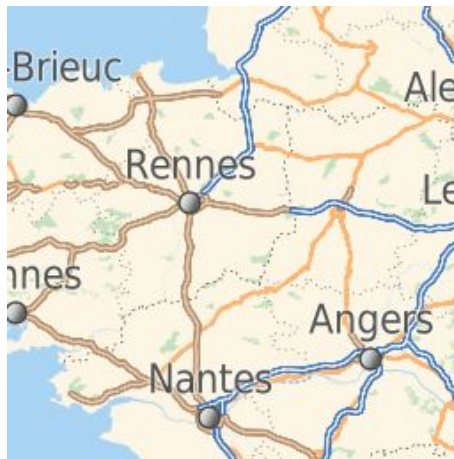


---

# Programmation par Objet

---



*Auteurs :*  
AMINE AIT ALI  
SIMON DESCAMPS

*Tuteur :*  
SANTIAGO BARGAGNOLO

## Table des matières

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Analyse des packages et des classes</b>	<b>2</b>
2.1	Package Solveur . . . . .	3
2.1.1	Voie . . . . .	3
2.1.2	Autoroutes et Routes . . . . .	3
2.1.3	Villes . . . . .	3
2.1.4	Carte . . . . .	4
2.2	Package Algorithme . . . . .	4
2.2.1	Dijkstra . . . . .	4
2.3	Package Application . . . . .	4
2.3.1	Main . . . . .	4
<b>3</b>	<b>Structure de données utilisée</b>	<b>5</b>
<b>4</b>	<b>Pseudo-Code JAVA</b>	<b>7</b>
<b>5</b>	<b>Premiers tests</b>	<b>9</b>
<b>6</b>	<b>Le parseur SAX</b>	<b>10</b>
<b>7</b>	<b>Gestion des exception</b>	<b>10</b>
<b>8</b>	<b>Fonctionnement du projet</b>	<b>11</b>
8.1	Contenu de l'archive . . . . .	11
8.2	Chargement du projet . . . . .	11
8.2.1	Sous Eclipse . . . . .	11
8.3	L'interface graphique . . . . .	12
8.4	Fonctionnement de l'interface graphique . . . . .	13
8.5	Exemple d'utilisation . . . . .	15
<b>9</b>	<b>Difficultés rencontrées et pistes d'amélioration</b>	<b>17</b>
<b>10</b>	<b>Conclusion</b>	<b>18</b>

