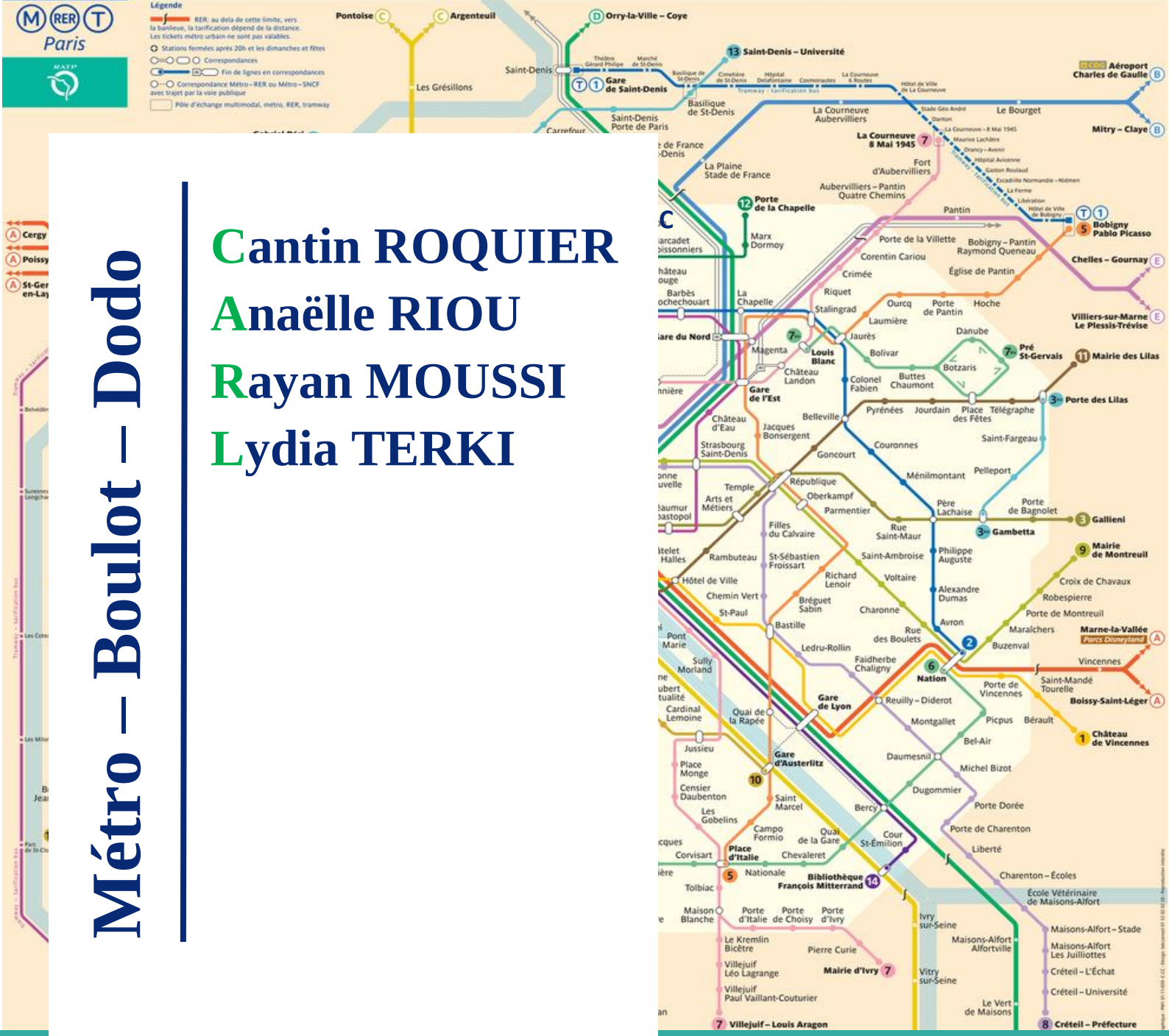
Métro – Boulot – Dodo

Cantin ROQUIER
Anaëlle RIOU
Rayan MOUSSI
Lydia TERKI



LSI – 2
EFREI Paris

CARL

Sommaire :

I. Répartition du travail

En tant qu'équipe nous avons organisé le travail de manière à ce que tout le monde puisse s'investir tout en créant une bonne cohésion dans ce projet. Pour cela, nous avons décidé d'unir nos forces et répartir les tâches selon les points faibles et les points forts propres à chaque membre.

