TP de Java
Plus cours chemin vers Bilbao!

MEIM

2024-2025

L'objectif de ce TP est de vous offrir une première expérience de développement en objet à travers des exercices qui vont vous faire coder deux algorithmes classiques de théorie des graphes :

- l'algorithme du plus court chemin (algorithme de Dijkstra)
- et le parcours en profondeur

Lors des exercices, vous manipulerez les notions vues lors des cours 2 à 4 : champs, méthodes et constructeurs.

Contenu du TP

Dans ce TP, vous allez concevoir un petit programme de navigation permettant de se déplacer en voiture. Ce programme vous permet de calculer le chemin optimal permettant d'aller d'une ville à une autre en utilisant l'algorithme du plus court chemin de Dijkstra. Pour cela, vous avez besoin de manipuler des graphes pondérés. Un graphe est une structure mathématique dans laquelle des nœuds sont reliés par des arcs possédant un poids. Dans notre cas, les nœuds sont des villes, les arcs symbolisent les routes et le poids d'un arc donne la distance à parcourir sur la route.

Méthodes et champs de classe

Dans ce TP, il n'y a pas besoin d'utiliser ni d'attributs de classe ni de méthodes de classe (sauf pour la méthode `main` qui est nécessairement de classe). En d'autres termes, on vous demande de ne pas utiliser le mot clé **static**, sauf pour la méthode **main**.

Niveaux d'encapsulation

On vous demande de rendre privés tous les champs et publics tous les constructeurs et toutes les méthodes de vos classes. De façon générale, rendre systématiquement les champs privés est une bonne manière de programmer car vous éviterez ainsi d'utiliser des spécificités internes d'un objet. Pour les méthodes, on peut les définir privées, leur laisser la visibilité par défaut (celle du package) ou les définir publiques, tout dépend du contexte. Ici, la visibilité par défaut est suffisante.

Lisez attentivement le sujet une fois en entier avant de vous lancer dans la programmation.

Exercice 1 : Les villes et les routes

Dans cette première partie, nous construisons quelques villes et les relions entre elles. Une ville possède un nom et des coordonnées GPS. Elle est reliée à d'autres villes par des routes.

Question 1.a Dans un dossier nommé **shortestpath**, créez une classe **Main** dans le package **ensiie.shortestpath** et ajoutez y une méthode statique **main**.

Question 1.b Dans le même package, créez une classe nommée **Location** représentant une ville. Une instance de **Location** doit pour le moment posséder trois champs :

- un champ **name** donnant le nom de la ville,
- des champs **latitude** et **longitude** de type **double** donnant les coordonnées GPS de la ville.

Question 1.c Ajoutez à la classe **Location** un constructeur permettant d'initialiser les champs **name**, **latitude** et **longitude** de votre ville. Ajoutez aussi une méthode d'instance **display** permettant d'afficher la ville avec ses coordonnées GPS. Dans la méthode main, créez une variable locale nommée **evry**, et référençant la ville d'Evry qui se trouve aux coordonnées (48.629828, 2.4417819999999892). Vérifier que l'affichage de votre ville est correct.

Question 1.d Les coordonnées GPS sont données au constructeur en degré, mais il est beaucoup plus commode de manipuler des radians. Pour cette raison, lorsque vous initialisez les champs latitude et longitude d'une instance d'une ville, il faut convertir les paramètres. On vous rappelle que si **deg** est donné en degré, $(\pi \times \text{deg})/180$ donne l'équivalent en radian. En Java, **Math.PI** donne la valeur de π. Vérifiez que Evry se trouve bien aux coordonnées (0.8487506132785291, 0.04261713551593199) en radian.

Question 1.e On souhaite maintenant calculer la distance entre deux villes en kilomètres. La formule permettant de calculer la distance entre deux points d'une sphère dont les coordonnées sont données en radian est :

```
R * (Math.PI/2 - Math.asin(Math.sin(lat2) * Math.sin(lat1) + Math.cos(long2
    - long1) * Math.cos(lat2) * Math.cos(lat1)));
```

Dans cette formule, R est le rayon de la terre (6378km). Cette formule utilise directement les méthodes de la class Math de Java permettant d'effectuer des opérations de trigonométrie.

Écrivez une méthode d'instance **double distanceTo(Location to)** renvoyant la distance en kilomètres entre le receveur de l'appel (c'est-à-dire this) et to.

