Conservatoire National des Arts et Métiers
École SITI - Département Informatique

Année 2014–2015

UE NFP 136 (VARI 2)

Sujet du mini-projet 2014-2015 : Recherche d'itinéraires dans le métro parisien

Le but de ce mini-projet est d'implémenter en JAVA un algorithme de recherche du meilleur itinéraire dans le métro parisien, basé sur un parcours en largeur du graphe représentant le réseau du métro. Le programme JAVA ainsi obtenu sera capable d'indiquer à l'utilisateur la façon la plus rapide (en temps) de se rendre d'une station de départ à une station d'arrivée, en prenant en compte les éventuels changements nécessaires. Les informations concernant les stations du métro parisien, ainsi que ses 16 lignes, devront être récupérées dans un fichier texte qui vous sera fourni. Ce document a pour objectif de vous guider dans les différentes étapes de la réalisation de ce mini-projet, et est à lire attentivement. Il est d'ailleurs à noter que la dernière étape est la seule à nécessiter de savoir ce qu'est un graphe et un parcours en largeur d'un graphe. Pour une description détaillée du parcours en largeur à l'aide d'une file, se référer au cours et à l'ED correspondants.

Présentation du projet

On considère donc dans ce projet un réseau de métro (ici, le métro parisien), que l'on modélisera à l'aide d'un graphe orienté. Ainsi, rechercher un itinéraire dans le réseau reviendra à rechercher un chemin dans le graphe associé, ce qui peut se faire à l'aide d'algorithmes de parcours vus en cours. Idéalement, les stations seraient les sommets, et les arcs seraient formés par les couples de stations reliées entre elles par les lignes de métro. Dans les faits, on veut pouvoir prendre en compte les temps moyens de parcours entre stations (le temps de trajet entre deux stations consécutives d'une ligne étant par défaut 2 minutes, mais dans certains cas cela peut aller jusqu'à 5 minutes), et on sera donc amené à introduire des sommets "fictifs", qui ne correspondent pas à des stations, de façon à ce que chaque arc du graphe obtenu corresponde à 1 minute de trajet.

Par exemple, aller de *Réaumur - Sébastopol* à *Arts et Métiers* par la ligne 3 du métro nécessite en moyenne 2 minutes, et on crée donc un sommet fictif entre ces deux stations, puis deux arcs (un dans chaque sens) entre *Réaumur - Sébastopol* et ce sommet fictif, et deux arcs (un dans chaque sens) entre ce sommet fictif et *Arts et Métiers*. Ainsi, se rendre de *Réaumur - Sébastopol* à *Arts et Métiers* nécessite d'emprunter deux arcs, et correspond donc bien dans ce graphe à un trajet de 2 minutes.

De même, on veut pouvoir prendre en compte les temps de trajet supplémentaires induits par les changements, et il faut donc modifier quelque peu le graphe représentant le réseau de métro. Concrètement, à chaque station du métro seront associés autant de sommets du graphe que de lignes passant par ce sommet. Ainsi, comme seule la ligne 3 passe par la station "Temple", un seul sommet du graphe portera ce nom. À l'inverse, 5 lignes différentes passent par la station "République" (les lignes 3, 5, 8, 9 et 11), et il y aura donc dans le graphe 5 sommets différents qui s'appelleront "République" (qui se trouveront donc respectivement sur les lignes 3, 5, 8, 9 et 11).

Ensuite, on complète la modification du graphe en ajoutant deux arcs (un dans chaque sens) entre chacune des 10 paires de sommets formées sur ces 5 sommets, puis on subdivise chacun de ces arcs à l'aide de sommets fictifs (voir ci-dessus), de façon à prendre en compte le temps nécessaire pour effectuer le changement. Par exemple, en supposant que chacun des changements possibles à la station République s'effectue en 5 minutes, on devra utiliser en tout $10 * 2 * 5 = 100$ arcs pour modéliser tous les changements possibles et leurs durées. On peut d'ailleurs observer que toutes ces modifications ont pour conséquence d'ajouter un nombre important d'arcs et de sommets (par rapport au réseau de métro considéré au départ).

Enfin, il convient de noter que le graphe obtenu est bien un graphe orienté, car certaines liaisons du réseau de métro parisien sont à sens unique. Par exemple, la liaison entre les stations "Michel Ange Auteuil" et "Porte d'Auteuil" sur la ligne 10 ne fonctionne que dans ce sens-là (et ne permet donc pas de se rendre de "Porte d'Auteuil" à "Michel Ange Auteuil"). **La dernière remarque à formuler est qu'on ne vous demande pas concrètement de transformer le réseau du métro parisien à l'aide de ces opérations** : le fichier fourni les prend déjà toutes en compte, et vous n'avez donc qu'à utiliser les informations contenues dans ce fichier. Ces explications n'ont donc pour seul objectif que d'expliquer et de justifier le contenu de ce fichier, pour que vous sachiez comment il a été généré.