La partie algorithmique étant la majeure partie du projet, il a été décidé que 3 membres collaborent dessus : Cantin, Anaëlle et Lydia. Rayan étant passionné par les interfaces graphiques, il a souhaité travailler sur la partie graphique du projet.

L'écriture du rapport a été collaboratif entre tous les membres de l'équipe.

II. Structure de données

a. Choix du langage

La première étape a été de nous mettre d'accord sur le langage à utiliser pour ce projet. Nous avons hésité entre Python et Java. Finalement, notre choix s'est porté sur le langage Java étant donné que la plupart des membres étaient plus confortables avec ce dernier.

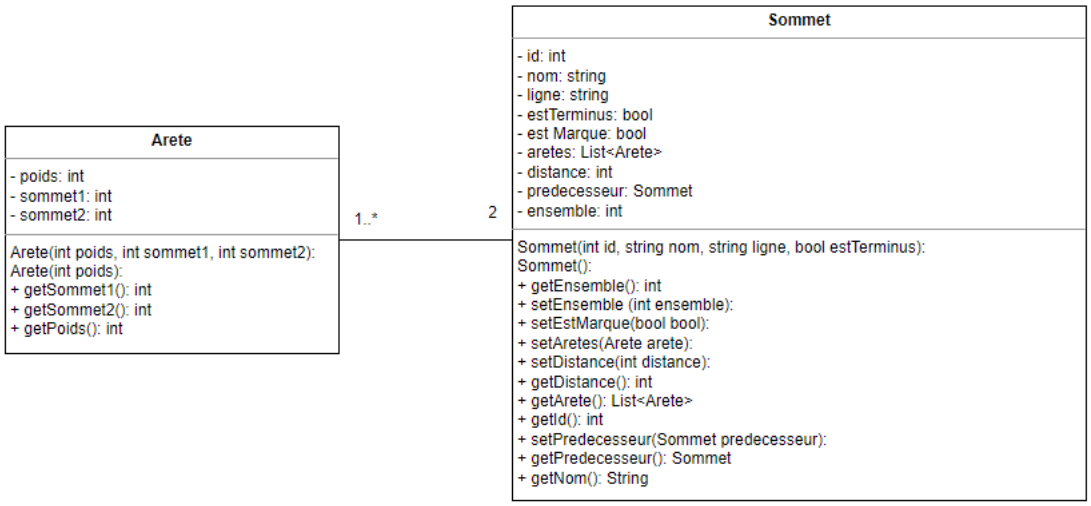
b. Modèle de conception

Compte tenu du fait que nous avons intégré une interface graphique au projet, nous avons opté pour une architecture de projet « MVC » (Model View Controller). Ce modèle de conception composé de trois couches, permet de séparer la logique de l'application et l'interface utilisateur.

L'architecture du modèle « MVC » se décompose en 3 couches :

- **Modèle** : utilisé pour transporter les données et contenir la logique pour mettre à jour le contrôler si nécessaire.
- **Vue** : utilisé pour visualiser les données contenues dans le modèle.
- **Contrôler** : utilisé pour gérer le flux de l'application, c'est-à-dire le flux de données dans l'objet modèle et pour mettre à jour la vue chaque fois que des données sont modifiées.

c. Diagramme de classe



Dès le départ du projet, nous avons défini les classes Sommet et Arête qui nous aideraient pour l'entièreté du projet.

- **« Sommet.class » :**

La classe « Sommet » est instancié comme étant composé d'un id, un nom, une ligne, un booléen afin de savoir si le sommet est un terminus ou non, un booléen pour savoir si le sommet est marqué, une liste d'arête, une distance, un prédécesseur et un ensemble.

Le constructeur de sommet prend en argument un entier id, un string pour le nom et un pour la ligne, et un booléen pour savoir si le sommet est un terminus.

- **« Arete.class » :**

La classe « Arete » est instancié comme étant composé d'un poids et de deux sommets (ce qui correspond à un trajet entre 2 arrêts de métro).

Son constructeur prend toutes les caractéristiques d'une arête en argument.