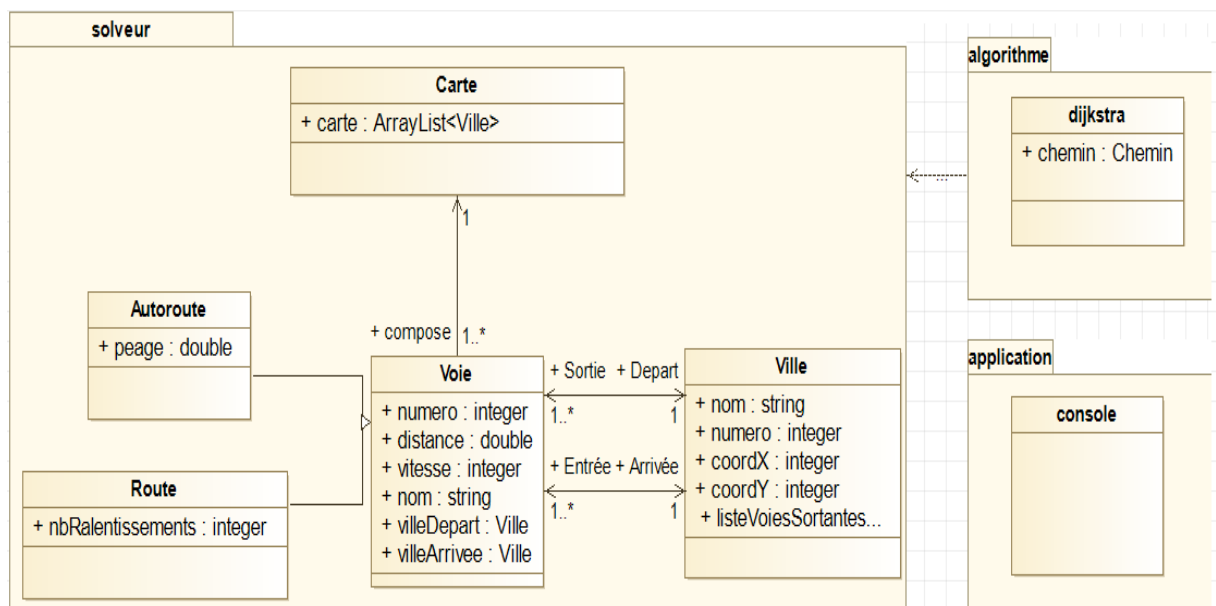
# 1 Introduction

Dans le cadre de notre formation d'ingénieur statistique et informatique à Polytech Lille, nous sommes amenés à réaliser un projet portant sur les graphes et combinatoire ainsi que la programmation orientée objet. A ces connaissances seront associées celles de structures de données. Les graphes permettent de modéliser un problème et aider à le résoudre. Les structures de données quant à elles, permettent la manipulation des graphes. Dans ce projet, on souhaite trouver le meilleur chemin entre 2 lieux sur une carte routière à la façon d'outils tels que Michelin, Google Map ou Mappy par exemple. La carte routière est représentée par un graphe orienté. Pour cela il faut trouver l'itinéraire optimal en fonction de différents critères qui peuvent être la rapidité, le coût ou la distance. Pour résoudre ce problème nous utiliserons l'algorithme de Dijkstra que nous implémenterons en JAVA.

## 2 Analyse des packages et des classes

La carte est représentée par un graphe dont les sommets sont les lieux (villes) et les arcs orientés sont les voies d'accès entre les villes. Pour des raisons de simplifications, nous ne retiendrons pour les lieux que les noms des villes et pour les voies que les routes et les autoroutes.

Nous aurons besoin de l'ensemble des classes et des packages suivants :



Le package **solveur** comprendra l'ensemble des classes associées au graphe et aux méthodes associées.

Le package **algorithme** comportera principalement l'implémentation de l'algorithme de Dijkstra adaptée à notre structure de données.

Le package **applications** comprendra une version complète de l'application nommée console utilisable dans un terminal. Ce package ne sera pas étudié dans cette partie.

## 2.1 Package Solveur

### 2.1.1 Voie

Une **voie** est définie par un numéro de voie, ce qui est sa clé primaire. En effet, chaque voie correspond à une portion de route ou d'autoroute entre 2 villes. Chaque voie a ses critères : sa distance kilométrique, la vitesse réglementaire à respecter sur cette voie et son nom. On peut calculer le temps nécessaire pour parcourir la voie, on peut également calculer le coût en euros à payer en empruntant cette voie. Ce coût est calculé en fonction de la consommation en carburant du véhicule, qui dépend lui-même de la vitesse, il utilise aussi la distance kilométrique de la voie.

Cette classe sera composée des méthodes suivantes :

Méthode	
getDistance()	Renvoie la distance kilométrique de la voie.
getVitesse()	Renvoie la vitesse réglementaire sur la voie
getTemps()	Renvoie le temps nécessaire pour parcourir la voie (calcul).
getNom()	Renvoie le nom de la voie.
getCout()	Renvoie le coût associé à la traversée de la voie.
getVilleArrivee()	Renvoie la ville d'arrivée.
getVilleDepart()	Renvoie la ville de départ.
getPoids()	Appelle la méthode adéquate de calcul.

### 2.1.2 Autoroutes et Routes

Une voie peut être une route ou une autoroute. Il est important de différencier ces deux types de voies puisque des spécificités sont apportées par cette information. Pour ce qui est de l'autoroute, on peut rencontrer un ou plusieurs péages. Les classes autoroute et route sont des sous-classe de voie. Elles possèdent donc les mêmes méthodes que la classe voie. Il est cependant nécessaire de modifier certaines fonctions dans les sous classes pour prendre en compte les péages ou les ralentissements par exemple. Pour ce faire, il faut re-définir la méthode du calcul de temps de trajet dans la sous-classe route, et re-définir la méthode du calcul du coût de trajet dans la sous-classe autoroute.

Nous devons également écrire les méthodes suivantes qui étaient définies de façon abstraite dans la classe voie :

Méthode	
getPeage()	Renvoie le coût associé au péage ( Autoroute ).
getNbRalentissements()	Renvoie le nombre de ralentissements ( Route ).

