Modélisation d'un réseau routier sous la forme d'un graphe

Parmi les structures arborescentes les plus utilisées aujourd'hui dans les systèmes d'information, les graphes ont une place privilégiée: on les retrouve dans de nombreux champs d'application, comme la cartographie routière et la sociologie, où ils permettent la résolution de problèmes complexes.

A l'origine utilisés dans le but de déterminer les flux de transport optimums pour les marchandises, puis pour tout autre type d'information, les graphes sont désormais couramment mis en oeuvre dans de nombreux contextes. Ils permettent de modéliser des situations très concrètes de la vie courante, dans lesquelles des objets sont en interaction. Parmi les applications les plus classiques, on peut citer les interconnexions routières ou ferroviaires entre agglomérations, les réseaux de télécommunications, ou les relations interpersonnelles au sein d'une entreprise. Dans un cadre moins habituel, mais tout aussi important, on retrouve également les graphes dans le domaine de la génomique où ils sont utilisés en particulier pour l'élaboration et l'exploitation de cartes génétiques.
Cependant, ce sont surtout les algorithmes applicables aux graphes qui les rendent particulièrement intéressants: les objets et relations modélisés peuvent en effet facilement être manipulés par l'emploi de techniques et outils mathématiques, mis au point dans ce qui est historiquement appelé "la théorie des graphes". Ces algorithmes rendent possible la fourniture de réponses à des questions complexes telles que la détermination du plus court chemin entre deux villes, ou la possibilité de mettre une rue en sens unique sans paralyser la circulation en ville.
L'objet de cet article est de proposer une approche sommaire de la théorie des graphes, en en présentant tout d'abord les principaux points terminologiques, puis en la mettant en oeuvre au travers de la modélisation d'un réseau routier à l'aide du langage Java.
Ce texte n'a donc pas vocation à être exhaustif sur le plan théorique; aussi le lecteur est-il invité à se référer à l'un des nombreux ouvrages disponibles en la matière, afin d'obtenir s'il le souhaite une vision plus globale du sujet.

Eléments de terminologie relatifs aux graphes

Comme nous l'avons indiqué plus haut, les graphes permettent de symboliser une situation concrète dans laquelle des objets sont en relation. Dans ce type de représentation, objets et relations sont respectivement nommés "sommets" et "arcs".
Par définition, un graphe est orienté: les liens entre les sommets ne sont pas symétriques. Cependant, certaines situations ne tiennent pas compte de l'orientation et le graphe est alors qualifié de non orienté (les arcs sont dans ce cas appelés "arêtes"). A titre d'exemple, une carte routière nationale ou départementale peut être vue comme un graphe non orienté (une même route peut être empruntée dans les deux sens), au contraire d'un plan de ville qui doit nécessairement être considéré comme un graphe orienté pour tenir compte des sens uniques.
Schématiquement, les arcs sont représentés conventionnellement par des flèches dont les extrémités initiales et finales (sommets prédécesseur et successeur) sont clairement identifiées par un nom (A1,A2,..., An). De la même manière, les arcs ou arêtes peuvent également se voir attribuer une valeur correspondant à un type de données simple ou complexe, et on parlera dans ce cas de graphe valué.
Concernant le réseau routier qui sera modélisé dans la suite de cet article, les routes disposeront d'une valeur égale à la distance séparant les deux villes placées à leurs extrémités.

L'une des caractéristiques les plus utilisées lors de la manipulation des graphes est le nombre d'arcs reliés à un sommet particulier. Afin d'être plus précis, il est intéressant de connaître pour chaque sommet du graphe le nombre d'arcs "entrants" (demi-degré intérieur), le nombre d'arcs "sortants" (demi-degré extérieur), ainsi que la somme de ces deux valeurs: le degré du sommet en question. Sur la figure n°1, on constate que le sommet A4 est de degré 5; ses demi-degrés extérieur et intérieur valant respectivement 2 et 3.

