STOCKAGE DE DONNÉES PERSONNELLES

PROJ631 - PROJET ALGORITHMIQUE

RIBES MAËL

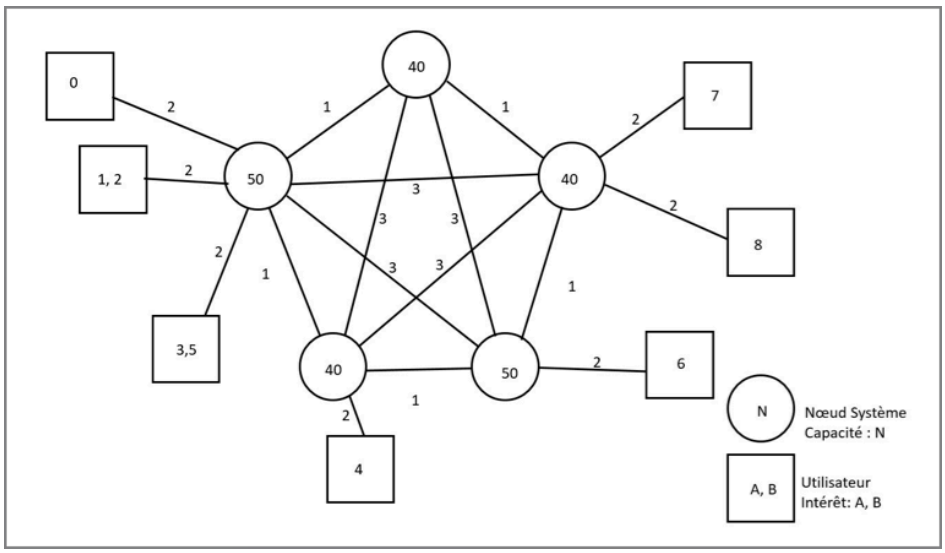
IDU3 - 2022

HTTPS://GITHUB.COM/MAELRIBES/PROJET_ALGO_STOCKAGE

INTRODUCTION

PRÉSENTATION DU PROJET

L'idée de ce projet est de modéliser un système de stockage de données personnelles. Ce système est composé de noeuds tous interconnectés les uns avec les autres. Ces noeuds ont un certain espace de stockage et peuvent stocker des données. Les utilisateurs peuvent communiquer avec un unique noeud et peuvent demander l'accès à une ou plusieurs données. Les données quant à elles ont une taille qui leur est propre et sont dans un premier temps accessibles par seulement un utilisateur.



Un exemple de système

CHOIX DU LANGAGE

La nature du projet présenté comme tel nous encourage à nous tourner vers une programmation orientée objet. J'ai choisi d'utiliser Java pour la réalisation du projet, car malgré le typage statique et la nature stricte du langage par rapport à un langage plus « simple » comme Python, je trouve Java plus clair et mieux ordonné pour gérer les classes.

ORGANISATION

Ayant des difficultés à comprendre et utiliser git, je n'ai fait que push sur mon master les fonctionnalités au fur et à mesure de l'avancée du projet sans créer de branches. Concernant mon organisation dans les tâches à réaliser, j'ai passé ma première séance à comprendre le sujet, la deuxième à créer la structure de donnée, la troisième à créer la classe **Matrice** et enfin la

dernière à commencer d'implémenter le placement des données. J'ai dû finir le projet en dehors des heures de cours.

STRUCTURE DE DONNÉES

À première vue, le projet nécessite la création de trois classes : **Données**, **Utilisateurs** et **Noeuds système**. Je vais à présent détailler les attributs et les caractéristiques de ses classes.