### 2.1.3 Villes

Chaque **ville** possède ses caractéristiques. Une ville a un numéro de ville, qui représente sa clé primaire, elle possède également un nom. Chaque ville sera représentée sur la carte grâce à des coordonnées. Pour cette partie du projet, nous ne traiterons pas cette question. Cette classe sera composée des méthodes suivantes :

Méthode	
getNom()	Renvoie le nom de la ville.
getCoordX()	Renvoie la coordonnée X de la ville.
getCoordY()	Renvoie la coordonnée Y de la ville.

#### 2.1.4 Carte

Dans cette classe, il s'agira de stocker la structure de données que nous allons utiliser, c'est-à-dire l'ensemble des voies et l'ensemble des villes qui vont constituer notre zone d'étude. Dans notre cas précis, ce sera la carte du Nord-Pas-De-Calais. Les méthodes servant à l'affichage seront aussi implémentées dans cette classe :

Méthode	
addVille(Ville)	Permet d'ajouter une ville.
addVoie(Voie)	Permet d'ajouter une voie.
getVilleFromNom(NomVille)	Permet de trouver une ville à partir de son nom.

## 2.2 Package Algorithmme

Nous avons choisi de créer un package algorithme dans une idée d'évolution du projet. Si un jour on souhaite prendre le projet en implémentant un autre algorithme du plus court chemin, il suffira simplement de créer une nouvelle classe dans ce package.

#### 2.2.1 Dijkstra

Cette classe est capitale dans le projet puisqu'elle permet l'appel de la méthode Dijkstra. Tout le projet tourne autour de l'efficacité de cet algorithme. Elle prend en paramètre les villes d'arrivées et de départ qui sont de type "ville". Elle prend également un paramètre "method" où l'on précise s'il on veut utiliser le trajet le moins coûteux d'un point de vue argent, temps ou distance.

Méthode	
PCC(dep, arr, method)	Appelle l'algorithme de Dijkstra en fonction de la méthode choisie (coût, temps, distance)

## 2.3 Package Application

#### 2.3.1 Main

Cet fichier nous permet d'exécuter et ainsi tester les différentes méthodes créées lors du projet. Il est aussi chargé d'afficher les résultats.

Une fois que toutes ces classes sont créées, nous allons évoquer la structure de données utilisée pour satisfaire le projet.

Le projet correspond donc à l'arborescence suivante :



**Remarque :** Les fichiers *.class* sont obtenus par compilation de nos fichiers.

### 3 Structure de données utilisée

Afin de représenter notre graphe, nous avons d'abord pensé à utiliser une Tree-Map de ville et de voies, mais l'utilisation de ce type de structure de donnée aurait causé de la redondance. En effet, la représentation Modelio nous montre que l'on connaît déjà les successeurs (Voies) d'une ville. Il n'est donc pas nécessaire de les stocker sous forme de TreeMap. Notre map est donc composée d'une liste de Villes. Nous utiliserons une ArrayList car l'accès par indice est plus facile qu'avant les LinkedList.

Comme dit précédemment, d'une ville, on peut obtenir ses prédécesseurs. Cette représentation nous permet de gagner beaucoup d'espace mémoire. De plus, on

connaît également tous les prédécesseurs d'une ville. Par conséquent, il sera plus simple d'implémenter l'algorithme de Dijkstra tel que nous l'avons écrit.

Dans la plupart des cas, nous avons défini les variables de nos classes en *private*. De cette manière, seules les classes dans lesquelles sont définies ces variables peuvent y accéder. L'utilisation de *getters* nous permet de récupérer leur valeur mais pas de la modifier.

**Rappel :** *Les getters sont les méthodes permettant de renvoyer la valeur d'une variable privée.*

## 4 Pseudo-Code JAVA

Dans cette partie, nous allons présenter notre version de l'algorithme de Dijkstra. En premier lieu, nous avons utilisé des HashMaps afin de créer Mark, Père et Potentiel. L'idée, à la différence de tableaux est que la taille de ces derniers n'est pas fixe et que l'on ajoutera des éléments à ces Maps que si on visite un des sommets. Cela nous permettra une nouvelle fois de gagner en mémoire. De plus, l'utilisation des Maps permet un accès aux éléments par clé ; dans notre cas on utilisera toujours des villes.

