TP 2-3 Programmation objet et Langage Java

ETN4

Thème : Développement d'un programme Objet

Objectif : Recherche d'itinéraire dans un métro

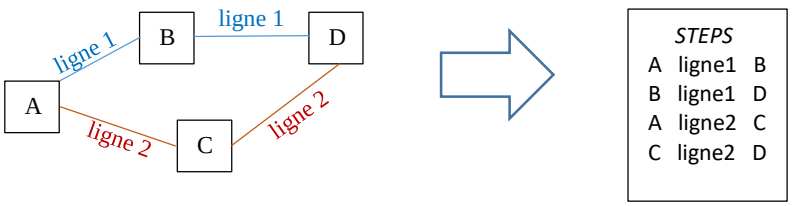
Fonctionnement du TP

Pour faciliter l'avancement du projet, certaines classes ont déjà été écrites. Aller sur MADOC pour récupérer le fichier zip contenant les classes et fichiers nécessaires au TP. Utiliser un dossier *data* pour y placer les fichiers correspondants. Installer les classes dans votre projet en respectant l'organisation des paquetages : créer deux nouveaux packages *tube* et *tube.gui*. Les classes de *tube.gui* peuvent être éventuellement consultées, mais il n'est pas recommandé de les modifier. Pour des raisons de compatibilité, vous devez respecter la signature des méthodes des deux classes figurant dans le paquetage *tube* : *Network* et *Control*. Les premières questions utilisent le mode console d'Eclipse pour l'affichage des résultats. La question 5 exploite l'interface graphique fournie par *tube.gui.TubeView.*
Le projet contient deux parties correspondant aux deux séances de TP.

Partie A : Définition du métro

1. Classe Step

Le réseau du métro est modélisé par les liens directs reliant les stations du réseau.



La classe *Step* modélise un triplet représentant deux stations successives sur une ligne, comme sur l'exemple suivant : `Paddington Bakerloo_line Marylebone`
Sur cet exemple, *Paddington* et *Marylebone* sont deux stations reliées directement par la ligne *Bakerloo*.
Ecrire la classe *Step* comprenant 3 attributs de type String (station1, ligne, station2) et les méthodes associées (constructeur, *getters*). Ecrire une méthode *getNext(String station)* qui renvoie la station opposée : si l'instance représente le triplet (A L B), alors *getNext(A)* renvoie B et *getNext(B)* renvoie A. La méthode renvoie *null* si l'instance de *Step* ne contient pas l'argument *station*. Ecrire une méthode de test de la classe.

2. Classe Network

2.1 Lecture du réseau

La classe *Network* modélise le métro en mémorisant les stations et leurs liens directs. Elle possède un attribut *steps* de type tableau d'instances de *Step*. Les données du métro sont stockées dans le fichier *steps.txt* du dossier *data* (ouvrir le fichier pour examiner son contenu). Ecrire une méthode de lecture du fichier en utilisant la classe *ReadTabFile* déjà abordée au tp1. Cette méthode récupère les valeurs du fichier, les convertit en instances de type *Step*, puis les stocke dans le tableau *steps*, comme indiqué par l'algorithme suivant :

```
Lire le fichier steps.txt avec la méthode readTextFile de TabFileReader
nsteps= nombre de lignes du fichier (méthode nrow() de TabFileReader)
Pour i de 0 à nsteps-1 faire
      Lire les 3 valeurs A, L, B de la ligne i du fichier en utilisant la méthode
      wordAt(i,j) de TabFileReader pour j=0..2
      Créer une nouvelle instance de Step (new Step(A,L,B)) et la stocker dans le tableau steps
```

Ecrire des méthodes pour faciliter la lecture de cet attribut (*nsteps()* pour renvoyer le nombre d'éléments ; *stepAt(int index)* pour récupérer l'instance de *Step* relative à un index donné). On créera une méthode de test qui affichera les instances de *steps*.

2.2 Extraction des stations

La classe *Network* possède un autre attribut qui mémorise la liste des noms de stations du métro. On utilisera pour cela la classe *TermList* du paquetage *utilitaires*. Comme son nom l'indique, cette classe gère une liste de termes (de type String ; ici des noms de stations). On conseille de lire la méthode *main* de la classe *TermList* pour avoir un aperçu de son utilisation. Ecrire une méthode qui crée la liste de toutes les stations en parcourant l'attribut *steps* défini à la question précédente. On respectera la règle suivante : le nom d'une station ne doit être présent qu'une seule fois dans la liste. Ecrire la méthode *numberOfStations* qui renvoie le nombre de stations de la liste. On créera une méthode de test qui affichera le nombre et le nom des stations.

3. Classe Lines

Cette classe a pour objet d'aider l'usager du métro. Elle mémorise la liste des lignes et est capable d'extraire la liste des noms de stations appartenant à une ligne donnée.
La classe *Lines* possède comme attribut une instance de *Network*, donnée en paramètre de constructeur. Ce dernier appelle une méthode privée qui parcourt les instances de *Step* du métro et mémorise la liste des noms de lignes. Pour mémoriser la liste des lignes, on procédera de la même manière que pour les stations (création d'un attribut *lignes* de type *TermList*).

