Licence 2 – MIASHS
Algorithmique et programmation par objets
Jérôme David
Durée : 2 heures

barème donné à titre indicatif (sur 25 points)

TD 9 (4 pages)

L'objectif de ce travail est de réaliser un programme calculant le plus court chemin entre deux villes. Cet algorithme appelé algorithme de Dijkstra, est utilisé par de nombreuses applications dont la recherche de chemin pour navigateur GPS.

Introduction

Pour réaliser ce programme, trois classes vont être réalisées. La première représentera une ville. Une ville a un nom, un tableau de villes voisines (i.e. reliées par une route directe) et un tableau de distances vers ces villes voisines. Une deuxième classe représentera un chemin entre deux villes. Un chemin possède une distance ainsi que le tableau des villes traversées. La première ville de ce tableau est la ville de départ, la dernière est la ville d'arrivée, et les villes entre les deux sont les villes intermédiaires. Un chemin qui ne contient qu'une seule ville est un chemin de distance 0.

La dernière classe contiendra les méthodes pour la réalisation de l'algorithme principal.

1- Classe `Ville`

Son squelette est le suivant :

```
public class Ville {
   private String nom;
   private Ville[] voisines;
   private int[] distances;
   private int nbVoisines;

   public Ville(String nom) {...}
   public void ajouteVoisine(Ville v, int distance) {...}
   public int getNbVoisines() {...}
   public Ville getVoisine(int idx) {...}
   public int getDistance(int idx) {...}
   public String toString() {...}
}
```

L'attribut `voisines` contient l'ensemble des voisines d'une ville. La distance entre cette ville et sa voisine d'indice `i` est stockée dans le tableau `distances` à l'indice `i` (les deux tableaux ont donc toujours la même taille). L'attribut `nbVoisines` désigne l'ensemble des voisines réellement présentes dans `voisines`.

1.1- Le constructeur (1 point)

Réaliser le constructeur de la classe ville. Les tableaux `voisines` et `distances` ont une capacité initiale de 10 éléments. `nbVoisines` est initialisé à 0.

1.2- Méthode `int getNbVoisines()` (0,5 point)

Réaliser la méthode `getNbVoisines()` qui retourne la valeur de l'attribut `nbVoisines`.

1.3- Méthodes `Ville getVoisine(int idx)` et `int getDistance(int idx)` (2 points)

Réaliser les deux méthodes `getVoisine(int idx)` et `getDistance(int idx)`. La première méthode retourne la ville voisine d'indice `idx` de l'attribut voisines. La méthode distance retourne la distance la distance d'indice `idx`. Si l'indice fourni est supérieur ou égal au nombre de voisines **réellement présentes** alors une exception de type `ArrayIndexOutOfBoundsException` sera levée.

1.4- Méthode `String toString()` (0,5 point)

Réaliser la méthode `toString()` qui retourne le nom de la ville.

1.5- Méthode `boolean equals(Object o)` (bonus TD)

Réaliser la méthode `equals` qui retourne `true` que si les villes possèdent le même nom

1.6- Méthode `void ajouteVoisine(Ville v, int distance)` (4 points)

Réaliser la méthode `ajouteVoisine(Ville v, int distance)`. Cette méthode permet d'ajouter la ville en paramètre à l'ensemble des voisines ainsi que sa distance à l'attribut distance. Si la ville passée en paramètre est la même que la ville sur laquelle la méthode est appelée alors on lèvera l'exception de type `RuntimeException` avec le message `"Pas de route entre la même ville"`.

On lèvera également une exception de type `NullPointerException` si la ville passée en paramètre est nulle.

Une ville ne peut être présente que une seule fois dans le tableau de voisines.

Les tableaux seront agrandis (+10 éléments) si besoin. Les routes sont toutes à double sens, il faudra penser à ajouter la ville sur laquelle la méthode a été appelée aux voisines de la ville v passée en paramètre.

2- Classe Chemin

```
public class Chemin {
   private Ville[] villes;
   private int distance;

   public Chemin(Ville villeDepart) {...}
   private Chemin(Chemin debut, Ville villeSuivante, int distance) {...}
   public Ville villeArrivee() {...}
   public boolean contient(Ville v) {...}
```

```
    public int getDistance() {...}
    public Chemin[] etendre() {...}
    public String toString() {…}
}
```

L'attribut `villes` représente les villes constituant le chemin. `villes[0]` est la ville de départ, et `villes[villes.length-1]` est la ville d'arrivée. L'attribut `distance` est la somme des distances entre les villes.

2.1- Constructeur `Chemin(Ville villeDepart)` (1 point)

Réaliser le constructeur `Chemin(Ville villeDepart)` qui initialise un chemin avec une seule ville. Un tel chemin à une distance de 0.