Mark, Père, Potentiel ainsi que le chemin de type Chemin sont définis dans la classe Dijkstra comme suit :

- **private** Chemin chemin = **new** Chemin()
- **private** Map<Ville, Boolean> Mark = **new** HashMap<Ville, Boolean>()
- **private** Map<Ville, Voie> Pere = **new** HashMap<Ville, Voie>()
- **private** Map<Ville, Double> Potentiel = **new** HashMap<Ville, Double>()

La map **Mark** contient un booléen permettant de savoir si la ville en clé a été marquée.

La map **Pere** contient la voie précédant la clé. Elle est utile pour stocker le chemin avant de le construire. En effet, pour construire ce dernier on part de la ville d'arrivée et on remonte une à une les villes grâce aux voies qui les précèdent.

La Map **Potentiel** contient le potentiel associé à chaque sommet. Ce dernier est calculé en fonction de la méthode passée en paramètre.

L'algorithme de Dijkstra permettant de trouver le plus court chemin entre deux villes. Dans notre implémentation, le plus court chemin est défini en fonction de la méthode (le moins cher, le plus rapide ou le plus court).

Voici le code associé :



```

1  fonction Dijkstra(depart, arrivee, methode) :
2      depart, arrivee : Villes
3      Current : Element de carte
4      curr Voie
5      //Initialisation
6      Mark.put(depart, true)
7      Potentiel.put(depart, true)
8      Current = depart
9      //Execution
10     //On propage les potentiels
11     Tant que ( Current != arrivee ) :
12         listeVoies = Current.getVoiesSortantes()
13         Pour chaque Voie succ de listeVoies :
14             Si ( Mark.get(succ.getVilleArrivee()) == null ) :
15                 Si ( Potentiel.get(succ.getVilleArrivee()) == null
16                     OU Potentiel.get(succ.getVilleArrivee())
17                     > Potentiel.get(succ.getVilleDepart()) + succ.getPoids(methode)) Alors :
18                     Potentiel.put(succ.getVilleArrivee(),(Potentiel.get(succ.getVilleDepart())
19                         + succ.getPoids(methode)))
20                     Pere.put(succ.getVilleArrivee(), succ)
21                 Fin Si
22             Fin Si
23         Fin Pour
24
25
26     //On recherche la ville non marquée de potentiel minimum
27     double min = -1
28     Boolean premier = true
29     Current = null
30     Pour chaque Ville v de Potentiel.keySet() :
31         Si ( Mark.get(v) == null ) Alors :
32             Si ( premier OU Potentiel.get(v) <= min ) Alors :
33                 min = Potentiel.get(v)
34                 Current = v
35                 premier = false
36             Fin Si
37         Fin Si
38     Fin Pour
39
40     //S'il n'existe pas de chemin
41     Si ( Current == null ) Alors :
42         renvoyer null
43     Sinon :
44         Mark.put(Current, true)
45     Fin Tant que
46     //Construction du chemin
47     curr = Pere.get(arr)
48     Tant que ( curr.getVilleDepart() != dep ) :
49         chemin.add(curr)
50         curr = Pere.get(curr.getVilleDepart())
51     Fin Tant que
52     chemin.add(curr)
53
54     renvoyer chemin
55 Fin Fonction

```

## 5 Premiers tests

Avant de se lancer dans l'exécution du projet, et donc dans l'utilisation des données fournies, il nous a été conseillé, dans le sujet du projet, de concevoir un léger exemple pour vérifier notre code, sans nécessité d'avoir une interaction avec l'utilisateur. Nous avons décidé de suivre cette recommandation et nous sommes lancés dans la création en "dur" d'une petite carte regroupant 5 villes dans le fichier test : Lille, Paris, Caen, Le Mans et Rennes.

Il n'est pas nécessaire de réaliser un beau menu ou de permettre la communication avec l'utilisateur. Nous avons testé quelques cas de figure :

Trajet	Résultat
Lille - Rennes	Bonne méthode
Rennes - Paris	Bonne méthode
Paris - Caen	Bonne méthode

En effet, nous avons opté pour différents critères et avons testé quel était le bon chemin. Par exemple, pour le trajet entre Lille et Rennes, il était intéressant de savoir quel chemin allait être sélectionné, celui passant par Paris, ou celui longeant la côte, en fonction du temps, du coût et de la distance. Peu importe, le critère, cela nous a toujours choisi le trajet adéquate. Une fois ces différents tests exécutés, nous sommes passés au contenu de l'archive, le travail sur les données à l'aide du parseur XML.