CLASSE DONNÉES

```
7       private final int idD;
8       private final int taille;
9       private static int idDonnes;
10      private ArrayList<Utilisateurs> utilisateursInteret = new ArrayList<>();
11      private static ArrayList<Donnees> listDonnees = new ArrayList<>();
```

Attributs de la classe Données

Comme indiqué en introduction, chaque donnée possède un idD unique et une taille. L'attribut static idDonnees s'incrémente à chaque nouvelle donnée créée (cf. Constructeur de Données) et permet de donner à celle-ci son ID. idDonnees permet par conséquent de connaître le nombre total de données créées. Les trois classes du projet possèdent un système similaire d'attribution d'ID

Les deux derniers attributs ***utilisateursInteret*** et ***listDonnees*** ont été créés au cours du projet pour faciliter le développement de certaines fonctionnalités. ***utilisateursInteret*** est la liste des utilisateurs qui accèdent à cette donnée. Cet attribut est utilisé dans le cas où le placement d'une donnée devrait être optimisé en fonction de ses multiples utilisateurs. ***listDonnees*** est tout simplement une liste qui contient toutes les données créées.

```
14      // Constructor
15      public Donnees(int taille) {
16          this.taille = taille;
17          this.idD = idDonnes;
18          idDonnes++;
19          listDonnees.add(this);
20      }
```

Constructeur de la classe Données

Le constructeur de la classe reste assez basique. Comme expliqué plus haut, ***idDonnees*** attribue à la donnée créée son id puis s'incrémente et la donnée est ajoutée à ***listDonnees***.

Données possède des getters pour ses différents attributs, mais ne possède pas de setters car je ne voyais pas d'utilité à la modification des attributs.

CLASSE UTILISATEURS

```
7         private final int idU;
8         private static int idUtilisateurs;
9         private ArrayList<Donnees> donnesInteret = new ArrayList<>();
10        private NoeudsSysteme noeudAccessible;
```

Attributs de la classe Utilisateurs

Utilisateurs possède un noeud ***noeudAccessible*** qui est l'unique noeud auquel l'utilisateur est lié et une liste ***donnesInteret*** qui contient les données auxquelles l'utilisateur peut accéder.

```
12        // Constructor
13        public Utilisateurs(@NotNull NoeudsSysteme noeudAccessible) {
14            this.noeudAccessible = noeudAccessible;
15            this.idU = idUtilisateurs;
16            idUtilisateurs++;
17            noeudAccessible.ajoutUtilisateurAccessible( user: this);
18        }
```

Constructeur de la classe Utilisateur

La classe **Utilisateurs** utilise le même système de système de numérotation d'id. Elle comporte des getters mais pas de setters pour les mêmes raisons que dans la classe **Données**. Le constructeur met aussi à jour la liste des utilisateurs accessibles du noeud.

```
54        public void ajoutDonneesInteret(Donnees donnees){
55            this.donnesInteret.add(donnees);
56            donnees.getUtilisateursInteret().add(this);
57        }
```

Méthode permettant l'ajout d'une donnée à la liste des intérêts

En plus d'ajouter une donnée à la liste des données d'intérêt d'un utilisateur, cette méthode met aussi à jour la liste contenant les utilisateurs qui ont un intérêt pour la donnée.

CLASSE NOEUDS SYSTÈME

Cette classe est plus complexe que les deux précédentes. En effet, elle comporte de nombreux attributs et des méthodes permettant leur manipulation.

Noeuds système possède une taille ***capaMemoire*** qui permet de stocker des données. Elle possède également une liste de données stockées ***donnesStockees***, une liste de noeuds accessibles ***noeudsAccessibles***, une liste d'utilisateurs accessibles ***utilisateursAccessibles*** et une

```java
private final int idN;
private static int idNoeuds = 0;
private int capaMemoire;
private ArrayList<Donnees> donnesStockees = new ArrayList<>();
private ArrayList<NoeudsSysteme> noeudsAccessibles = new ArrayList<>();
private ArrayList<Utilisateurs> utilisateursAccessibles = new ArrayList<>();
static Matrice matrice;
private static ArrayList<NoeudsSysteme> listNoeuds = new ArrayList<>();
```

Attributs de la classe Noeuds Système

liste globale *listNoeuds* de tous les noeuds créés. Je reviendrais dans la partie suivante sur la classe **Matrice** que j'ai dû créer pour gérer les liens entre les différents noeuds.

