de ce fait, on peut conclure que celle-ci est tout à fait appropriée.

Exercice 7 :
Question 5 :
Analyse du comportement du code
1. **Création du graphe et initialisation de la matrice** :
   - Le graphe est créé à partir du réseau, et une matrice 'utilisationArete' est initialisée pour comptabiliser les utilisations de chaque arête entre les sommets.

2. **Calcul des chemins les plus courts et comptage de l'utilisation des arêtes** :

- Pour chaque commodité, le chemin le plus court entre les sommets spécifiés est calculé. Chaque fois qu'un segment de chemin entre deux sommets u et v est trouvé, les entrées correspondantes dans la matrice ‘utilisationArete’ sont incrémentées.
- Après chaque mise à jour, le code vérifie si cette arête est surchargée, c'est-à-dire si le comptage dépasse gamma.

3. **Détection de surcharge et gestion des erreurs** :
- Si une surcharge est détectée pour une arête, la fonction libère la mémoire allouée et retourne immédiatement ‘0’, indiquant qu'une ou plusieurs arêtes sont surchargées.

Raison du retour ‘0’
Le code retourne ‘0’ dans nos tests probablement parce que pour au moins une des commodités, le chemin le plus court calculé utilise une arête plus de fois que la capacité gamma ne le permet. Cela peut arriver si :
- **Les commodités sont nombreuses ou mal distribuées** : Si beaucoup de commodités nécessitent de passer par une même arête, la limite gamma peut être rapidement atteinte.
- **Manque d'alternatives dans les chemins** : Si le graphe manque de diversité dans ses connexions, les chemins les plus courts tendent à converger sur les mêmes arêtes.

Propositions d'amélioration
1. **Optimisation des chemins** : Plutôt que de choisir systématiquement le chemin le plus court, envisager des chemins alternatifs qui pourraient réduire la charge sur les arêtes les plus utilisées.
2. **Rééquilibrage du réseau** : Modifier la structure du réseau pour augmenter gamma sur les arêtes stratégiques ou ajouter de nouvelles arêtes là où les surcharges sont fréquentes.
3. **Analyse préliminaire** : Avant de choisir les chemins, effectuer une analyse pour prévoir les congestions potentielles et ajuster les routes en conséquence.

En ajustant la stratégie de sélection de chemins et en améliorant la structure du réseau, il serait possible de mieux répartir l'utilisation des arêtes et d'éviter les surcharges, permettant ainsi à la fonction ‘reorganiseReseau’ de retourner ‘1’ plus fréquemment, indiquant un réseau bien équilibré.