Étape 1 : la classe `SommetMetro`

On se propose dans un premier temps d'écrire une classe `SommetMetro`, qui n'est destinée qu'à contenir les informations relatives à un sommet du réseau du métro. En d'autres termes, chaque objet de cette classe contiendra uniquement un certain nombre de variables d'instance (appelées aussi *attributs*), les accesseurs en lecture et en écriture associés, et un constructeur. Aucune autre méthode ne sera a priori nécessaire. On rappelle par ailleurs que le bon usage en JAVA est de déclarer ces 6 attributs comme `private`.

Voici les 6 attributs qui seront utiles. Les 3 premiers sont des informations liées à l'identité du sommet (station), et sont les suivants :

1. Une variable entière représentant **le numéro (ou l'identifiant) du sommet**, et compris entre 1 et le nombre total de sommets.

2. Une variable de type `String` contenant **le nom du sommet** (par exemple "République" ou "Nation"). On rappelle que plusieurs sommets peuvent correspondre à une même station (ce point a été justifié précédemment), et peuvent donc porter le même nom. Par ailleurs, les sommets fictifs (c'est-à-dire qui ne sont pas associés à des stations de métro) auront tous pour nom "SANS_NOM".

3. Une variable entière représentant **le numéro de la ligne de métro** à laquelle appartient ce sommet (chaque sommet n'appartient qu'à une seule ligne ; ce point a été justifié précédemment).

Les 3 autres attributs seront utilisés lors du parcours du graphe :

1. Une variable entière qui a vocation à représenter **la distance (en nombre d'arcs) entre le sommet (station) de départ et ce sommet**. La valeur de cette variable sera bien évidemment fixée lors du parcours en largeur.

2. Une variable booléenne, initialisée à *faux*, qui vaudra ***vrai* si ce sommet a déjà été marqué lors du parcours, et *faux* (sa valeur initiale) sinon**. La valeur de cette variable sera bien évidemment modifiée lors du parcours en largeur.

3. Une variable de type `SommetMetro`, qui représentera **le prédécesseur (sommet précédent) de ce sommet dans l'itinéraire**. La valeur de cette variable sera bien évidemment fixée lors du parcours en largeur.

Les valeurs des 3 premiers attributs seront fixées dès la création du sommet, et donc dans le constructeur, dont voici le prototype :

```
public SommetMetro(int, String, int)
```

Les valeurs des 3 autres attributs devront bien sûr être initialisées, mais elles seront modifiées durant le parcours. Quoi qu'il en soit, on devra également avoir la possibilité d'accéder aux valeurs de tous les attributs. Pour cela, on aura donc besoin d'accesseurs en lecture et en écriture :

- Une méthode `getNumero()` pour récupérer le numéro du sommet,
- Une méthode `getNom()` pour récupérer le nom du sommet,
- Une méthode `getLigne()` pour récupérer le numéro de la ligne où se trouve le sommet,
- Une méthode `getDistance()` pour récupérer la distance entre le sommet de départ et ce sommet,

- Une méthode `estMarque()`, qui retourne *vrai* si le sommet a déjà été marqué durant le parcours, et *faux* sinon,
- Une méthode `getPredecesseur()`, pour récupérer le sommet précédant ce sommet dans l'itinéraire,
- Une méthode `setDistance()`, pour modifier la distance entre le sommet de départ et ce sommet,
- Une méthode `marquer()`, pour indiquer que le sommet est marqué,
- Une méthode `setPredecesseur()`, pour récupérer le sommet précédant ce sommet dans l'itinéraire.

Grâce à cette classe, l'ensemble des sommets du graphe pourra être représenté à l'aide d'un tableau d'éléments de type `SommetMetro`.