## 6 Le parseur SAX

Nous avons utilisé un parseur SAX (de l'acronyme Simple API for XML). Le parseur va parcourir le document XML qu'on lui passe en paramètre et pour chaque élément syntaxique qu'il va rencontrer, il va invoquer une méthode correspondant au type de l'élément syntaxique. Ce dernier repose principalement sur le ContentHandler qui définit l'ensemble des méthodes qui seront appliquées.

Dans notre cas, nous avons différencié tous les types de syntaxe que peut rencontrer le parseur dans notre document et nous avons codé une méthode associée à chaque type. À chaque fois que notre handler rencontre une fin de syntaxe telles que Route ou Autoroute par exemple, ce dernier exécute le constructeur associé. Par exemple lorsque le handler reconnaît une fin de route, alors il appelle le constructeur de Route.

**Remarque :** *Nous avons supposé que l'on ne passait que des fichiers valides dans le parser. Ce dernier n'est donc pas programmé pour reconnaître les erreurs, ni les modifier.*

## 7 Gestion des exception

Dans notre projet, nous avons défini que nous pouvions rencontrer 3 types d'exception :

- **MethodeInconnueException** : Lorsque l'on appelle la méthode PCC du package Dijkstra, il se peut que la méthode passée en paramètre ne corresponde à aucune méthode existante. Puisque le choix de la méthode sur l'interface graphique ce fait avec des boutons il n'est possible de rencontrer cette exception que si l'utilisateur ne choisit aucune méthode.
- **VilleInconnueException** : On peut tomber sur cette exception dans le cas de l'appel aux méthodes `getVilleFromCoord()` et `getVilleFromNom()`. Cela se produit si les coordonnées ne correspondent à aucune ville de notre map ou si le nom passé en paramètre de notre seconde méthode ne correspond à aucune ville.
- **CheminInconnuException** : On peut tomber sur cette exception par l'appel de la méthode PCC dans l'unique cas où il n'existe aucun chemin entre les villes de départ et d'arrivée.

Toutes les exceptions que nous avons créées sont des extensions de la classe `Exception`. Pour assurer le bon fonctionnement du programme, il est nécessaire de "throw" ces exceptions dès qu'on les rencontre. L'idée ensuite est de "catch" ces dernières dans la méthode `Main` de notre interface graphique et d'afficher à message à la place.

## 8 Fonctionnement du projet

### 8.1 Contenu de l'archive

Le projet est rendu sous forme d'archive. Dans cette archive, on trouve les éléments suivants :

- Le fichier map.xml
- Le fichier map.jpg
- L'archive du projet ppo-sdescamp
- Un dossier comportant le projet ppo-sdescamp décompressé