La classe possède tous les getters permettant l'accès à ses attributs et un setter permettant de mettre à jour la capacité mémoire du noeud.

La manipulation des listes de **Noeuds système** se fait à l'aide des méthodes suivantes :

```java
public void ajoutDonneesStockage(Donnees donnees){
    if (this.getCapaMemoire() >= donnees.getTaille()){
        this.getDonnesStockees().add(donnees);
        this.setCapaMemoire(this.getCapaMemoire() - donnees.getTaille());
    }
}
public void ajoutNoeudAccessible(NoeudsSysteme noeud, int poid){
    this.noeudsAccessibles.add(noeud);
    noeud.noeudsAccessibles.add(this);
    matrice.ajoutArc( n1: this,noeud,poid);
}
public void ajoutUtilisateurAccessible(Utilisateurs user){
    this.utilisateursAccessibles.add(user);
}
```

Méthodes de manipulations des listes de Noeuds Système

ajoutDonneesStockage() vérifie si la mémoire du noeud est suffisante pour ajouter la donnée puis ajoute la donnée si c'est le cas et met à jour la capacité mémoire du noeud.

ajoutNoeudAccessible() ajoute un lien entre deux noeuds en plaçant l'un l'autre dans leur liste de noeuds accessibles respectif. La méthode ajoute le l'arc créé avec son poids dans la matrice.
ajoutUtilisateurAccessible() ajoute simplement un utilisateur à la liste des utilisateurs accessibles.

CLASSE MATRICE

Ce système de stockage composé de noeuds interconnectés se rapporte finalement à un graphe non orienté. La méthode la plus simple pour représenter ce graphe est une matrice d'adjacence carrée symétrique.

```
private static ArrayList<ArrayList<Integer>> matriceAdjacence = new ArrayList<>();
```

Celle-ci se présente sous la forme d'une liste à deux dimension d'entier représentant le poids de l'arc entre les deux noeuds. Les noeuds étant tous identifiés grâce à un identifiant numéroté de 0 à n (n = nb total de noeud -1), les indices de colonnes et de lignes correspondent directement aux identifiant des noeuds ce qui facilite grandement l'implémentation de la suite du projet.

```
public void updateMatrice(){
    int n = NoeudsSysteme.getIdNoeuds()+1;

    if (matriceAdjacence.isEmpty()){
        matriceAdjacence.add(new ArrayList<>());
        matriceAdjacence.get(0).add(0);
    }

    for (int i = 0 ; i < n-1 ; i++) {
        matriceAdjacence.get(i).add(0);
    }

    if (n > 1){
        matriceAdjacence.add(new ArrayList<>());
        for(int i = 0 ; i < n ; i++) {
            matriceAdjacence.get(n-1).add(0);
        }
    }
}
```

Cette méthode updateMatrice() est appelée dans le constructeur de la classe **Noeuds système** et permet d'augmenter la taille de la matrice à chaque noeud créé et de l'initialiser avec des 0 en attendant que les arcs avec les autres noeuds ne soient déclarés. Une fois la matrice correctement initialisée, la méthode ajoutArc() peut s'occuper de modifier les 0 situées aux emplacements des deux noeuds à lier dans la matrice par le poids de l'arc.

```
public void ajoutArc(NoeudsSysteme n1, NoeudsSysteme n2, int poid){
    int id1 = n1.getIdN();
    int id2 = n2.getIdN();
    matriceAdjacence.get(id1).set(id2,poid);
    matriceAdjacence.get(id2).set(id1,poid);
}
```