Chemins et cycles

Dans un graphe, il semble intuitif de vouloir se déplacer d'un sommet à un autre en suivant les arcs. Une telle marche, appelée "chemin", est à l'origine d'un certain nombre de questions intéressantes qui peuvent alors se poser: pour deux sommets du graphe, existe t'il un chemin pour aller de l'un à l'autre? Quel est l'ensemble des sommets que l'on peut atteindre depuis un sommet donné? Comment trouver le plus court chemin pour aller d'un sommet à un autre?
Mathématiquement, un chemin c=(A1,..., An) est défini comme une liste de sommets dans laquelle deux sommets successifs sont toujours reliés par un arc (ou une arête). La longueur du chemin est, par convention, égale au nombre d'arcs parcourus.
Sur le graphe présenté en figure n°1, il existe plusieurs chemins pour aller du sommet A2 au sommet A5. Par exemple, on trouve c1=(A2,A3,A5) de longueur 2, c2=(A2,A4,A1,A3,A5) de longueur 4, ou c3=(A2,A4,A1,A3,A4,A1,A3,A5) de longueur 7. Notons que le groupe de sommets (A4,A1,A3) est un cycle (le premier et le dernier sommet sont identiques) qui peut naturellement être emprunté autant de fois que l'on veut... Ce qui signifie qu'il existe une infinité de chemin entre A2 et A5.

La connexité d'un graphe

Cette notion est directement liée à l'existence d'un chemin entre deux sommets. En effet, un graphe est dit connexe s'il existe un chemin (de longueur quelconque) entre chaque paire de sommets. Lorsque ce n'est pas le cas, le graphe est non connexe et il est considéré comme la juxtaposition de plusieurs sous-graphes appelés "composantes connexes". Le réseau routier français est un bon exemple d'illustration de cette notion de connexité. Celui-ci est à l'évidence composé de deux parties: l'une pour le continent, l'autre pour la Corse. Chacune de ces parties, prise individuellement, constitue un graphe connexe. Une fois réunies, elles forment un ensemble équivalent à un graphe non connexe dans la mesure où il n'existe aucune liaison routière entre le continent et l'île de Beauté.

Eléments nécessaires à l'implémentation d'un graphe:
fonctions, axiomes et outils mathématiques

Les aspects terminologiques qui viennent d'être énoncés doivent servir de base à tout développement informatique d'une structure de données de type "Graphe". De fait, il faut maintenant préciser d'une part les fonctions (ou méthodes) nécessairement attendues, d'autre part les axiomes qu'il convient de respecter. Nous verrons par ailleurs que l'emploi d'outils mathématiques s'avèrera indispensable à l'exploitation de modèles basés sur les graphes.

Sur le plan fonctionnel, on retrouvera généralement les opérations permettant de réaliser les actions suivantes:
- calcul des demi-degrés intérieur et extérieur, ainsi que celui du degré du graphe;
- ajout et suppression d'un sommet;
- ajout et suppression d'un arc entre deux sommets donnés;
- récupération d'une référence au ième successeur d'un sommet donné;
- test de l'existence d'un arc de sommet initial "x" et de sommet final "y";
- calcul du nombre total de sommets (c'est ce que l'on appelle l'ordre du graphe);

- calcul du nombre total d'arcs.

Cette liste d'opérations peut conduire à l'élaboration de l'interface présentée dans le listing n°1, qui pourrait par la suite servir de référence pour la conception de classes d'implémentation. Notez que la compilation de cette interface nécessite la déclaration préalable d'une classe "Sommet", dont le code Java peut (dans le cas le plus simple) se limiter à l'instruction:

```
public class Sommet {}
```