### 8.2 Chargement du projet

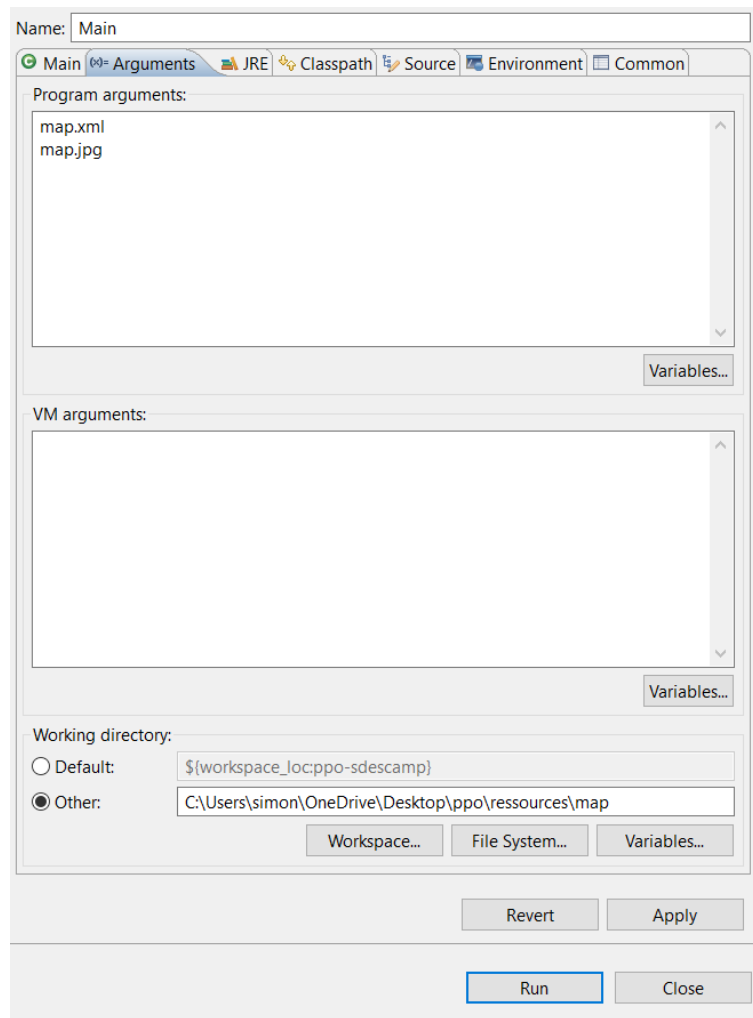
#### 8.2.1 Sous Eclipse

Pour pouvoir exécuter le projet, il faut importer ce dernier sous éclipse néon et exécuter l'application avec des arguments précis. Pour cela, cliquez sur *File > Import > Archive File > Browse* et sélectionnez l'archive *ppo-sdescamp*.

Une fois le projet importé, double cliquez sur *ppo-sdescamp > src > application*.

Faites ensuite clic droit sur le fichier *Main.java > Run As > Run Configurations...*

Cliquez ensuite sur l'onglet Arguments, et ajouter map.xml et map.jpg comme suit :



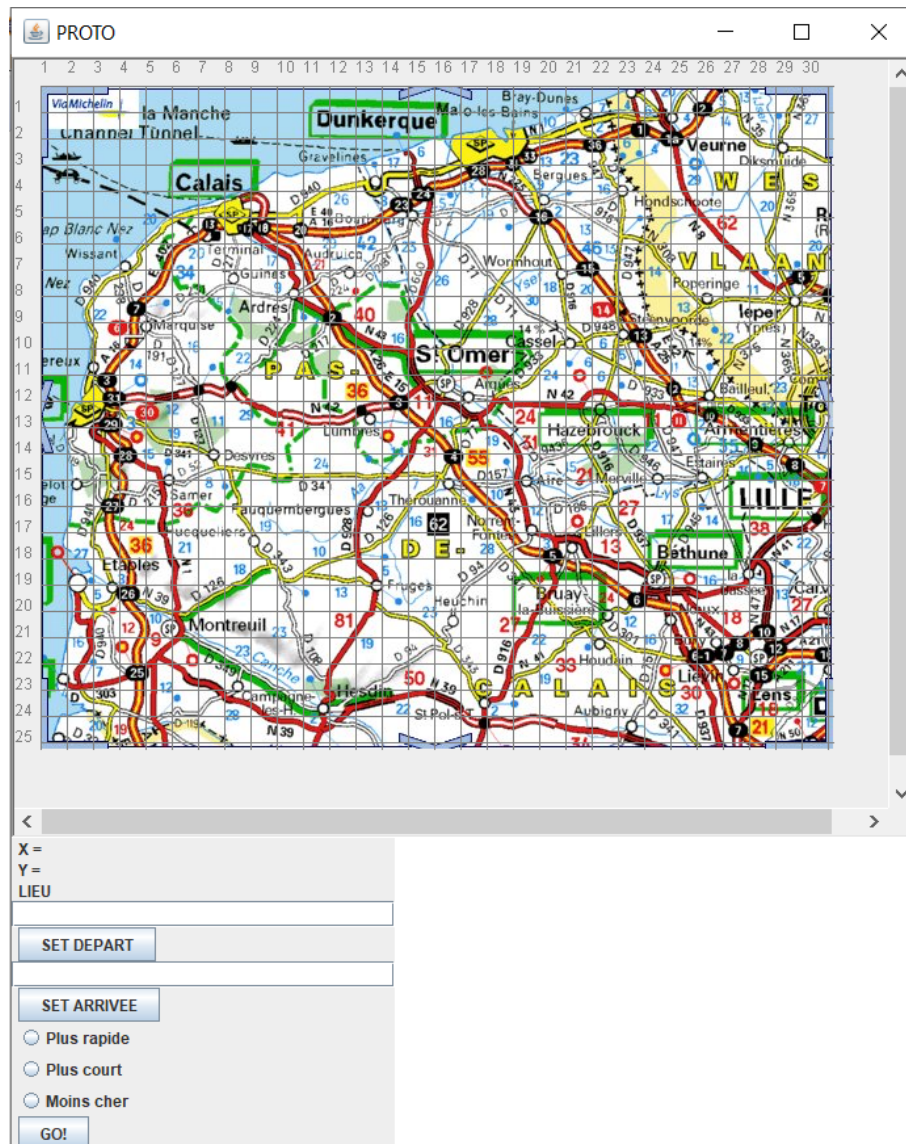
**Remarques :** Si les fichiers *map.xml* et *map.jpg* ne sont pas dans votre répertoire de travail, cliquez comme sur l'image ci-dessus sur *other* et placez vous à la racine du dossier dans lequel ils se situent.