2.2- Constructeur `Chemin(Chemin debut, Ville villeSuivante, int distance)` (2 points)

Réaliser le constructeur privé `Chemin(Chemin debut, Ville villeSuivante, int distance)` qui initialise un chemin qui est la concaténation du chemin `debut` et de la `villeSuivante` passés en paramètre. La longueur de ce chemin sera la longueur du chemin début plus la distance passée en paramètre.

2.3- Méthodes `int getDistance()` et `Ville villeArrivee()` (1 point)

Réaliser les méthodes `getDistance()` et `villeArrivee()` qui retournent respectivement la distance du chemin et la dernière ville du chemin.

2.4- Méthode `String toString()` (1 point)

Réaliser la méthode `toString()` qui retourne une chaîne de caractères représentant le chemin. Un chemin constitué des villes A, C, D de distance 345 sera représenté par la chaîne « Chemin [A, C, D] d=345 ».

2.5- Méthode `boolean contient(Ville v)` (1 point)

Réaliser la méthode `contient(Ville v)` qui retourne vrai si la ville passée en paramètre est contenue dans le chemin.

2.6- Méthode `Chemin[] etendre()` (3 points)

La méthode `etendre()` calcule et retourne l'ensemble des chemins possibles par l'extension du chemin actuel vers les villes voisines de l'arrivée du chemin actuel. Les chemins sont acycliques : la nouvelle ville d'arrivée ne doit pas être contenue dans le chemin actuel. La taille du tableau retourné devra être égale au nombre de chemin possibles (il n'y a pas de « place vide »).

Exemple : Soit le chemin [A, C, D], les voisines de D sont C, E, F. Les chemins possibles retournés par la méthode étendre seront [A, C, D, E] et [A, C, D, F].

3- Classe `PlusCourtChemin`

```
public class PlusCourtChemin {
  public static Chemin plusCourtChemin(Ville depart, Ville arrivee) {...}
```

```
  public static Chemin[] fusionTriee(Chemin[] trie, Chemin[] nonTrie)
                                                                {...}
}
```

Cette classe contient deux méthodes statiques permettant de réaliser l'algorithme de calcul du plus court chemin.

3.1- Méthode `Chemin[] fusionTriee(Chemin[] trie, Chemin[] nonTrie)` (4 points)

Réaliser la méthode `fusionTriee(Chemin[] trie, Chemin[] nonTrie)`. Cette méthode prend en paramètre deux tableaux. `trie` est un tableau de chemins trié par distance **décroissante**. `nonTrie` est un tableau de chemins non trié. La méthode devra fusionner les deux tableaux en **ignorant le dernier élément** du tableau `trie` (i.e. le plus court chemin). Le tableau résultat devra être trié par distance **décroissante**.

3.2- Méthode `Chemin plusCourtChemin(Ville depart, Ville arrivee)` (4 points)

Réaliser la méthode `plusCourtChemin(Ville depart, Ville arrivee)` qui retourne le plus court chemin entre la ville `depart` et la ville `arrivee` passées en paramètre. L'algorithme de calcul du plus court chemin est le suivant :

1. On commence avec un tableau de résultat contenant seulement le chemin constitué de la ville de départ.
2. Ensuite, il faut étendre le dernier chemin du tableau (i.e. le plus court),
3. puis fusionner les deux tableaux en utilisant la méthode `fusionTriee(…)` précédente.
4. Les étapes 2 et 3 sont répétées tant que la taille du tableau est supérieure à 0 et que la ville d'arrivée du dernier chemin du tableau n'est pas la ville d'arrivée passée en paramètre
5. Finalement, si le tableau résultat est vide alors il n'y a pas de chemin entre les deux villes et on retourne `null`, sinon le plus court chemin est le dernier chemin du tableau résultat.

Annexes

Classe Arrays :

`static Object[]`	**`copyOf`**`(Object[] original, int newLength)` Copies the specified array, truncating or padding with nulls (if necessary) so the copy has the specified length.
`static String`	**`toString`**`(Object[] a)` Returns a string representation of the contents of the specified array.

Un des constructeurs de la classe ArrayIndexOutOfBoundException :

> **`ArrayIndexOutOfBoundsException`**`(int index)`
>
> Constructs a new `ArrayIndexOutOfBoundsException` class with an argument indicating the illegal index.

Un des constructeurs de la classe RuntimeException :

RuntimeException(String message)

Constructs a new runtime exception with the specified detail message.

Un des constructeurs de la classe NullPointerException :

NullPointerException(String s)

Constructs a `NullPointerException` with the specified detail message.