Pour tester votre méthode, créez une variable locale nommée **paris** dans votre main pour référencer la ville de Paris se trouvant aux coordonnées (48.85661400000001, 2.3522219000000177). Ensuite, vérifier que la distance à vol d'oiseau entre Evry et Paris est bien de 26 kilomètres environ.

Question 1.f En plus d'Evry et Paris, ajoutez quelques villes à votre main en copiant-collant le code ci-dessous :

```
Location lemans = new Location("Le Mans", 48.00611000000001, 0
    .1995560000000296);
Location orleans = new Location("Orleans", 47.902964, 1.9092510000000402);
Location angers = new Location("Angers", 47.478419, -0.5631660000000238);
Location tours = new Location("Tours", 47.39414399999999, 0
    .6848400000000083);
Location bourges = new Location("Bourges", 47.081012, 2.398781999999983);
Location poitiers = new Location("Poitiers", 46.58022400000001, 0
    .34037499999999454);
Location limoges = new Location("Limoges", 45.83361900000001, 1
    .2611050000000432);
Location angouleme = new Location("Angouleme", 45.648377, 0
    .15623690000006718);
Location bordeaux = new Location("Bordeaux", 44.837789,
    -0.5791799999999512);
Location agen = new Location("Agen", 44.203142, 0.6163629999999785);
Location toulouse = new Location("Toulouse", 43.604652, 1.4442090000000007);
Location bayonne = new Location("Bayonne", 43.492949, -1.4748409999999694);
Location pau = new Location("Pau", 43.2951, -0.3707970000000387);
```

```
Location sansebestian = new Location("San Sebastian", 43.318334,
    -1.9812312999999904);
Location pampelune = new Location("Pampelune", 42.812526,
    -1.645774500000016);
Location bilbao = new Location("Bilbao", 43.2630126, -2.9349852000000283);
```

Le code est disponible dans `/pub`, dans le fichier `villes.txt`.

Question 1.g Nous souhaitons maintenant associer des voisins aux villes. Si une ville B est voisine de A, c'est qu'il existe une route directe pour aller de A à B. Comme il existe des sens unique, ce n'est pas parce que B est voisine de A que A est forcément voisine de B.

De façon à modéliser les routes, commencez par ajouter à la classe **Location** un champ nommé **neighbors** référençant un tableau de **Location**. Ensuite, ajoutez une méthode d'instance

```
void setNeighbors(Location... neighbors)
```

à votre classe **Location**. Les trois petits points permettent de spécifier que le nombre de paramètres n'est pas à priori connu, mais que ces paramètres sont tous du type **Location**. Du côté de l'appelant, on peut alors spécifier un nombre quelconque de voisins, comme avec l'expression :

```
bourges.setNeighbors(limoges, tours, orleans);
```

Du côté de l'appelé, **neighbors** est simplement un tableau dont les éléments sont les arguments : avec notre exemple, **neighbors** est un tableau dont les éléments sont des références vers Limoges, Tours et Orléans. Vous pouvez donc directement assigner **neighbors** à **this.neighbors** dans **setNeighbors**.

Question 1.h Complétez votre méthode **main** avec le code suivant permettant de créer quelques routes :

```
evry.setNeighbors(paris);
paris.setNeighbors(evry, lemans, orleans);
lemans.setNeighbors(orleans, tours, angers);
orleans.setNeighbors(lemans, paris, bourges, tours);
angers.setNeighbors(lemans, tours, poitiers);
tours.setNeighbors(angers, lemans, orleans, bourges, poitiers);
bourges.setNeighbors(limoges, tours, orleans);
poitiers.setNeighbors(limoges, angouleme);
limoges.setNeighbors(agen, angouleme, poitiers);
angouleme.setNeighbors(poitiers, limoges, agen, bordeaux);
bordeaux.setNeighbors(angouleme, agen, bayonne);
agen.setNeighbors(toulouse, pau, bordeaux, angouleme, limoges);
toulouse.setNeighbors(agen, pau);
bayonne.setNeighbors(bordeaux, pau, sansebestian);
pau.setNeighbors(pampelune, bayonne, agen, toulouse);
sansebestian.setNeighbors(bayonne, pampelune, bilbao);
pampelune.setNeighbors(bilbao, sansebestian, pau);
bilbao.setNeighbors(sansebestian, pampelune);
```

Exercice 2 : La ville à distance minimale

À haut niveau, pour calculer le plus court chemin d'une ville A à une ville B, l'algorithme parcourt le graphe des villes en partant de A et propage la distance minimale pour atteindre chaque ville de voisin en voisin. À chaque étape de l'algorithme, il faut être capable de trouver la ville la plus proche de l'origine. Pour cela, nous concevons une structure de données annexe permettant de trouver la ville la plus proche de l'origine dans cet exercice.