Il est impératif d'entrer les deux arguments dans le même ordre que sur l'image car le programme attend d'abord le fichier *xml* puis l'image.

Une fois toutes les étapes précédentes faites, cliquez sur *Run*.

### 8.3 L'interface graphique

Si vous avez respecté toutes les étapes de la section précédente, la fenêtre suivante devrait s'ouvrir :



Il s'agit de l'interface graphique de notre programme. Son fonctionnement sera expliqué dans la sous-section suivante.

## 8.4 Fonctionnement de l'interface graphique

Le fonctionnement est plutôt intuitif. En premier lieu, il y a deux champs : X et Y. Ces derniers correspondent aux coordonnées des cases qui s'affichent lorsque vous cliquez sur la carte. Le champ du dessous (LIEU) affiche la ville correspondant aux coordonnées X et Y. Si vous voulez sélectionner cette ville comme départ ou d'arrivée, il suffit de cliquer sur **SET DEPART** ou **SET ARRIVEE**. Il est également possible d'entrer les villes de départ à la main, il faut pour cela écrire le nom de la ville dans le champ au dessus des boutons **SET DEPART** et **SET ARRIVEE**. Si la ville n'existe pas, un message s'affichera dans la fenêtre de texte située à droite, dans le cas contraire, un message de confirmation s'affichera.

**Remarque :** Le programme est conçu tel que le contenu du champ ait la prio-

*rité sur les coordonnées. Autrement dit, si vous voulez choisir une ville de départ ou d'arrivée grâce aux coordonnées, il est impératif que le champ associé soit vide.*

Pour terminer, il est impératif de cliquer sur une des méthodes (plus rapide, plus court ou moins cher) avant de cliquer sur **GO**.

Si vous cliquez sur **GO** et que le programme rencontre une erreur, celle-ci sera affichée dans le champ de texte pour vous en informer.

Le fonctionnement de l'interface graphique est très simple, nous avons créé tous les éléments qui sont affichés et nous avons ajouté des event listeners à nos boutons. Autrement dit, à chaque fois qu'un bouton est activé, un certain nombre d'actions que nous avons programmées s'exécutent. De plus, afin de récupérer les coordonnées X et Y de la carte, on a ajouté une action à chaque fois que l'on lâche la clic de la souris.




## 8.5 Exemple d'utilisation

Nous allons rechercher le plus court chemin entre Lille et Calais selon deux méthodes :

On s'intéresse en premier lieu au chemin **le plus court** :

PROTO



X = 8  
Y = 5  
Lieu : Calais  
Lille  
SET DEPART  
Calais  
SET ARRIVEE  
☐ Plus rapide  
☒ Plus court  
☐ Moins cher  
GO!

Vous avez choisi la ville de départ : Lille  
Vous avez choisi comme ville d'arrivée : Calais  
Vous avez choisi comme methode : Le plus court  
Veuillez suivre l'itinéraire suivant :  
Meilleur chemin entre Lille et Calais :  
- A25 de Lille à Bailleul  
- N42 de Bailleul à Hazebrouck  
- N42 de Hazebrouck à Saint-Omer  
- N43 de Saint-Omer à Calais  
Durée du trajet : 2h 11m 29s  
Distance du trajet : 112.0km  
Cout du trajet : 10.08

On constate alors que le plus court chemin entre Lille et Calais se fait en 4 étapes et est long de 112.0km. Ce trajet dure 2 heures 11 minutes et 29 secondes.