On a ensuite ajouté des getters et des setters dans les 2 classes et diverses fonctions afin de permettre l'affichage (public String *toString*(), public void *printAretes*()).

III. Méthodes algorithmiques

« Metro.java » est le fichier principal (main).

Ce fichier lance tous les algorithmes précédemment élaborés. Il permet grâce à différentes méthodes de parser le fichier « metro.txt » et d'affecter les données du graphe à *listeSommets* (Hashmap) ou *listeAretes* (Liste). La HashMap *listeSommets* associe chaque sommet avec un entier (ce qui nous permet d'associer un id avec un sommet).

On instancie un objet AlgoChemins (plus court chemin) qui a différentes méthodes permettant de savoir si le graphe est connexe, de trouver le plus court chemin entre deux stations et de déterminer l'arbre couvrant de poids minimum.

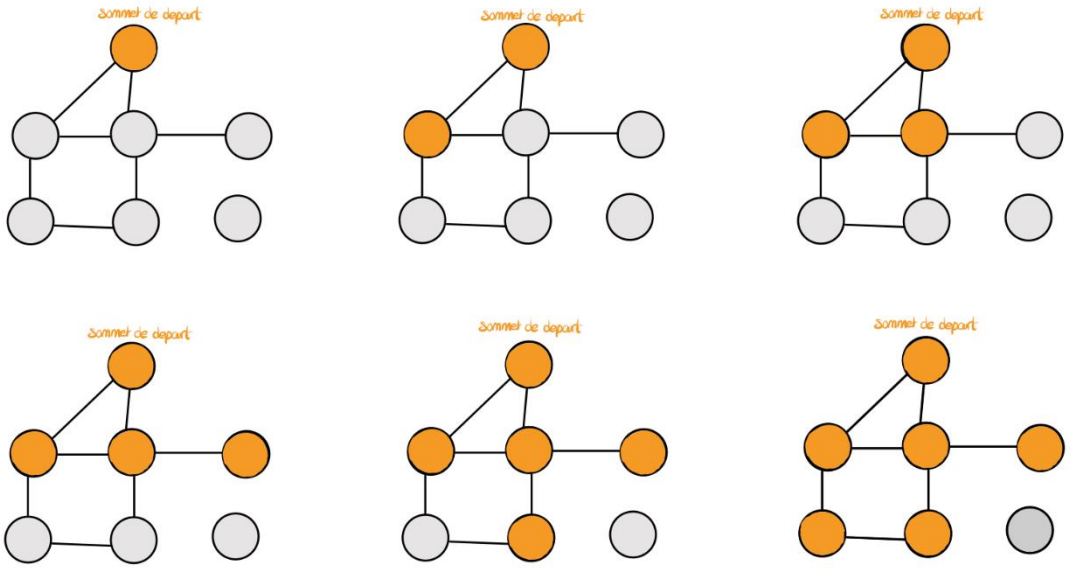
On initialise également la partie graphique dans ce fichier.

a. Connexité

Afin de vérifier la connexité du graphe, on utilises les méthodes de l'objet AlgoChemins : *visiterSommets*(Sommet sommet) et *estConnexe*().

Pour la fonction *visiterSommets* (Sommet sommet), on marque le sommet comme « true » et pour chaque arête reliée à ce sommet, tmp prend la valeur du sommet opposé et regarde si ce dernier a été marqué en « true ». S'il n'a pas été marqué, on appelle la fonction *visiterSommets* sur ce dernier. On répète ainsi l'opération pour tous les sommets reliés au sommet tmp.

Pour la fonction *estConnexe*() on appelle la fonction *visiterSommets*(Sommet sommet) avec le premier élément de listeSommets. Si tous les sommets n'ont pas été marqués à « true », la fonction return false ce qui signifie que le graphe n'est pas connexe. Sinon, on retourne true ce qui signifie que le graphe est connexe.



b. Le plus court chemin

La fonction plus court chemin a été inspiré du pseudo-code Wikipédia ci-dessous :

```
P := ∅
d[a] := +∞ pour chaque sommet a
d[s_deb] = 0
Tant qu'il existe un sommet hors de P
    Choisir un sommet a hors de P de plus petite distance d[a]
    Mettre a dans P
    Pour chaque sommet b hors de  P voisin de a
       Si d[b] > d[a] + poids(a, b)
           d[b] = d[a] + poids(a, b)
           prédécesseur[b] := a
    Fin Pour
Fin Tant Que
```

Afin de trouver le plus court chemin entre deux stations, on utilises les méthodes de l'objet AlgoChemins : *initCheminLePlusCourt*(HashMap<Integer,Sommet> listeSommets, int idSommetDepart) , *retirerLePlusPetit*(ArrayList<Sommet> sommetsAParcourir), *plusCourtChemin*(int idSommetDepart,int idSommetArrive).