PLACEMENT EFFICACE DES DONNÉES

IMPLÉMENTATION DE DIJKSTRA

Afin de trouver le meilleur emplacement pour placer une donnée au plus proche de l'utilisateur, il faut pouvoir parcourir tous les chemins possibles du ***noeudAccessible*** vers tous les autres noeuds et garder pour chacun d'entre eux la distance minimale. J'ai donc adapté l'algorithme de Dijkstra au projet pour pouvoir récupérer, pour un noeud donné, un tableau contenant la distance minimale à tous les autres noeuds, l'index du tableau correspondant à l'id du noeud. Je me suis inspiré du cours de prépa du lycée Thiers sur l'Algorithme de Dijkstra : [lien ici](#).

```java
/* Cette méthode retourne un tableau contenant les distances minimales à chaque nœud du système.
   L'indice des nœuds dans le tableau correspondant à leur id. */
3 usages  MaelRibes *
public double[] dijkstra(){
    // Initialisation des variables :
    int id = this.getIdN();
    double inf = Double.POSITIVE_INFINITY;
    ArrayList<ArrayList<Integer>> matAdj = Matrice.getMatriceAdjacence();
    int n = matAdj.size();

    // Création du tableau de distance et initialisation de celui-ci à avec l'infini
    double[] dist = new double[n];
    for(int i=0 ; i<n ; i++){
        if(i == id){
            dist[i] = 0;
        }
        else{
            dist[i] = inf;
        }
    }
    // Création de la liste des nœuds non marqués et remplissage avec les id de tous les nœuds
    ArrayList<Integer> nonMarques = new ArrayList<>();
    for(int i=0 ; i<n ; i++) {
        nonMarques.add(i);
    }
    // Création de la liste de nœuds marqués. Vide au début
    ArrayList<Integer> marques = new ArrayList<>();

    // Dijkstra :
    while(marques.size() < n){ // Tant que tous les nœuds ne sont pas marqués
        int i = plusCourt(dist,nonMarques); // i prend la valeur de l'id du nœud le plus proche non marqué
        marques.add(i); // On marque i
        nonMarques.remove(Integer.valueOf(i)); // et on le retire des non marqués.
        for(int k : nonMarques) { // Pour chaque nœud non marqué,
            // si la distance de i + le poids de l'arc i-k est inférieur à la distance de k,
            if ((dist[i] + matAdj.get(i).get(k)) < dist[k]){
                dist[k] = dist[i] + matAdj.get(i).get(k); // k = distance de i + le poids de l'arc i-k
            }
        }
    }
    return dist;

}
```

Algorithme de Dijkstra commenté

PLACEMENT AU PLUS PROCHE DE L'UTILISATEUR

À présent, nous avons une méthode qui permet d'obtenir les distance d'un noeud à tous les autres, il devient simple de savoir où placer une donnée. Il suffit de placer la donnée dans le noeud le plus proche possédant suffisamment de place. C'est la méthode meilleurEmplacement() qui se charge de trouver ce noeud en fonction de la donnée placée en paramètre et du tableau de distances.

```java
public static NoeudsSysteme meilleurEmplacement(@NotNull Donnees donnees, double[] distances) {
    int cpt = 0;
    NoeudsSysteme noeudCourant;
    NoeudsSysteme noeudChoisi = null;
    while (distances.length > cpt) {
        int indice = minimumTableau(distances);
        noeudCourant = NoeudsSysteme.getListNoeuds().get(indice);
        if (noeudCourant.getCapaMemoire() >= donnees.getTaille()) {
            noeudChoisi = noeudCourant;
            return noeudChoisi;
        } else {
            distances[indice] = Double.POSITIVE_INFINITY;
        }
        cpt++;
    }
    System.out.println("Pas de place pour la donnée n°" + donnees.getIdD());
    return noeudChoisi;
}
```

Tant que l'on n'a pas testé si la donnée ne rentre pas dans tous les noeuds (compteur cpt), la méthode récupère l'indice de la plus petite distance (cf. Méthodes annexes). Cet indice permet de récupérer le noeud correspondant. Si la donnée rentre dans ce noeud, on retourne le noeud. Sinon, on modifie à l'infini la distance correspondant au noeud dans le tableau. On incrémenté le compteur et on recommence avec le prochain noeud le plus proche.