Si on s'intéresse maintenant au chemin le plus rapide :

PROTO

Vous avez choisi la ville de départ : Lille  
 Vous avez choisi comme ville d'arrivée : Calais  
 Vous avez choisi comme méthode : Le plus rapide  
 Veuillez suivre l'itinéraire suivant :  
 Meilleur chemin entre Lille et Calais :  
 - N41 de Lille à Bethune  
 - A26 de Bethune à Calais  
 Durée du trajet : 1h 12m 15s  
 Distance du trajet : 124.0km  
 Cout du trajet : 17.41

☒ Plus rapide  
☐ Plus court  
☐ Moins cher

On constate avec cette méthode que le plus court chemin se fait en 2 étapes. Ce dernier dure 1 heures 12 minutes et 15 secondes. Il dure donc bien moins longtemps que le trajet le plus court.

## 9 Difficultés rencontrées et pistes d'amélioration

Dans l'ensemble le projet n'était pas très compliqué puisqu'il s'agissait de manière générale d'appliquer ce que nous avons vu en cours et en TP tout le long du semestre. Cependant, il fallait être très vigilant de manière à choisir la bonne structure de données dès le départ. Pour notre part, nous avons changé de structure de données avant de commencer à programmer nos méthodes, cela ne nous a donc pas trop porté préjudice. De plus, il fallait prévoir toutes les erreurs en programmant nos méthodes, notamment Dijkstra. Dans notre cas, nous n'avions pas prévu le cas où il n'existe pas de chemin entre deux villes par exemple. Cela menait donc à une boucle infinie dans notre méthode du plus court chemin.

Outre cette erreur, nous avons trouvé que le temps destiné au projet était suffisant. Bien évidemment nous aurions pu faire de nombreuses améliorations avec plus de temps mais nous sommes parvenus à faire tout ce que nous voulions.

Parmi les améliorations possibles, nous pensons qu'il aurait été intéressant d'ajouter la possibilité d'ajouter un détour sur les itinéraires. Cette amélioration n'est pas si compliquée car il suffit simplement de calculer le plus court chemin entre la ville de départ et la ville intermédiaire puis le plus court chemin entre la ville intermédiaire et la ville d'arrivée.

Ensuite, une amélioration beaucoup plus compliquée à mettre en place serait de prendre en compte le trafic en temps réel. Pour cela il aurait fallu déployer de plus gros moyens avec des serveurs qui récupèrent les informations trafic, etc.

Nous avons également pensé qu'il était possible de rajouter la possibilité de changer le prix de l'essence sur l'interface graphique. Cette option est très simple à mettre en place, il suffit de modifier la valeur de la variable statique "prixEssence".

Finalement, il aurait été intéressant de permettre aux utilisateurs d'entrer leur véhicule afin d'adapter le calcul du coût en fonction de leur consommation. Cette amélioration est possible à condition de disposer d'une base de données donnant la consommation de chaque véhicule.

Pour terminer, nous voulions permettre la compilation et l'exécution du projet à la main, mais nous n'avons pas réussi à réaliser cela. En premier lieu, nous utilisons la commande `javac -encoding ISO-8859-15 src/application/*.java src/solveur/*.java src/algorithmes/*.java src/application/parseur/*.java src/ihm/*.java`. Cela fonctionne mais nous n'avons jamais réussi à exécuter le programme grâce à la commande `java` car nous obtenons systématiquement l'erreur suivante : *Erreur : impossible de trouver ou charger la classe principale Main*. Nous avons supposé que le problème venait du fichier `.classpath` généré par `modelio`.

## 10 Conclusion

En conclusion, nous avons apprécié passer du temps sur ce projet car il était très instructif. Il nous a permis de bien comprendre l'intérêt de la programmation par objet et de l'utilisation des packages. De plus, nous avons trouvé intéressant de découvrir les interfaces graphiques en JAVA. Cela nous a aidé à prendre conscience de ce que l'on peut faire en JAVA. Également, nous avons apprécié le fait d'utiliser Eclipse Néon pour ce projet. Pour notre part, nous nous en étions déjà servis en TP mais cela nous a permis d'approfondir notre maîtrise de cet environnement. Nous espérons à l'avenir pouvoir se servir de ce projet dans le cadre d'autres projets, il serait notamment intéressant de réutiliser la base de l'interface graphique pour en construire une autre.