Rapport Projet Informatique Groupe NDAG

La répartition des tâches pour chaque membre du groupe a été faite comme suit:

Partie Communication Client- Serveur : **BERNARD NGAMBILI**

Mise en place de la communication réseau en TCP, à l'aide des sockets natifs de C++. 6 fichiers en tout ont été créés dont, client.hpp, client.cpp, serveur.hpp, serveur.cpp et deux mains distinctifs, l'un pour client et un autre pour le serveur. La première implémentation de Cbor également a mis en place
Quelques détails sur le contenu:

Fichier `client.hpp` :

- Contient la déclaration de la classe `ClientSocket`, qui représente un client socket.
- Les membres privés comprennent le descripteur de fichier pour le socket client et la structure `sockaddr_in` pour l'adresse du serveur.
- Les méthodes publiques incluent le constructeur, une méthode pour envoyer des données au serveur et une méthode pour fermer la connexion.

Constructeur `ClientSocket::ClientSocket(const char* adresseServeurIP, int port)` :

- Crée un socket client en appelant la fonction `socket()` avec les paramètres appropriés.
- Configure l'adresse du serveur en spécifiant la famille d'adresses, l'adresse IP et le port.
- Établit une connexion au serveur en utilisant la fonction `connect()`.

Méthode `ClientSocket::envoyerDonnees(cbor_item_t* message)` :

- Cette méthode est utilisée pour envoyer des données au serveur.
- Sérialise le message CBOR en utilisant la fonction `cbor_serialize_alloc()`.
- Envoie le message sérialisé au serveur en utilisant la fonction `send()`.
- Libère la mémoire allouée pour le tampon après l'envoi.

Méthode `ClientSocket::fermerConnexion()` :

- Cette méthode est utilisée pour fermer la connexion avec le serveur en fermant le socket client à l'aide de la fonction `close()`.

Fonction `UserInteraction()` :

- Cette fonction interagit avec l'utilisateur pour obtenir le type d'étude qu'il souhaite effectuer (Trajectoires ou Ensembles).

- Si l'utilisateur entre "T" ou "t", cela signifie qu'il souhaite étudier des trajectoires.
- Si l'utilisateur entre "E" ou "e", cela signifie qu'il souhaite étudier des ensembles.
- Retourne le type d'étude choisi par l'utilisateur ou "FALSE" si une entrée invalide est saisie.

Fonction `main()` :

- Crée une instance de `ClientSocket` en spécifiant l'adresse IP et le port du serveur auquel se connecter.
- Créé quelques trajectoires à des fins de démonstration.
- Interagit avec l'utilisateur pour obtenir le type d'étude qu'il souhaite effectuer.
- Construit un message CBOR contenant le type d'étude et les trajectoires correspondantes.
- Envoie le message CBOR au serveur à l'aide de la méthode `envoyerDonnees()`.
- Si l'entrée de l'utilisateur est incorrecte, affiche un message d'erreur.

CBOR (Concise Binary Object Representation) :

- Utilisé pour la sérialisation des données à envoyer au serveur.
- Permet de représenter les données de manière compacte et binaire.
- Dans ce code, CBOR est utilisé pour encapsuler le type d'étude (trajectoires ou ensembles) et les données de trajectoires avant de les envoyer au serveur.

Fichier `serveur.hpp` :

- Ce fichier contient la déclaration de la classe `ServeurSocket`, qui représente un serveur socket.
- Les membres privés de la classe comprennent les descripteurs de fichiers pour le socket serveur et le socket client, ainsi que les structures `sockaddr_in` pour les adresses du serveur et du client.
- Les méthodes publiques incluent le constructeur, les méthodes pour écouter les clients, traiter les connexions avec les clients et fermer la connexion.

Constructeur `ServeurSocket::ServeurSocket(int port)` :

- Initialise le socket serveur en appelant la fonction `socket()` avec les paramètres appropriés.
- Configure l'adresse du serveur en spécifiant la famille d'adresses, l'adresse IP et le port.
- Lie le socket à l'adresse du serveur en utilisant la fonction `bind()`.