Étape 2 : construction du graphe du métro à partir du fichier

La méthode décrite ici peut être intégrée à votre classe principale. Son objectif est de lire les informations contenues dans le fichier texte fourni, et de les utiliser pour créer le graphe dans lequel le parcours en largeur sera effectué. Voici quelques indications concernant le format du fichier à lire :

- **Le fichier contient de nombreuses lignes de commentaires (commençant par "######"). Néanmoins, il n'y a jamais deux lignes de commentaires consécutives.**
- **La première information à récupérer (elle se trouve sur la 2ème ligne) est n, le nombre de sommets du graphe.**
- Chacune des n lignes suivantes (si on ne prend pas en compte les lignes de commentaires, évidemment) contient la description d'un sommet du graphe, selon le format suivant :

  numéro_sommet:nom_sommet:numéro_ligne

  où numéro_sommet est son numéro (identifiant), nom_sommet son nom, et numéro_ligne le numéro de la ligne de métro à laquelle il appartient. Il faut noter la convention suivante : **le numéro associé à la ligne 3bis est 33, et celui associé à la ligne 7bis est 77**.
- **La ligne qui se situe juste après la ligne de commentaires qui suit ces n lignes (de nouveau, sans tenir compte des lignes de commentaires) contient m, le nombre d'arcs du graphe.**

- Chacune des m lignes suivantes contient la description d'un arc (x, y) du graphe, selon le format suivant :

  numéro_arc:x:y

  où numéro_arc est le numéro de cet arc (x, y), et x et y les numéros des sommets qui forment les deux extrémités de cet arc.

Sur chacune des $n+m$ lignes contenant 3 informations, ces dernières sont séparées par ":". Voici le début du fichier fourni (appelons-le "toto.txt") :

```
###### Liste des sommets (ligne suivante = nombre de sommets)
1026
###### Stations Ligne 1
1:Grande Arche de La Défense:1
2:SANS_NOM:1
3:Esplanade de La Défense:1
4:SANS_NOM:1
5:Pont de Neuilly:1
```

Le graphe ainsi décrit contient 1026 sommets (2ème ligne). Pour lire des données dans un fichier, on peut utiliser un objet de la classe `BufferedReader`. Voici comment initialiser un objet `lecteur` de cette classe, pour accéder aux informations stockées dans le fichier de nom "toto.txt" :

```
BufferedReader lecteur = new BufferedReader(new FileReader("toto.txt"));
```

En appelant la méthode `readLine` (qui renvoie un objet de la classe `String`) sur cet objet, on lira le contenu de la ligne suivante du fichier de nom "toto.txt" (la lecture d'un fichier étant séquentielle). Ainsi, après le premier appel à cette méthode, la variable `ligne` contiendra la valeur `"###### Liste des sommets (ligne suivante = nombre de sommets)"` (qui correspond à une ligne de commentaires) :

```
String ligne=lecteur.readLine();
```

Après un 2ème appel, la variable `ligne` contiendra la valeur "1026" :

```
ligne=lecteur.readLine();
```

La construction d'un objet de la classe `BufferedReader` est susceptible de générer une exception (si le fichier de nom "toto.txt" n'existe pas, par exemple), et il serait donc préférable de gérer toutes ces instructions dans un bloc `try{...} catch(Exception e){...}`. Enfin, après utilisation, cet objet `lecteur` doit être fermé, à l'aide de l'instruction `lecteur.close();`.

Une fois les informations d'une ligne associée à un sommet récupérées, il faudra les stocker dans une variable de la classe `SommetMetro`. Pour cela, on vous rappelle l'existence de :

- la méthode `split` de la classe `String`, qui permet de "découper" l'objet de la classe `String` sur lequel la méthode est appelée en fonction du séparateur passé en paramètre, et qui stocke les objets de la classe `String` qui résultent de ce découpage dans un tableau.
- la méthode de classe `parseInt()` de la classe `Integer`, qui permet de convertir en entier (variable de type `int`) l'objet de la classe `String` passé en paramètre (par exemple, "2" sera converti en l'entier 2).

(Pour plus d'information sur ces deux méthodes, se référer à l'énoncé du deuxième TP.) **De même, il faut garder trace de l'existence de chaque arc stocké dans le fichier.** On peut le faire simplement en déclarant une variable `listeArcs` de type `boolean[][]` (*matrice d'adjacence*) : l'existence d'un arc du sommet $i+1$ vers le sommet $j+1$ se traduira alors par le fait que `listeArcs[i][j]` sera égal à *vrai* (il sera égal à *faux* sinon).

Étape 3 : la classe `FileSommets`

L'objectif de cette classe est de définir une file (structure FIFO) capable de stocker, non pas des entiers, mais des objets de la classe `SommetMetro`.

Cette classe est donc identique à la classe `File` codée en TP (séance 5), à ceci près que les objets manipulés sont de type `SommetMetro`, et non `int`.