On peut à présent écrire une méthode placerToutesDonnees() qui place toutes les données étant demandées par un ou plusieurs utilisateurs dans le noeud plus proche possible de ceux-ci en prenant en compte le cas où plusieurs utilisateurs demandent la même donnée.
Dans le cas simple où un seul utilisateur accède à la donnée, on récupère son ***noeudAccessible*** et on applique lui dijkstra() pour récupérer le tableau de distance associé. Si plusieurs utilisateurs accèdent à la donnée, on récupère le tableau de distance de chaque ***noeudAccessible*** et on les ajoute (cf. Méthodes annexes). Cela nous donne un tableau de « score de distance » pour chaque noeuds, le noeud ayant le score le plus bas étant le noeud le plus proche en moyenne de tous les utilisateurs.
Il nous reste plus à utiliser meilleurEmplacement() en prenant en paramètre la donnée et le tableau de distance trouvé précédemment pour trouver le noeud où la donnée doit être placée.

```java
public static void placerToutesDonnees() {
    NoeudsSysteme meilleurEmplacement;
    Utilisateurs utilisateurs;
    double[] tabDist = new double[NoeudsSysteme.getIdNoeuds()];
    for (Donnees donnees : listDonnees) {
        // si la donnée n'est demandée par aucun utilisateur, on passe.
        if (donnees.getUtilisateursInteret().size() == 0){
            System.out.println("La donnée n°" + donnees.getIdD() +
            " n'est demandée par aucun utilisateur : impossible de la placer");
        }
        // si demandée par 1 util., on récupère le tableau de distance de son noeudAccessible
        else if (donnees.getUtilisateursInteret().size() == 1) {
            utilisateurs = donnees.getUtilisateursInteret().get(0);
            tabDist = utilisateurs.getNoeudAccessible().dijkstra();
        }
        // si demandée par plusieurs utili,
        // on récupère le tableau de distance du noeudAccessible de chaque utilisateur
        // et on les somme pour obtenir un unique tableau.
        else if (donnees.getUtilisateursInteret().size() > 1) {
            for (Utilisateurs utilisateur : donnees.getUtilisateursInteret()) {
                tabDist = sumTableau(tabDist,utilisateur.getNoeudAccessible().dijkstra());
            }
        }
        // On recherche le meilleur emplacement grace au tableau calculé précédemment
        meilleurEmplacement = meilleurEmplacement(donnees,tabDist);
        if (meilleurEmplacement != null) {
            meilleurEmplacement.ajoutDonneesStockage(donnees);
        }
    }
}
```

Méthode placerToutesDonnees() commenté

MKP PROBLEM

Le placement des données au plus proche de l'utilisateur grâce aux méthodes précédentes peut cependant générer des problèmes. En effet, placer les données dans l'ordre de leur id peut entraîner un placement inefficace des données et celles-ci peuvent ne pas toutes rentrer dans le stockage alors que celui-ci était théoriquement suffisant. On cherche donc une manière plus efficace de placer les données.

La solution que l'on pourrait imaginer serait de trier les données par ordre décroissant de taille et de tenter de les placer dans le noeud le plus petit possible. Grâce à cette méthode, l'espace « perdu » dans les noeuds est minimal.

```java
public static NoeudsSysteme plusPetitEcart(Donnees donnees){
    NoeudsSysteme node = null;
    int ecart = Integer.MAX_VALUE;
    int gap;
    for( NoeudsSysteme noeudsSysteme : NoeudsSysteme.getListNoeuds()){
        gap = noeudsSysteme.getCapaMemoire() - donnees.getTaille();
        if(gap < ecart && gap >= 0){
            node = noeudsSysteme;
            ecart = gap;
        }
    }
    return node;
}
```

Cette méthode plusPetitEcart() nous permet de trouver le noeud qui aura le plus petit stockage restant après le placement de la donnée. La liste des données doit être triée au préalable.