Méthode `ServeurSocket::ecouterClients()` :

- Met le serveur en attente de connexions entrantes en utilisant la fonction `listen()`.
- Dans une boucle infinie, attend les connexions entrantes en utilisant la fonction `accept()` et traite chaque connexion avec la méthode `traiterConnexionAvecClient()`.

Méthode `ServeurSocket::traiterConnexionAvecClient()` :

- Cette méthode est appelée pour traiter chaque connexion avec un client.
- Elle reçoit les données envoyées par le client en utilisant la fonction `recv()`.
- Les données reçues sont interprétées comme un élément CBOR en utilisant la bibliothèque CBOR.
- Le contenu du message CBOR est affiché sur la sortie standard.
- La mémoire allouée pour le message CBOR est libérée.

Méthode `ServeurSocket::fermerConnexion()` :

- Ferme les sockets du serveur et du client en appelant la fonction `close()`

Tout cela a été fait avant l'évolution du code pour intégrer la partie de l'algorithme pour le calcul de similarité des trajectoires par un autre membre du groupe.

Partie Implémentation de la distance de Fréchet et tout ce qui y est relié : **Omar GOUMGOUM**

La classe 'Parser' : c'est une classe sensée d'extraire les données de la base de données, on a implémenté cette classe en sort que l'entête de fichier contient les mêmes attributs que le fichier fournit 'pigeons.csv' peu importe l'ordre, l'idée est d'exécuter le code de cette classe lors de déploiements de l'application une fois par toute.
La classe contient les méthodes suivantes :
ReadColumnPositions :
Cette fonction lit les positions des colonnes à partir de l'en-tête d'un fichier et stocke ces positions dans une map.
La complexité de cette fonction dépend principalement du nombre de colonnes dans l'en-tête du fichier.
Complexité : O(n), où n est le nombre de colonnes dans l'en-tête du fichier.
ExtractTrajectoriesFromFile :
Cette fonction extrait les trajectoires à partir d'un fichier CSV en utilisant les positions des colonnes fournies.
La complexité de cette fonction dépend du nombre de lignes dans le fichier et du nombre de colonnes à lire pour chaque ligne.
Complexité : O (m * n), où m est le nombre de lignes dans le fichier et n est le nombre de colonnes à lire pour chaque ligne.
La grande partie de code est dans la classe 'Trajectoire', cette classe propose les méthodes suivantes :
AjouterPoint :

Cette fonction ajoute un point à la trajectoire, ce qui implique simplement l'ajout d'un élément à un vecteur.
Complexité : O (1)
ObtenirPoints :
Cette fonction retourne une référence constante au vecteur de points de la trajectoire.
Complexité : O (1)
Afficher :
Cette fonction parcourt le vecteur de points et affiche chaque point.
Complexité : O(n), où n est le nombre de points dans la trajectoire.
ComputeDiscreteFrechetDistance :
Cette fonction calcule la distance de Fréchet discrète entre deux trajectoires en utilisant une approche récursive, c'est fonction qui dépend sur le calcule d'une matrice entière, en fait il y'a d'autre approche pour éviter le calcul des valeurs inutiles, ce qu'on permet de minimiser le temps d'exécution.
La complexité dépend de la taille des deux trajectoires.
Complexité : O (n * m), où n et m sont les tailles des deux trajectoires respectivement.
Les prochaines fonctions tous dépendent sur la fonction ComputeDiscreteFrechetDistance, l'idée était d'implémenter une fonction qui calcule la distance entre un trajectoire donnée et d'autre trajectoire, après on implémente cette fonction dans les différant parties de code
FindTrajectoriesWithinDistance :
Cette fonction cherche les trajectoires similaires dans un vecteur de trajectoire par rapport à une trajectoire donnée en utilisant une distance seuil.
La complexité dépend du nombre de trajectoires dans le vecteur et de la taille de chaque trajectoire.
Complexité : O (n * m), où n est le nombre de trajectoire et m est la taille de chaque trajectoire.
CountTrajectoriesWithinDistance :
Cette fonction compte le nombre de trajectoires similaires dans un vecteur de trajectoire par rapport à une trajectoire donnée en utilisant une distance seuil.
La complexité est similaire à findTrajectoriesWithinDistance, mais sans la surcharge due au stockage des trajectoires similaires.
Complexité : O (n * m), où n est le nombre de trajectoire et m est la taille de chaque trajectoire.
FindSmallestDiscreteFrechetDistance :
Cette fonction recherche la trajectoire la plus proche d'une trajectoire donnée dans un vecteur de trajectoire en utilisant la distance de Fréchet discrète.
La complexité est similaire à computeDiscreteFrechetDistance, avec une surcharge supplémentaire due à la recherche de la meilleure trajectoire.
Complexité : O (n * m), où n est le nombre de trajectoire et m est la taille de chaque trajectoire.