Étape 4 : parcours en largeur du graphe

La dernière grande étape du projet consiste à implémenter une méthode (qui peut aussi être intégrée à votre classe principale) effectuant un parcours en largeur du graphe obtenu, à partir d'un sommet donné, en utilisant une file (objet de la classe `FileSommets`). On rappelle le principe d'un tel parcours :

1. Définir une file vide ;
2. Enfiler le sommet de départ, et le marquer ;
3. Tant que la file n'est pas vide faire :
   - Défiler le sommet s en tête de file ;
   - Pour chaque sommet s' du graphe faire :
     - S'il existe un arc (s, s') et si s' n'est pas marqué alors :
       * Enfiler et marquer s' ;
       * Poser prédécesseur(s') := s ;
       * Poser distance(s') := distance(s) + 1 ;

Cette étape nécessite d'utiliser tout ce qui a été implémenté aux étapes précédentes. Dans le cadre de ce projet, on pourra éventuellement accélérer le parcours en largeur, par exemple en l'arrêtant dès qu'un sommet donné (la destination) a été atteint.

Touches finales

Pour compléter le projet, il reste à écrire le programme principal. Ce dernier récupérera les trois paramètres suivants (dans cet ordre), au moment de l'appel du programme :

1. le nom du fichier de données,
2. le numéro du sommet de départ,
3. le numéro du sommet d'arrivée.

Il faudra ensuite construire le graphe du métro (étape 2), et effectuer un parcours en largeur de ce graphe (étape 4), en partant du sommet de départ. Enfin, on affichera les noms des sommets (stations) de départ et d'arrivée, ainsi qu'un certain nombre d'informations concernant l'itinéraire calculé :

- Sa durée (en minutes, c'est-à-dire en nombre d'arcs),
- La liste (ordonnée) des noms des stations de métro parcourues entre le départ et l'arrivée. On n'affichera pas les noms des sommets fictifs (de nom "SANS_NOM"), mais uniquement ceux des sommets qui correspondent à de véritables stations de métro. On affichera également la ligne sur laquelle se trouve la station considérée, ainsi que la durée du trajet depuis la station précédente dans l'itinéraire.
- Pour finir, on indiquera explicitement chacun des changements (un changement correspond -pour simplifier, car il y a des exceptions- à deux stations portant le même nom, et qui sont consécutives dans l'itinéraire), ainsi que leurs durées.

Voici par exemple ce qu'affiche un tel programme si le sommet de départ est 236 (Richard Lenoir, ligne 5) et celui d'arrivée 303 (Picpus, ligne 6) :

```
Départ de Richard Lenoir
Arrivée à Picpus

Votre itinéraire, dont la durée est de 19 minutes, est le suivant :

Station 1 : Richard Lenoir (ligne 5)
 [Trajet de 2 minutes]
Station 2 : Bréguet - Sabin (ligne 5)
 [Trajet de 2 minutes]
Station 3 : Bastille (ligne 5)
>>>>>> CHANGEMENT [3 minutes] à Bastille (ligne 5 --> ligne 1)
 [Trajet de 2 minutes]
Station 4 : Gare de Lyon (ligne 1)
```

```
 [Trajet de 2 minutes]
Station 5 : Reuilly - Diderot (ligne 1)
 [Trajet de 2 minutes]
Station 6 : Nation (ligne 1)
>>>>>> CHANGEMENT [4 minutes] à Nation (ligne 1 --> ligne 6)
 [Trajet de 2 minutes]
Station 7 : Picpus (ligne 6)
```

Pour simplifier l'affichage des détails de l'itinéraire, on fournit ici le code d'une méthode récursive calculant la liste des sommets de l'itinéraire, à partir des informations calculées lors du parcours en largeur :

```
public static void itineraire(SommetMetro origine, SommetMetro destination,
int indiceSommet, SommetMetro[] etapesItineraire) {
   etapesItineraire[indiceSommet]=destination;
   if(origine.getNumero()!=destination.getNumero())
      itineraire(origine, destination.getPredecesseur(), indiceSommet-1,
      etapesItineraire);
}
```

Tous les sommets du chemin (itinéraire) reliant l'origine (sommet de départ) à la destination (sommet d'arrivée) seront alors stockés dans le tableau `etapesItineraire` (l'origine tout à gauche, et la destination tout à droite).

L'appel récursif initial (à faire dans le programme principal) sera alors :

```
itineraire(origine, destination, destination.getDistance(), mesEtapes);
```

(où `mesEtapes` est un tableau à déclarer et allouer).

Le projet est à rendre par e-mail, à l'adresse : `cedric.bentz@cnam.fr`. Lors du rendu, il faudra bien évidemment fournir le code source commenté de toutes vos classes (fichiers `.java`), et leur bytecode (fichiers `.class`).