Ecrire une méthode qui recherche toutes les stations appartenant à une ligne donnée :

```
public TermList findStations(int numLigne)
```

La méthode renvoie un objet de type *TermList* et possède un argument entier *numLigne* indiquant le numéro de ligne concerné (et non pas le nom de la ligne pour des raisons de compatibilité avec le code de l'interface graphique (question 5)). Pour retrouver le nom de la ligne, il suffit d'appeler l'instruction suivante : `String line=lignes.termAt(numLigne);`

Remarque : le numéro *numLigne* correspond à l'ordre alphabétique d'apparition des lignes dans le fichier *steps* et à l'ordre des lignes dans l'attribut *lines*.
Ecrire une méthode de test de la classe qui (1) affiche la liste des lignes sous la forme suivante :

```
Liste des lignes :
0 : Bakerloo_line
1 : Central_line
2 : Circle_line
3 : Hammersmith_line
4 : Jubilee_line
5 : Nothern_line
6 : Piccadilly_line
7 : Victoria_line
```

(2) affiche les stations d'une ligne donnée (l'utilisateur saisit le numéro de ligne souhaité).

Partie B : Recherche d'itinéraires

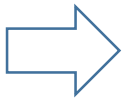
4. Classe Itinerary

Cette classe permet de rechercher un itinéraire entre deux stations du métro. La recherche de plus court chemin dans un graphe est un problème classique en algorithmie qui ne sera pas traité dans cette question en raison de la complexité de son implémentation. Nous proposons d'utiliser une technique de marche aléatoire dans un graphe : à chaque pas élémentaire, un voyageur choisit une station voisine au hasard en ne respectant qu'une seule contrainte : ne jamais revenir sur une station déjà visitée[1]. Une telle marche aléatoire ne garantit pas que le voyageur teste toutes les stations du métro : ce dernier peut arriver à une fin de ligne sans possibilité de revenir en arrière. Pour avoir plus de réussite, on créera un ensemble de marches aléatoires et on retiendra celle qui donne le meilleur résultat : arrivée à destination avec un nombre minimal de stations parcourues[2].

Avant de présenter la méthode de recherche, nous allons d'abord aborder un de ces points particuliers : le choix de la station suivante à partir d'une station *s.* En principe, ce choix devrait nécessiter de connaître l'ensemble des stations voisines et de tirer au hasard l'une d'entre elles. Pour éviter cette difficulté, on propose de parcourir les arcs du réseau dans un ordre aléatoire ; il suffit donc de prendre la première station disponible qui se présente, comme le montre l'exemple ci-dessous :

```
Ordre normal des indices :
0,1,2,3,4,5

Ordre dans le tableau steps
de Network :
0       A       L1      C
1       C       L1      E
2       E       L1      B
3       B       L1      D
4       C       L2      F
5       F       L2      B
```



```
Permutation des indices :
0,5,1,3,4
Ordre de lecture :

0       A       L1      C
5       F       L2      B
1       C       L1      E
3       B       L1      D
4       C       L2      F
2       E       L1      B
```

Supposons dans cet exemple que la station courante est s=B. Ses voisins sont {E,D,F} (tableau de gauche). On permute aléatoirement l'ordre de lecture des *steps*, puis on parcourt le tableau dans cet ordre aléatoire jusqu'à rencontrer la station B[3]. Sur le tableau de droite, la deuxième valeur de *step* contient B et son voisin est F. On choisira donc F comme station suivante.

[1] Cette contrainte permet d'avoir des trajets optimaux sur une ligne qui ne comporte pas de correspondances.
[2] Ce critère de qualité peut impliquer un nombre de changement de lignes préjudiciables (amélioration à apporter en question optionnelle).
[3] On ignore sur cet exemple la contrainte de non-retour sur ses pas.

Note : pour lire aléatoirement les *steps* d'une instance de *Network*, il suffit de respecter les instructions suivantes :

```
int []order={0,5,1,3,4,2}
Pour i de 0 à nsteps-1 faire
      int index=order[i];
      Step step=network.stepAt(index);
```

Pour affecter au tableau *order* des valeurs aléatoires, on utilisera la méthode *swap()* de la classe *RandomIndexes* du paquetage *utilitaires*. Consulter les instructions indiquées dans la méthode main pour avoir un exemple de fonctionnement.

Nous pouvons aborder maintenant l'algorithme de recherche d'un itinéraire. La méthode utilise les variables suivantes :

- *visited*, une instance de *TermList* : la liste mémorise les stations visitées durant le parcours. Cet objet sert à la fois de marqueur pour ne pas revenir sur ses pas, et de variable résultat indiquant le trajet effectué.
- *current,* la station courante (initialement la station de départ).