Question 2.a Commencez par ajouter un champ privé distance de type **double** à la classe **Location** et une méthode **double getDistance()** permettant de consulter cette distance. À terme, ce champ nous servira à stocker les distances minimales pour atteindre les villes. Pour le moment, initialisez ce champ à une valeur aléatoire comprise entre 0 et 100 dans le constructeur de **Location** en utilisant **Math.random()** qui renvoie une valeur aléatoire comprise entre 0 et 1.

Dans la suite de l'exercice, la méthode **getDistance** vous permet de consulter le champ privé distance de l'extérieur de la classe. Attention, ne confondez pas **getDistance** (qui donne la distance cumulée pour atteindre une ville à partir de l'origine) et **distanceTo** (qui permet de calculer la distance à vol d'oiseau entre deux villes et que vous avez mise en œuvre dans le première exercice !).

Question 2.b Nous commençons maintenant à définir la classe **LocationSet** permettant de trouver la ville avec le champ distance minimal. Comme le nombre de villes stockées dans le **LocationSet** n'est pas connu à priori, nous allons utiliser un tableau extensible. Ajoutez une classe nommée **LocationSet** dans le package **ensiie.shortestpath** contenant deux champs privés :

- **locations** : un tableau de **Location**,
- **nbLocations** : un entier indiquant combien de **Location** sont stockées dans le tableau.

Dans le constructeur de **LocationSet**, initialisez **nbLocations** à 0 et **locations** à **null**.

Allouez une instance de **LocationSet** que vous stockerez dans une variable nommée **test** à la fin de la méthode**main**.

Question 2.c Ajoutez maintenant une méthode

```
void add(Location location)
```

à la classe **LocationSet** permettant d'ajouter un nouvel élément au tableau. Vous devez vérifier que votre tableau ne soit pas **null** ou alors qu'il ne soit pas déjà plein avant d'ajouter la nouvelle **location**. Dans le premier cas, vous créerez un nouveau tableau de taille 2 et et ajouterez l'élément à ce nouveau tableau. Dans le deuxième cas, vous créerez un nouveau tableau de taille 2 + nbLocations, recopierez le tableau référencé par **locations** dans le nouveau tableau, et ajouterez la nouvelle **location** à ce tableau.

Dans les deux cas, vous n'oublierez pas de stocker le nouveau tableau dans **locations** et de mettre à jour la valeur de **nbLocations**.

Dans la méthode **main**, ajoutez les villes de Paris, Evry et Orléans au **LocationSet** référencé par test.

Tableaux extensibles

Nous utilisons donc ici une implémentation de tableaux extensibles où la taille est augmentée de deux éléments quand le tableau est plein. On pourrait ici déclarer une classe propre à ces tableaux avec les fonctionnalités nécessaires (constructeur, ajout élément, suppression élément etc). Vous pouvez le faire pour vous entraîner.

Question 2.d Ajoutez enfin une méthode d'instance **Location removeMin()** à la classe **LocationSet**. Cette méthode doit trouver la ville avec le plus petit champ distance du **LocationSet**, retirer cette ville du **LocationSet**, et enfin renvoyer cette ville. Vérifiez que votre code est correct en ajoutant, à la fin du main, le code suivant qui retire les villes en suivant l'ordre croissant des distances :

```
for(Location cur=test.removeMin(); cur!=null; cur=test.removeMin()) {
  cur.display();
  System.out.println("  distance: " + cur.getDistance());
}
```

Aide

- `Location removeMin()` doit renvoyer null si `nbLocations` est égal à 0
- sinon, elle doit :
  - trouver l'indice min du nœud avec la plus petite distance parmi les nœuds enregistrés dans l'ensemble,
  - supprimer ce nœud (pour supprimer l'élément se trouvant dans la case min du tableau, il suffit de copier l'élément `locations[nbLocations-1]` dans la case `min` avant d'ôter 1 à `nbLocations`)
  - et enfin renvoyer ce nœud.