- Pour la fonction *initCheminLePlusCourt*, on prend en argument une liste de sommets et l'id du sommet de départ. On initialise tous les sommets à une distance maximale et on les marqué à « false ». Une fois que tous les sommets

ont été initialisé, on met la distance du sommet de départ à 0 puis on retourne la liste de sommets.

- Pour la fonction *retirerLePlusPetit*, on prend en argument une liste de sommets à parcourir (par rapport à un sommet donné). On retire ensuite de la liste des sommets à parcourir le sommet le plus proche du sommet donné puis on retourne ce dernier.
- Pour la fonction *plusCourtChemin*, on prend en argument un sommet de départ et un sommet d'arrivée. On initialise l'algorithme avec la fonction *initCheminLePlusCourt*() puis on crée deux ArrayList pour les Sommets à parcourir et les sommets déjà parcouru. On récupère l'id du sommet de départ et d'arrivée avant d'entrer dans la boucle while. Tant qu'il reste des sommets qui n'ont pas été parcourus, on récupère le plus petit sommet avec la méthode *retirerLePlusPetit* puis on ajoute ce sommet à la liste des sommets parcourus. Une fois cela fait, nous récupérons les voisins du plus petit sommet et si la distance du voisin est > à la distance du PlusPetitSommet + poids de l'arête, on actualise la nouvelle distance du voisin avec la nouvelle valeur (distance du PlusPetitSommet + poids de l'arête) et on attribue le PlusPetitSommet comme prédécesseur du voisin.

c. Arbre couvrant de poids minimal

La fonction de l'arbre couvrant de poids minimal a été inspiré du pseudo-code Wikipédia ci-dessous :

```
Kruskal(G) :
1   A := ∅
2   pour chaque sommet v de G :
3      créerEnsemble(v)
4   trier les arêtes de G par poids croissant
5   pour chaque arête (u, v) de G prise par poids croissant :
6      si find(u) ≠ find(v) :
7         ajouter l'arête (u, v) à l'ensemble A
8         union(u, v)
9   renvoyer A
```

- Pour la fonction de l'ACPM, on prend en argument une liste d'arêtes. Tout d'abord, on crée un ensemble pour chaque station (une station = un nom de sommet). Ensuite, on trie les arêtes selon leur poids par ordre croissant. Pour chaque arête de la liste, on récupère les ensembles {U} et {V} : ils correspondent

aux ensembles des stations associés respectivement à une arrête. Si U ≠ V, on ajoute l'arête à la liste des arêtes les plus optimale et si d'autres stations ont le même ensemble que ceux de la liste des arêtes les plus optimales, alors on fusionne (union) les deux ensembles.

IV. Interface graphique

Afin de proposer à l'utilisateur une expérience plus intéressante que des affichages console, nous avons choisi de mettre en place une interface graphique pour notre projet Java. Notre choix s'est tourné vers Swing, une bibliothèque graphique pour le langage de programmation Java.

Pour faire simple, nous avons utilisé des classes JPanel et JFrame en les héritant, ces classes font office de fenêtre et permettent d'ajouter des composants tels que des boutons, des zones de texte etc. Ces classes sont localisées dans le package Views.

Ces composants peuvent permettre d'une gestion d'évènements : en implémentant des interfaces telles que MouseListener ou KeyListener, on peut appeler des méthodes et faire des traitements lors de clics de souris ou d'autres interactions avec la fenêtre. Les classes qui font cette gestion d'événements sont situées dans le package Controllers.

On dispose alors d'une interface graphique affichant la carte des métros parisiens, on peut afficher ou masquer l'acpm correspondant à ce graphe, ainsi que vérifier les plus courts chemins d'une station à une autre directement sur la carte.