Cependant, en dehors de quelques cas d'école bien particuliers, l'interface proposée n'est généralement pas utilisée telle quelle:
- la signature des méthodes est souvent modifiée dans le but d'être plus explicite, compte tenu du champ d'application du futur logiciel. Qui, en effet, parmi les sociologues, s'amuserait à concevoir et maintenir un outil dont le code source fait mention d'arcs et de sommets, là où on pourrait plus naturellement parler de relations et d'individus? Dans le même ordre d'idée, nous verrons par la suite que pour modéliser un réseau routier, on choisira de donner aux méthodes des noms plus compréhensibles (comme "ajouterRoute" ou "nombreVilles", plutôt que "ajouterArc" ou "ordre").
- le nombre de méthodes pourra être sensiblement différent. Toujours dans l'objectif d'être au plus proche du contexte d'application, on pourra librement choisir d'ajouter ou de retirer des opérations pour "coller" le plus possible aux fonctionnalités attendues du produit à créer. Dans le cas précis du réseau routier nous concernant, il ne sera par exemple pas nécessaire d'implémenter des méthodes chargées de calculer les demi-degrés.

Intéressons-nous maintenant aux axiomes à respecter. Ceux qui viennent le plus aisément à l'esprit sont les suivants:
- Initialement, l'ordre d'un graphe possède la valeur zéro car il ne contient aucun sommet.
- l'ordre du graphe est incrémenté à chaque ajout de sommet, et à l'inverse, il est décrémenté à chaque suppression de sommet;
- le degré d'un sommet est égal à la somme des demi-degrés intérieur et extérieur;
- l'ajout d'un arc d'extrémité initiale "x" et d'extrémité finale "y" implique l'incrémentation du demi-degré extérieur de "x" et du demi-degré intérieur de "y";
- Au contraire, la suppression du même arc entraîne la décrémentation des demi-degrés évoqués.

Il existe quelques autres axiomes, que l'on passe ici volontairement sous silence afin de ne pas trop s'étendre sur le sujet. Ceux que l'on vient de mentionner sont les plus importants, et il est indispensable qu'ils soient implémentés au sein des méthodes évoquées plus haut.

Venons-en à présent aux principaux outils mathématiques qu'il est nécessaire d'employer pour faciliter l'exploitation des graphes. On en dénombre deux: la matrice d'adjacence et la liste d'adjacence. Dans un souci de précision, on parlera plus volontiers de listes d'adjacences (au pluriel), dans la mesure où pour un graphe donné, on manipulera autant de listes qu'il y a de sommets. Quel est le rôle de ces outils? Ils permettent de mémoriser la structure du graphe proprement dite, c'est-à-dire les relations qui lient les sommets entre eux. On peut affirmer sans équivoque qu'ils constituent, à eux seuls, une représentation complète du graphe considéré. Ils sont utilisés par la plupart des algorithmes, où ils facilitent le parcours de l'ensemble des sommets en vue d'un traitement particulier.
- La matrice d'adjacence comptabilise un nombre identique de lignes et de colonnes, égal à la valeur de l'ordre du graphe. En général, les éléments qui la

composent sont des valeurs binaires qui permettent de préciser l'existence d'une relation entre deux sommets (figure n°2). Mais il est parfois judicieux d'y placer d'autres types de valeurs, qui pourront éventuellement correspondre à des structures de données plus élaborées. Ainsi, nous constaterons par la suite que la matrice représentant le réseau routier contiendra des éléments complexes, qui sont en fait des objets instanciés à partir d'une classe modélisant une route située entre 2 villes.
Le net avantage de l'utilisation de la matrice réside dans la facilité de découverte de l'existence d'un arc entre 2 sommets. Par contre, cet outil est peu performant lorsqu'il s'agit d'évaluer d'autres critères, comme les demi-degrés.
- La représentation du graphe par listes d'adjacence consiste à associer à chaque sommet la liste ordonnée de ses successeurs (figure n°3). Pour un sommet donné, il est très simple de déterminer la valeur du demi-degré extérieur: elle est égale à la longueur de la liste d'adjacence correspondante. A l'inverse, le calcul du demi-degré intérieur est beaucoup plus long à établir, dans la mesure où son évaluation nécessite le parcours de toutes les listes d'adjacences du graphe.