La méthode de recherche contient deux arguments : *départ* et *arrivée* qui sont les stations relatives au trajet souhaité par l'usager. L'algorithme général peut s'écrire de la manière suivante :

1. *visited* :=Ø (aucune station visitée initialement)
2. *current* :=*départ*
3. *fini* :=faux
4. **TantQue non** *fini* **faire** :
5. ajouter *current* à la liste *visited*
6. **si** *current*=*arrivée* **alors renvoyer** *visited* (fin fructueuse de la recherche)
7. **sinon**
8. rechercher la station suivante *next* à partir de *current*.
9. **si** next=null **alors** fini :=vrai (le trajet s'arrête sur un cul de sac)
10. **sinon** current :=next (on progresse d'une station sur le parcours)
11. **renvoyer** null (recherche infructueuse)

L'algorithme pour la recherche de la station suivante *next* (ligne 8 ci-dessus) s'écrit sous la forme :

permuter aléatoirement le tableau *order* (ordre de parcours des *steps*)
next := null (station suivante non encore trouvée)
i :=0
TantQue next=null **et** i<nsteps **faire** :
index=order[i]
step= network.stepAt(index)
nextS=step.getNext(current) (station opposée sur le lien step)
si nextS ≠ null **et** nextS n'existe pas dans *visited* **alors**
next=nextS (on a trouvé une nouvelle station)
i++

Cette méthode de recherche correspond à une seule marche aléatoire. Ecrire une méthode qui effectue un nombre *n* de marches aléatoires et recherche celle qui donne le meilleur résultat (critère : minimisation du nombre de stations donné par la taille de la liste *visited*).

5. Interface Graphique et la classe Control

Lancer l'interface graphique (main de la classe *TubeView* dans le paquetage *tube.gui*) pour comprendre le fonctionnement de ce programme servant de base à votre propre travail.

Une vue du métro s'affiche, avec la liste des lignes figurant en haut et à droite de l'image. Cliquer sur une ligne : un message apparaît sur la console, indiquant le numéro de la ligne (paramètre *numLigne* décrit dans la question 3). De même en cliquant sur une station, un message sur la console fournit le nom de la station de départ pour la recherche d'un itinéraire. La sélection d'une deuxième station entraîne un message identique et une fenêtre de dialogue indiquant le caractère infructueux de la recherche. Enfin, un clic sur l'icône du métro *Underground* génère un message d'effacement des stations. Le but de cette question est d'utiliser le travail précédent pour répondre graphiquement aux actions de l'utilisateur. Il n'est pas recommandé de modifier la classe *TubeView* : c'est la classe *Control* qui gère toutes les requêtes via deux méthodes exposées ci-dessous.

5.1 Affichage des stations d'une ligne

La méthode *showLine(int numLigne)* de la classe *Control* récupère le numéro de ligne sélectionné par l'utilisateur. Vous n'avez pas à appeler cette méthode : elle est lancée automatiquement par la classe *TubeView*. Le code actuel de la méthode consiste à afficher le numéro de ligne, de créer un objet de type *TermList* censé contenir la liste des noms de stations de la ligne. En l'état, l'objet est référencé par la valeur *null*. Cet objet est ensuite retourné à la classe *TubeView* via sa méthode *show()*. Compléter le code en utilisant la classe *Lines* pour rechercher la liste effective des stations.

5.2 Affichage d'un itinéraire.

La méthode *showItinerary* de la classe *Control* a pour but d'afficher le parcours entre deux stations définies par les attributs *begin* et *end*. La méthode est appelée deux fois (à chaque clic de l'utilisateur : une fois pour le départ ; une fois pour l'arrivée). Son action est contextuelle : pour la station de départ, elle affiche celle-ci ; pour la station d'arrivée, tout le trajet est affiché. La seule ligne à modifier est celle définissant la liste *selection* donnant les stations du trajet. Utiliser la classe *Itinerary* pour rechercher cette liste.

Questions optionnelles

En fonction du temps imparti, vous pouvez traiter la question de votre choix, sachant que la question B.2 est la plus complexe.

A. Classe Analysis : caractérisation de la complexité du réseau

1. Indiquer pour chaque station son degré : nombre de stations directement voisines.
2. Analyser le réseau en recherchant les trajets entre toutes les paires de stations du réseau (méthode de la question 4). Donner la longueur moyenne d'un trajet (en nombre de stations) ainsi que la longueur maximale et la paire de stations concernée.
3. Utiliser la recherche précédente pour calculer le nombre de fois qu'une station est visitée sur l'ensemble des trajets possibles : donner la station la plus fréquentée.

B. Amélioration de la recherche d'itinéraires

1. Le critère de recherche utilisé à la question 4 est le nombre de stations. Ce critère n'est pas très judicieux : il oblige parfois l'utilisateur à changer inutilement de ligne. Utiliser un critère plus réaliste en estimant le temps de trajet : compter 2 minutes par station et 5 minutes par changement de ligne. Rechercher le trajet qui minimise le temps de parcours.
2. La méthode de marches aléatoires est facile à implémenter mais ne garantit pas de trouver l'itinéraire optimale. Proposer une méthode déterministe de recherche du plus court chemin dans un graphe en s'inspirant de la littérature sur ce sujet.