Question 2.e Avant de passer à l'exercice 3, nous allons commenter les tests que nous avons conçu dans cette exercice :

- commentez dans le constructeur de **Location** l'initialisation de distance à une valeur aléatoire (l'initialisation correcte de ce champ sera effectué dans l'exercice suivant),
- supprimez du **main** l'allocation de **test**, les ajouts de villes à **test** et enfin le code qui vérifie que **removeMin()** est correcte.

Exercice 3 : Principe de l'algorithme du plus court chemin

Cet exercice a pour but de vous présenter l'algorithme du plus court chemin et de préparer l'exercice suivant dans lequel vous allez coder cet algorithme.

Pour calculer le plus court chemin d'un nœud A à un nœud B, le principe est de calculer de proche en proche, en suivant les voisins, les distances minimales permettant d'atteindre chaque nœuds. Pour cela, on stocke la distance minimale pour aller de A à chaque nœud déjà rencontré dans le champ **distance** de **Location** (voir Exercice 2).

À haut niveau, l'algorithme choisit, à chaque étape, le nœud N ayant le plus petit champ distance. La Figure 1 illustre l'algorithme avec le calcul du plus court chemin pour aller de A à E dans le graphe A, B, C, D et E (les nombres sur les arcs sont les distances). La Table 1 présente le déroulé de l'algorithme sur ce graphe.

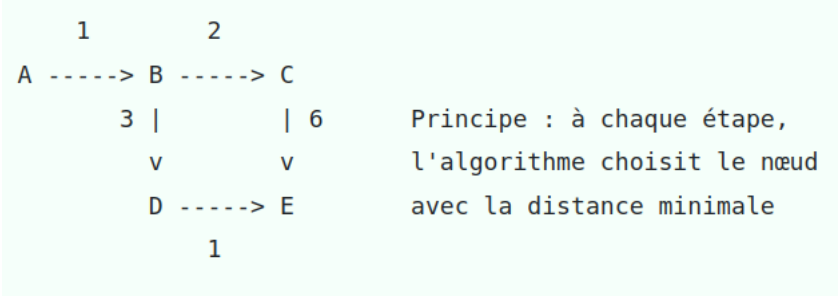

FIGURE 1 – Illustration du déroulé de l'algorithme de plus cours chemin pour aller de A à E.

	A	B	C	D	E	
Étape 0	0 (null)	∞ (?)	∞ (?)	∞ (?)	∞ (?)	Choisit A, marque A visité et propage la distance aux voisins (0+1 pour B qui vient de A)
Étape 1	- (null)	1 (A)	∞ (?)	∞ (?)	∞ (?)	Choisit B, marque B visité et propage la distance aux voisins (1+2 pour C et 1+3 pour D qui viennent de B)
Étape 2	- (null)	- (A)	3 (B)	4 (B)	∞ (?)	Choisit C, marque C visité et propage la distance aux voisins (3+6 pour E qui vient de C)
Étape 3	- (null)	- (A)	- (B)	4 (B)	9 (C)	Choisit D, marque D visité et propage la distance aux voisins ($4+1<9 \rightarrow$ met à jour E avec 5, et E vient maintenant de D)
Étape 4	- (null)	- (A)	- (B)	- (B)	5 (D)	Choisit E, destination atteinte, le chemin le plus court est de distance 5, et le chemin inverse le plus court est E → D → B → A

TABLE 1 – Déroulé de l'algorithme basé sur la Figure 1 pour calculer le plus court chemin de A à E.

L'état des nœuds dans le Tableau 1 est le suivant :

- ∞ = non atteint item nombre = distances minimale actuelle pour atteinre le nœud
- - : nœud visité
- (x) : champ 'from' qui indique le prédécesseur (x = null pour l'origine, x = ? si pas encore atteint)

En détail, à chaque étape de l'algorithme, on considère qu'un nœud peut être dans un des trois états suivants :

- **non atteint** : le nœud n'a pas encore été atteint par l'algorithme. Ce sont les nœuds à distance infini (par exemple C, D et E à l'étape 1).
- **atteint** : le nœud a été atteint et on le considère comme potentiel prochain départ pour l'étape suivante. Par exemple, à l'étape 2, C et D sont atteints.
- **visité** : le nœud est atteint et sa distance a été propagée à ses voisins à une des étape de l'algorithme. C'est, par exemple, le cas des nœuds A et B à l'étape 2 de l'algorithme.