Il est donc important de noter que ces deux outils ne se valent pas, en terme de performance, suivant le contexte. Outre les remarques qui viennent d'être faites, on peut par ailleurs indiquer que plus le graphe contient de sommets, plus l'utilisation de la matrice sera appropriée. Il convient donc de choisir l'un ou l'autre de ces outils compte tenu de la densité du graphe.

Modélisation d'un réseau routier
Maintenant que l'ensemble des éléments théoriques a été rappelé, nous pouvons nous intéresser à une démarche complète d'implémentation d'un graphe. Projetons pour cela de modéliser une partie du réseau routier français, représentée sur la figure n°4. Chaque ville y est symbolisée par un ovale de couleur verte disposant de 2 attributs: le nom et l'index de la ville. L'intérêt de l'utilisation d'un index sera précisé plus loin. Les routes sont quant à elles schématisées par de simples traits rouges, sur lesquels est inscrite la distance kilométrique séparant les 2 villes extrêmes.

- La première étape consiste à implémenter 2 classes représentant les sommets et les arcs du réseau (autrement dit, les villes et les routes):
La classe "Ville" (listing n°2) possède une propriété privée "nom" pouvant uniquement être mise à jour par le biais du constructeur. Cela impose la précision de ce nom lors de chaque instanciation d'un objet de type "Ville". La lecture de la valeur de cette propriété est rendue possible par l'emploi de l'accesseur "getNom()".
La classe "Route" (listing n°3) présente trois propriétés privées qui, pour les mêmes raisons que précédemment, ne peuvent être mises à jour que par l'intermédiaire du constructeur. Deux de ces propriétés sont les villes situées aux extrémités de la "Route". Les valeurs qu'elles peuvent contenir doivent obligatoirement être des objets de type "Ville". La troisième propriété est une valeur entière qui caractérise la distance séparant les 2 villes. Les valeurs de ces 3 propriétés peuvent être lues au moyen des accesseurs correspondants.

- La seconde étape est la plus importante, puisque c'est elle qui constituera la modélisation réelle d'un réseau routier sous la forme d'un graphe. Notons qu'il s'agit bien ici d'implémenter un réseau quelconque, constitué uniquement de villes et de routes, sans considération pour le moment du cas particulier du réseau français.
Cette modélisation peut se faire sous la forme d'une classe "ReseauRoutier" (listing n°4).
Celle-ci doit obligatoirement gérer la liste des villes, ainsi que celle des routes. Pour cela, 2 propriétés privées de type "Vector" sont déclarées. Pour mémoriser la structure du réseau, il est possible d'utiliser une matrice d'adjacence que l'on déclare aussi en tant que propriété privée. Les éléments de

cette matrice sont des objets de type "Route". On dispose donc au total de 3 propriétés, dont l'initialisation est réalisée au sein du constructeur.
Parmi les méthodes de la classe, on note en particulier:
- "ajouterVille(...)", chargée de créer un nouvel objet de type "Ville" et de l'ajouter à la liste des villes. La ville ainsi instanciée se voit automatiquement attribuer un index correspondant à sa place dans le vecteur;
- "ajouterRoute(...)", dont le rôle est successivement de créer un nouvel objet de type "Route" en prenant en compte les arguments passés, d'ajouter cet objet à la liste des routes du réseau, et de mettre à jour la matrice d'adjacence;
- "getVille(...)", renvoyant la référence de la ville positionnée à l'index "index" de la liste des villes;
- "getIndexVille(...)", permettant de faire l'opération inverse, à savoir récupérer l'index d'une instance de la classe "Ville";
- "getRouteDirecte(...)", renvoyant la référence d'une route située entre 2 villes, grâce à l'emploi de la matrice d'adjacence;
- "nombreVilles(...)" et "nombreRoutes(...)", retournant respectivement le nombre total de villes et de routes déclarées dans le réseau routier.
En prêtant attention au contenu de ces méthodes, on constate que les axiomes énoncés plus haut (en dehors de ceux relatifs aux demi-degrés, car non traités dans ce cas simple) sont toujours respectés.