Partie Extension (Implémentation de la distance de Jaccard et tout ce qui y est relié) :
Charles DIAW

L'extension du projet vise à intégrer une fonctionnalité de recherche de profils similaires dans une base de données en utilisant l'indice de Jaccard comme mesure de similarité. Cette extension repose sur le modèle client-serveur déjà mis en place dans le projet initial.
Détails de l'implémentation :

La classe EnsembleProfils est le cœur de cette extension. Elle permet de stocker les profils utilisateurs et de comparer un profil de référence avec les autres profils de la base de données. Voici les détails de cette classe

Constructeur : Le constructeur par défaut initialise simplement l'instance de EnsembleProfils.

Méthode chargerDepuisFichier :

Cette méthode prend en paramètre le nom de la base de donnée et charge les profils à partir de cette base. Elle ouvre le fichier en utilisant ifstream, puis lit chaque ligne une par une. Pour chaque ligne lue, elle utilise un istringstream pour extraire les entiers qui représentent les mots-clés du profil. Ces entiers sont ajoutés à un vecteur profil, représentant tous les profils chargés depuis le fichier.

Méthode trouverProfilsSimilaires :

Cette méthode prend en paramètres un profil de référence et un seuil de similarité. Elle compare le profil de référence avec chaque profil dans la base de données en utilisant la distance de Jaccard. Pour cela, elle itère à travers tous les profils de la base de données et utilise la méthode privée calculerDistanceJaccard pour calculer la distance de Jaccard entre le profil de référence et chaque profil de la base de données. Si la distance calculée est inférieure ou égale au seuil spécifié, le profil est ajouté à un vecteur profilsSimilaires, qui est ensuite retourné à la fin de la méthode.

Méthode calculerDistanceJaccard :

Cette méthode prend deux profils en entrée et calcule leur distance de Jaccard. Pour cela, elle convertit chaque profil en un ensemble de mots-clés à l'aide de la classe std::set, puis calcule la taille de l'intersection et de l'union de ces ensembles. La distance de Jaccard est ensuite calculée en utilisant ces valeurs, et la méthode retourne cette distance.

L'extension ajoute une fonctionnalité de recherche de profils similaires basée sur l'indice de Jaccard au projet initial. Elle repose sur une implémentation du calcul de l'indice de Jaccard et de la recherche de profils similaires dans une base de données.

Partie Assemblage du Code (Evolution du code du serveur et du client) : **Abdelali ICHOU**

Des méthodes rajoutées , la librairie Cbor améliorée, assemblage de toutes les parties , ajout de quelques méthodes dans la classe serveur et client.

Rapport d'Analyse de Code

Classe `serveur`:

Méthode `traiterConnexionAvecClient()`

- Description :

La fonction `traiterConnexionAvecClient()` est responsable de la réception et du traitement des données envoyées par un client connecté au serveur. Voici une description détaillée de ce qui se passe dans cette fonction :

2. Chargement des Données en tant qu'Élément CBOR : Les données reçues sont chargées en tant qu'élément CBOR à l'aide de la fonction `cbor_load()`. Cette fonction convertit les données binaires en une structure de données CBOR.

3. Traitement du Message CBOR: Si le chargement des données CBOR est réussi (c'est-à-dire si `messageRecu` n'est pas NULL), le contenu du message CBOR est affiché à l'aide de `cbor_describe()` pour un examen visuel.