À chaque étape, l'algorithme choisit le nœud atteint `N` ayant la plus petite distance cumulée. L'algorithme marque alors `N` comme visité de façon à éviter de le reprendre comme futur point de départ. Ensuite, pour chacun des voisins `V` de `N` :

- si `V` est toujours non atteint, l'algorithme le marque atteint,
- dans tous les cas, l'algorithme calcule la distance pour atteindre `V` en passant par `N`. Si cette nouvelle distance est plus petite qu'une distance précédemment trouvée (par exemple, c'est le cas à l'étape 3 de l'algorithme pour le voisin E de D), l'algorithme met à jour la distance du voisin. Dans ce cas, l'algorithme

mémorise aussi dans `V` que le plus court chemin vient de `N` dans un champ from que nous ajoutons à Location. Par exemple, à l'étape 3, comme atteindre E en venant de D est plus court qu'en venant de C, on note que la route la plus courte pour atteindre E provient de D et non de C (voir le résultat à étape 4).

Pour mettre en œuvre notre algorithme, il faut être capable de gérer les états des villes : atteint, non-atteint et visité. Pour cela, on utilise le champ distance de la classe Location et la classe annexe `LocationSet` que vous avez conçue à l'Exercice 2. Techniquement, on considère que :

- Si le champ distance d'un v vaut l'infini, c'est que le nœud est encore non atteint.
- Si son champ distance n'est pas égal à l'infini, c'est qu'il est soit atteint, soit visité. On considère qu'un nœud est atteint si il est dans le LocationSet et qu'il est visité lorsqu'il n'y est plus.

Question 3.a Pour calculer le plus court chemin d'une origine à une destination, nous avons déjà besoin d'ajouter un champ `from` à la classe `Location`. Ce champ indique le prédécesseur du nœud en suivant le chemin le plus court. Il n'est pas nécessaire d'initialiser ce champ dans le constructeur de `Location` car il sera initialisé lorsque son nœud sera marqué atteint. En revanche, on vous demande à cette question d'ajouter ce champ à Location.

Question 3.b Comme expliqué précédemment, les nœuds doivent être initialement à l'état non atteint, c'est-à-dire que leur champ distance doit être égal à l'infini. Modifiez le constructeur de `Location` de façon à initialiser le champ `distance` à l'infini (`Double.POSITIVE_INFINITY` en Java).

Question 3.c Ajouter enfin une méthode `void findPathTo(Location to)` ne faisant rien à la classe `Location`. Pour le moment, cette méthode ne fait rien, mais à terme, cette méthode va afficher le plus court chemin pour aller de `this` à `to`. Dans la méthode main, appelez `evry.findPathTo(bilbao)` qui va nous servir à tester notre code dans l'Exercice 4.

Exercice 4 : Mise en œuvre du plus court chemin

Nous allons coder l'algorithme de Dijskra la méthode `findPathTo` en quatre grandes étapes :

- Initialisation du calcul. Cette étape préliminaire doit, comme indiqué dans la figure au dessus, sélectionner le nœud d'origine après avoir initialisé son champ distance à 0.
- Mise en place de la boucle permettant de passer d'une étape à la suivante. Comme indiqué dans la figure au dessus, cette boucle doit sélectionner la ville atteinte ayant la distance la plus courte à l'origine.
- Réalisation d'une étape du calcul. Comme expliqué précédemment, une étape consiste en marquer le nœud sélectionné comme visité et en propager la distance à ses voisins.
- Affichage du chemin. Cet affichage doit partir de `to` et remonter le graphe en suivant les champs `from` jusqu'à l'origine.

Après un calcul de plus court chemin, l'affichage que devrait produire votre programme pour le plus court chemin entre Evry et Bilbao est le suivant :

```
Your trip from Evry to Bilbao in reverse order
  Bilbao at 839.900004625852
  San Sebastien at 762.3766098897902
  Bayonne at 717.0448287957582
  Bordeaux at 551.1372902800475
  Angouleme at 444.0668230106384
  Poitiers at 339.36769316298046
  Tours at 245.0644625176713
  Orleans at 137.18149761943013
  Paris at 26.087142174055035
```

Question 4.1 Afin d'initialiser l'algorithme, commencez par allouer un nouveau `LocationSet` dans `findPathTo` que vous stockerez dans une variable nommée `set`. Ensuite, positionnez le champ `distance` de this à 0. Enfin, définissez une variable de type `Location` nommée `cur`. Cette variable sert à référencer le nœud utilisé à l'étape courante. À l'initialisation (cf Figure 1), le nœud courant est le nœud d'origine.