- La dernière étape de l'implémentation va permettre de spécialiser le réseau routier quelconque que l'on vient de modéliser, afin de concevoir un réseau particulier: celui de la France. On va donc développer une classe "ReseauFrance", qui dérivera nécessairement de "ReseauRoutier" afin de pouvoir exploiter l'ensemble des méthodes que l'on vient de passer en revue (listing n°5).

La création des villes intervient naturellement avant celle des routes. On utilise à cet effet la méthode "ajouterVille(...)" en lui indiquant le nom de la ville en argument. Il est impératif d'instancier les villes dans l'ordre précisé par les index: en effet, ce sont ces derniers, et non pas les noms de ville, qui serviront toujours par la suite de référence pour identifier sans ambiguïté une ville dans le réseau. Ce type de fonctionnement garantit la possibilité de déclarer plusieurs villes dont les noms sont identiques.
On s'attache ensuite à l'instanciation des routes. Celle-ci est rendue possible par l'emploi de la méthode "ajouterRoute(...)", qui impose la précision des 2 villes extrêmes et de la distance kilométrique les séparant. Comme on vient de le voir, la référence des villes est obtenue par l'intermédiaire de leur index: par exemple, pour récupérer celle de la ville de Lille, on exploite la méthode "getVille(...)" en lui passant en argument la valeur "1". Ainsi, pour créer la route Lille-Lyon, on utilisera l'instruction:

```
ajouterRoute(getVille(1),getVille(8),670);
```

Une exploitation simple du réseau
Le réseau routier français est maintenant complètement modélisé. Il est intéressant de le tester afin d'être sûr de sa validité. Pour cela, l'ajout de quelques lignes de code supplémentaires destinées à afficher à l'écran la distance kilométrique séparant chaque couple de ville peut suffire.
Au sein de la classe "ReseauFrance", on déclare une méthode "afficherDistance(...)" prenant en argument les références de 2 villes. Les instructions qu'elle contient vérifient l'existence d'une route entre ces agglomérations avant d'en afficher la distance. Si la route en question n'existe pas, un message adapté est présenté.
L'appel de cette dernière méthode pour tout couple de villes du réseau est réalisé au moyen de 2 boucles imbriquées, que l'on place dans la méthode "main(...)" de la classe "ReseauFrance".

Il ne reste plus qu'à exécuter le programme. Pour cela, il suffit d'ouvrir la console Unix (ou Windows) dans le répertoire où se situent les classes Java, de compiler ces dernières au moyen de la commande "javac *.java", et de lancer l'exécution grâce à l'ordre "java ReseauFrance".

Pour aller plus loin

La figure n°5 montre une illustration du résultat obtenu par cette exploitation sommaire du réseau routier. On peut y constater que si le programme fournit des résultats corrects lorsqu'il existe une route directe entre 2 villes, il n'est pas en mesure de délivrer une information fiable si une ou plusieurs villes intermédiaires se trouvent le long du chemin.
Pourtant il existe bel et bien un itinéraire possible entre Lille et Bordeaux, ou entre Marseille et Nantes. Plus justement, on peut même affirmer intuitivement que de nombreux itinéraires sont empruntables entre chaque couple de villes. Comment donc les déterminer? Et parmi ceux-ci, comment sélectionner celui dont la distance sera la plus faible? Des algorithmes applicables aux structures de données de type "Graphe" permettent de donner des réponses à de telles questions. Nous verrons cela dans le cadre d'un prochain article.