4. Détermination du Type de Données : La fonction `dataType()` est appelée pour déterminer si les données reçues concernent des trajectoires (`T`) ou des ensembles (`E`). Le résultat est stocké dans la variable `choix`.

5. Traitement des Données : En fonction du type de données (`choix`), les données reçues sont déchiffrées et traitées :
Si les données concernent des trajectoires (`T`), elles sont déchiffrées en une trajectoire à l'aide de `dechiffrerTrajectoire()` et sont ensuite soumises à un traitement spécifique à l'aide de `extractTrajectoires()`.
Si les données concernent des ensembles (`E`), elles sont déchiffrées en un vecteur d'entiers à l'aide de `dechiffrerEnsemble()` et sont ensuite soumises à un traitement spécifique à l'aide de `extractProfils()`.
Si le type de données est invalide, un message d'erreur est affiché.
Libération de la Mémoire : La mémoire allouée pour le message CBOR est libérée à l'aide de `cbor_decref()` après son utilisation.
Gestion des Erreurs : Si aucune donnée n'est reçue (`octetsLus == 0`), un message indiquant que le client s'est déconnecté est affiché. Si une erreur se produit lors de la réception des données, un message d'erreur est affiché.
Cette fonction assure une communication bidirectionnelle entre le client et le serveur, en permettant au serveur de traiter les données reçues du client de manière appropriée en fonction de leur type. Elle offre également une gestion robuste des erreurs pour garantir la fiabilité de la communication.
Méthode `dataType(const cbor_item_t* cborMessage, char& choix)`
- Description :
Extrait le type de données du message CBOR (trajectoire ou ensemble) et stocke le choix dans une variable.
1. Fonction dataType() : Cette fonction est chargée d'extraire le type de données du message CBOR reçu du client et de le stocker dans une variable choix. Voici ce qui se passe dans cette fonction :
2. Extraction du Type de Données : La paire de clé-valeur correspondant au type de données est extraite du message CBOR. La clé est généralement à l'indice 1 dans le map. Le pointeur vers l'élément CBOR représentant le type de données est récupéré à partir de cette paire.
3. Allocation de Mémoire pour la Chaîne de Caractères : De la mémoire est allouée dynamiquement pour stocker la chaîne de caractères représentant le type. Cette mémoire doit être allouée dynamiquement car la taille de la chaîne est inconnue à l'avance.
4. Copie des Données de la Chaîne de Caractères : Les données de la chaîne de caractères sont copiées depuis l'élément CBOR dans le tampon alloué.
5. Terminaison de la Chaîne de Caractères : Le dernier caractère de la chaîne de caractères est remplacé par le caractère nul pour s'assurer que la chaîne est correctement terminée.
6. Stockage du type dans la Variable choix : Le premier caractère de la chaîne de caractères est assigné à la variable choix, qui représente le type de données (Trajectoire ou Ensemble).
7. Libération de la Mémoire Allouée : Une fois le type de données stocké dans la variable choix, la mémoire allouée dynamiquement pour la chaîne de caractères est libérée pour éviter les fuites de mémoire. Cette approche garantit une gestion efficace de la mémoire en libérant la mémoire allouée dynamiquement dès qu'elle n'est plus nécessaire, ce qui aide à prévenir les fuites de mémoire et à optimiser l'utilisation des ressources système.
Méthode `dechiffrerTrajectoire(const cbor_item_t* cborMessage)`
- Description :
Cette fonction est chargée de décoder une trajectoire à partir d'une réponse CBOR reçue du client. Voici ce qui se passe dans cette fonction :