Question 4.2 Afin de mettre en œuvre la boucle permettant de passer d'une étape à la suivante :

- l'algorithme se termine lorsque (i) cur vaut null, ce qui signifie qu'aucun chemin n'a été trouvé ou (ii) cur vaut `to`, ce qui signifie que l'algorithme a réussi à trouver le chemin minimum permettant d'atteindre `to`
- on passe d'une étape à la suivante en retirant le nœud ayant la distance minimum de `set` en appelant `removeMin()` et en le sélectionnant comme nœud courant. Comme le nœud est retiré de `set`, il passe à l'état visité sans avoir besoin d'opération supplémentaire.

À cette étape, écrivez la boucle permettant de passer d'une étape à la suivante. Vous écrirez le corps de cette boucle à la question suivante.

Question 4.3 Afin de réaliser une étape de l'algorithme, ajoutez la méthode d'instance `void proceedNode(LocationSet set)` à la classe `Location`. Ensuite, dans la boucle que vous venez de mettre en œuvre à la question précédente, ajoutez un appel à `cur.proceedNode(set)` afin de réaliser une étape de l'algorithme. Dans `proceedNode()`, vous devez propager la distance aux voisins. C'est ici que vous devez coder le contenu de la boucle de l'algorithme. Techniquement, vous devez donc parcourir les voisins de `this` et appliquer une étape de l'algorithme.

Question 4.4 Votre algorithme calcule maintenant le plus court chemin pour aller de `this` à `to` grâce à la méthode `Location.findPathTo`. On vous demande donc de l'afficher à la fin de cette méthode.

Après la boucle de calcul du plus court chemin, vous avez deux choix :

- si `cur` vaut null, c'est que l'algorithme n'a pas réussi à atteindre `to`. Affichez alors un message adéquat,
- sinon, c'est que l'algorithme a trouvé un chemin. Il vous suffit, dans une boucle, de remonter l'arbre des `from` en partant de `to` afin de remonter à l'origine (`this`). À chaque itération de la boucle, pensez à afficher la ville que vous avez trouvé.

Bonus

Vous remarquerez que l'on affiche le chemin en partant de `to`. Faites l'affichage dans le "bon" ordre, c'est à dire de `this` vers `to`.

Question 4.5 Testez votre algorithme pour aller de Evry à Bilbao, puis vérifiez qu'il n'existe pas de chemin partant de Bilbao et allant à Evry.

Exercice 5 : Parcours en profondeur

Notre algorithme ne permet d'effectuer qu'un unique calcul de plus court chemin. En effet, une fois que le champ distance a été affecté à une valeur pendant le calcul du plus court chemin, le nœud est définitivement considéré comme atteint ou visité, y compris lors des parcours suivant. Dans cet exercice, nous ajoutons donc une méthode `reinit()` permettant de remettre les nœuds à l'état non-atteint après un calcul du plus court chemin.

Cette méthode `reinit()` parcourt le graphe en profondeur, c'est-à-dire qu'elle explore le graphe des nœuds atteignables à partir de l'origine en commençant par aller le plus loin possible dans le graphe en suivant les voisins.

Question 5.1 Pour commencer, ajoutez une méthode d'instance vide `void reinit()` à la classe `Location` et appelez la à la fin de `findPathTo`.

Question 5.3 Codez la méthode `reinit()` de la classe `Location` permettant de réinitialiser les champs distance des villes.

Techniquement, `reinit()` doit réinitialiser le champs distance de `this` à `Double.POSITIVE_INFINITY` puis appeler `reinit()` sur les voisins de `this`. Les graphes que nous considérons dans cet exercice sont cycliques : de Paris, on peut aller au Mans, puis à Orléans avant de revenir à Paris.

Question 5.2 Dans la méthode main, après avoir calculé le plus court chemin pour aller d'Evry à Bilbao, calculez le plus court chemin pour aller d'Angers à Toulouse.