François-Xavier SENNESAL

(((légendes)))
ReseauRoutier1.tga: Figure n°1: Un exemple de graphe orienté.
ReseauRoutier2.tga: Figure n°2: Matrice d'adjacence du graphe de la figure n°1.
ReseauRoutier3.tga: Figure n°3: Les listes d'adjacence du graphe de la figure 1.
ReseauRoutier4.tga: Figure n°4: Le réseau routier à implémenter.
ReseauRoutier5.tga: Figure n°5: Affichage des distances séparant les villes.
ReseauRoutier6.tga: Le mathématicien Euler, célèbre pour ces travaux dans la théorie des graphes.
ReseauRoutier7.tga: Un exemple de cycle eulérien. Quel itinéraire choisir pour tracer cette figure sans lever le crayon?
ReseauRoutier8.tga: Les ponts de Königsberg. Peut-on traverser entièrement la ville en ne parcourant chacun d'eux qu'une seule fois?
ReseauRoutier9.tga: La modélisation du problème des ponts de Königsberg.
ReseauRoutier10.tga: Comment savoir si l'on peut modifier le sens de circulation dans cette rue sans paralyser le trafic?
ReseauRoutier11.tga: Toute la complexité des réseaux routiers peut être modélisée sous la forme de graphes.
ReseauRoutier12.tga: Les graphes au service de la sociologie par la cartographie des relations interpersonnelles.
ReseauRoutier13.tga: Sciences de la vie ou chimie... les champs d'application de la théorie des graphes sont nombreux.
ReseauRoutier14.tga: Les réseaux de télécommunication ne sont pas non plus en reste!
ReseauRoutier15.tga: Les aiguillages d'un réseau ferroviaire peuvent être considérés comme les sommets d'un graphe.
ReseauRoutier16.tga: Les systèmes de fichiers: un exemple classique de structure arborescente.
ReseauRoutier17.tga: Les algorithmes utilisés par Maporama permettent de déterminer l'itinéraire le plus court entre deux lieux.
ReseauRoutier18.tga: Quels itinéraires peut-on emprunter pour aller d'une station à une autre?

Listing 1

```
public interface Graphe {
      public int demiDegreExt(Sommet s);
      public int demiDegreInt(Sommet s);
      public int degre(Sommet s);
      public boolean ajouterSommet(Sommet s);
```

```
	public boolean retirerSommet(Sommet s);
	public boolean ajouterArc(Sommet s1,Sommet s2);
	public boolean retirerArc(Sommet s1,Sommet s2);
	public Sommet iemeSuccesseur(Sommet s);
	public boolean arc(Sommet s1, Sommet s2);
	public int ordre();
	public int nombreArcs();
}
```

Listing 2

```
public class Ville
{
	private String nom;

	public Ville(String n)
		{
		nom=n;
		}

	public String getNom()
		{
		return nom;
		}
}
```

Listing 3

```
public class Route
{
	private int distance;
	private Ville ville1;
	private Ville ville2;

	public Route(Ville v1, Ville v2,int d)
		{
		ville1=v1;
		ville2=v2;
		distance=d;
		}

	public int getDistance()
		{
		return distance;
		}

	public Ville getVille1()
		{
		return ville1;
		}

	public Ville getVille2()
		{
		return ville2;
		}
}
```