1. Création de la Trajectoire : Une instance de la classe `Trajectoire` est créée pour stocker les points extraits du message CBOR.
2. Extraction de la Trajectoire à partir du Message CBOR : La première paire de clé-valeur du message CBOR est extraite. Cette paire contient les données de la trajectoire.
Le pointeur vers l'élément CBOR représentant la liste des points de la trajectoire est récupéré à partir de cette paire.
3. Traitement des Points de la Trajectoire : La liste des points de la trajectoire est parcourue. Pour chaque point dans la liste :
- Il est vérifié si l'élément CBOR est bien un tableau de taille 2 et que chaque élément du tableau est un flottant.
- Si ces conditions sont satisfaites, les coordonnées x et y du point sont extraites à l'aide de la fonction `cbor_float_get_float8()` et un nouvel objet `Point` est créé avec ces coordonnées.
- Le point est ensuite ajouté à la trajectoire à l'aide de la méthode `ajouterPoint()` de la classe `Trajectoire`.
Cette fonction garantit une gestion efficace de la mémoire en évitant les fuites de mémoire. Aucune allocation dynamique de mémoire n'est effectuée ici, donc il n'y a pas besoin de libérer la mémoire explicitement. Les objets `Point` créés à l'intérieur de la fonction sont automatiquement détruits lorsque la trajectoire est retournée, ce qui évite les fuites de mémoire.
Méthode `dechiffrerEnsemble(const cbor_item_t* cborMessage)`
- Description : Déchiffre un ensemble à partir du message CBOR en construisant un vecteur d'entiers. Avant de l'envoyer, l'ensemble est transformé en un message CBOR.
- Le même principe comme la précédent
Méthode `extractProfils(const std::vector<int>& profilReference, double seuil)`
- Description : ici on appelle le travail de Ismael, l'Extrait les profils similaires à un profil de référence à partir d'un fichier de données Netflix en utilisant une distance seuil. Les profils sont convertis en messages CBOR avant d'être renvoyés.
Méthode `extractTrajectoires(const Trajectoire& path)`
- Description : ici on appelle le travail de Omar, Extrait les trajectoires similaires à une trajectoire donnée à partir d'un fichier CSV et affiche des informations sur ces trajectoires. Les trajectoires sont converties en messages CBOR avant d'être renvoyées.
Méthode `fermerConnexion()`
Conclusion :
Le serveur socket est conçu pour traiter les connexions des clients en utilisant le format CBOR pour échanger des données. Les données reçues, qu'elles soient des trajectoires ou des ensembles, sont transformées en messages CBOR avant d'être traitées. Cette approche permet une communication efficace entre le serveur et les clients, en minimisant la taille des données échangées tout en maintenant leur structure et leur sémantique.
Rapport d'Analyse de Code - Partie Client
La classe `main_client` implémente les fonctionnalités nécessaires à l'interaction avec l'utilisateur, à la conversion des données en messages CBOR et à leur envoi au serveur. Voici une analyse détaillée des différentes parties de cette classe :
Méthode `UserInteraction()`
- Description : Cette méthode demande à l'utilisateur de choisir le type d'étude qu'il souhaite effectuer : trajectoire (`T`) ou ensemble (`E`), sinon ('FALSE'). Elle renvoie le choix de l'utilisateur sous forme de chaîne de caractères.
Méthode `vector_to_cbor(const std::vector<int>& vec)`

- Description : Cette méthode convertit un vecteur d'entiers en un message CBOR, où chaque entier du vecteur est encodé en tant qu'élément du tableau CBOR.
Construction du Message CBOR
Construction du Message CBOR : En fonction du type d'étude choisi, les données sont préparées et ajoutées au message CBOR. Pour les ensembles, les profils sont convertis en un message CBOR à l'aide de la fonction `vector_to_cbor()`. Pour les trajectoires, les points sont convertis en un message CBOR en appelant la méthode `to_cbor()` de la classe `Trajectoire`.
Envoi du Message CBOR au Serveur : Le message CBOR est envoyé au serveur à l'aide de la méthode `envoyerDonnees()` de l'objet client.
Conversion des Trajectoires en Messages CBOR
- La méthode `to_cbor()` de la classe `Trajectoire` est utilisée pour convertir chaque trajectoire en un message CBOR. Cette méthode parcourt les points de la trajectoire et les encode en messages CBOR individuels avant de les regrouper dans un message CBOR de trajectoire.

Conclusion:
La classe `main_client` offre une interface conviviale pour permettre à l'utilisateur de choisir le type d'étude à effectuer et facilite la conversion des données en messages CBOR pour les transmettre au serveur. Cette approche garantit une communication efficace entre le client et le serveur en utilisant un format de données structuré, contribuant ainsi à la robustesse et à la flexibilité du système.