La méthode mkpProblem() trie la liste des données (cf. Méthodes annexes) puis place chaque donnée dans le noeud qu'elle rempli le plus.

```java
public static void mkpProblem(){
    triDecroissant();
    NoeudsSysteme nodeResult;
    for( Donnees donnees  : listDonnees){
        nodeResult = plusPetitEcart(donnees);
        if(nodeResult != null){
            nodeResult.ajoutDonneesStockage(donnees);
        }
        else{
            System.out.println("Pas de place pour la donnée n°" + donnees.getIdD());
        }
    }
}
```

MÉTHODES ANNEXES

```java
public static double[] sumTableau(double[] t1, double[] t2){
    double[] newTab = new double[t1.length];
    for(int i=0 ; i<t1.length ; i++){
        newTab[i] = t1[i] + t2[i];
        }
    return newTab;
}
```

Addition deux tableaux

```java
public static int minimumTableau(double[] tab){
    double min = Double.MAX_VALUE;
    int indice = 0;
    for(int i=0 ; i<tab.length ; i++){
        if(tab[i]<min){
            indice = i;
            min = tab[i];
        }
    }
    return indice;
}
```

Trouver l'indice minimal d'un tableau

```java
public static void triDecroissant(){
    Donnees data1, data2 = null;
    ArrayList<Donnees> listD = Donnees.listDonnees;
    for(int j = 0 ; j < Donnees.getIdDonnes();j++){
        for(int i = 0 ; i < Donnees.getIdDonnes()-1;i++){
            if(listD.get(i).getTaille() < listD.get(i + 1).getTaille()){
                data1 = listD.get(i);
                data2 = listD.get(i+1);
                listD.set(i,data2);
                listD.set(i+1,data1);
            }
```

Tri à bulle décroissant de la liste des données

Ces quatre méthodes sont utilisées dans les méthodes principales du code. Je ne détaillerais pas leur fonctionnement, d'autant plus que ces méthodes restent relativement classiques.

```java
/*** Cette méthode recherche dans un tableau T l'indice de la valeur qui
a la valeur minimale parmi une liste I d'indice. */
1 usage   MaelRibes *
static int plusCourt(double[] T, ArrayList<Integer> I){
    double minimum = Double.POSITIVE_INFINITY; // On initialise le minimum à l'infini
    int indice = 0;
    for(int i : I){ // Pour chaque indice dans I,
        if((T[i] < minimum) && T[i] >= 0){ // Si la valeur d'indice i dans T
            indice = i;                    // est inférieure au minimum et positive
            minimum = T[i];
        }
    }
    return indice;
}
```

Recherche du plus court chemin

CONCLUSION

Je suis plutôt satisfait du rendu final du projet. J'ai passé beaucoup de temps en dehors des heures de cours à essayer de le terminer. J'ai surtout perdu des plusieurs heures au début du projet car je ne savais pas comment implémenter correctement les structures de données. Puis j'ai eu l'idée de créer une matrice d'adjacence et c'est ce qui m'a débloqué. L'implémentation de l'algorithme de Dijkstra m'a également pris énormément de temps. C'est le coeur du projet, tout le placement des données repose sur notre capacité à parcourir le graphe et à trouver le plus petit chemin entre les différents noeuds. Il était donc primordial de trouver une manière efficace de le coder.

Le seul regret que j'ai à l'égard de ce projet, c'est ma mauvaise utilisation de git. Je n'ai jamais utilisé l'outil auparavant et je ne comprenais pas bien le concept de branch. Je ferais les choses plus correctement pour le deuxième projet.