Listing 4

```
import java.util.*;

public class ReseauRoutier
```

```
{
	private Vector ListeVilles;
	private Vector ListeRoutes;
	private Route [][] matriceRoutes ;

	public ReseauRoutier()
		{
		//Initialisation des listes
		ListeVilles=new Vector(0,1);
		ListeRoutes=new Vector(0,1);

		//Déclaration d'une matrice d'adjacence pour
		//un réseau de 10 sommets au maximum.
		matriceRoutes=new Route[10][10];
		}

	public int nombreVilles()
		{
		//renvoie le nombre de villes du réseau.
		return ListeVilles.size();
		}

	public int nombreRoutes()
		{
		//renvoie le nombre de routes du réseau.
		return ListeRoutes.size();
		}

	public int getIndexVille(Ville v)
		{
		//Retourne l'index de la ville v.
		//ou -1 en cas d'erreur.

		int retour=-1;
		boolean trouve=false;
		int k=0;
		Ville v1;

		while (!trouve)
			{
			v1=(Ville)ListeVilles.elementAt(k);
			if (v1==v)
				{
				trouve=true;
				retour=k;
				}
			k++;
			if (k>=ListeVilles.size()) trouve=true;
			}

		return retour;
		}

	public Ville getVille(int index)
		{
		//Retourne la "Ville" associée à l'index.
		return (Ville)ListeVilles.elementAt(index);
		}

	public Route ajouterRoute(Ville ville1, Ville ville2, int distance)
		{
```

```
            //Création d'une route entre "ville1" et "ville2"
            Route r=new Route(ville1,ville2,distance);

            //Ajout de la route créée à l'ensemble des routes du réseau.
            ListeRoutes.add(r);

            //Déclaration de la route dans la matrice d'adjacence.
            matriceRoutes[getIndexVille(ville1)][getIndexVille(ville2)]=r;
            //Le réseau routier étant non orienté, on référence à
            //nouveau la route dans la matrice, mais en changeant
            //l'ordre des indices.
            matriceRoutes[getIndexVille(ville2)][getIndexVille(ville1)]=r;

            return r;
            }


      public Ville ajouterVille(String nom_ville)
            {
            //Création d'une "Ville"
            Ville v=new Ville(nom_ville);

            //Ajout de la "Ville" à l'ensemble des villes.
            ListeVilles.add(v);

            return v;
            }

      public Route getRouteDirecte(Ville v1, Ville v2)
            {
            //Renvoie l'identifiant de la route située entre
            //les villes v1 et v2. Si aucune route n'existe,
            //la valeur NULL est renvoyée.
            Route r;

            r=matriceRoutes[getIndexVille(v1)][getIndexVille(v2)];

            return r;
            }
}
```

Listing 5

```
import java.util.*;

public class ReseauFrance extends ReseauRoutier
{
      public ReseauFrance()
            {
            super();

            CreerReseau();
            }

      private void CreerReseau()
            {
            //Création des villes dans l'ordre
            //précisé par les index.
            ajouterVille("Paris");
            ajouterVille("Lille");
            ajouterVille("Rennes");
```

```
            ajouterVille("Nantes");
            ajouterVille("Poitiers");
            ajouterVille("Bordeaux");
            ajouterVille("Montpellier");
            ajouterVille("Marseille");
            ajouterVille("Lyon");

            //Création des routes.
            ajouterRoute(getVille(0),getVille(1),250);
            ajouterRoute(getVille(0),getVille(2),350);
            ajouterRoute(getVille(0),getVille(3),400);
            ajouterRoute(getVille(0),getVille(4),350);
            ajouterRoute(getVille(4), getVille(5),240);
            ajouterRoute(getVille(5),getVille(6),475);
            ajouterRoute(getVille(6),getVille(7),150);
            ajouterRoute(getVille(7),getVille(8),280);
            ajouterRoute(getVille(0),getVille(8),425);
            ajouterRoute(getVille(3),getVille(4),280);
            ajouterRoute(getVille(1),getVille(8),670);
            ajouterRoute(getVille(8),getVille(6),300);
            }


      public void afficherDistance(Ville v1, Ville v2)
            {
            //Récupération de l'identifiant de la route directe située
            //entre v1 et v2.
            Route r=getRouteDirecte(v1,v2);
            if (r!=null)
                  {
                  System.out.println("Distance "+v1.getNom()+" ->
"+v2.getNom()+": "+r.getDistance()+"km.");
                  }
            else
                  {
                  System.out.println("Pas de route directe entre
"+v1.getNom()+" et "+v2.getNom()+".");
                  }
            }



      public static void main(String args[])
            {
            //Instanciation du réseau français
            ReseauFrance rf=new ReseauFrance();

            for (int i=0;i<=(rf.nombreVilles()-1);i++)
                  {
                  for (int k=(i+1);k<=(rf.nombreVilles()-1);k++)
                        {
                        //Affichage de la distance séparant chaque couple de
villes.
                        rf.afficherDistance(rf.getVille(i),rf.getVille(k));
                        }
                  }
            }
}
```