**Rapport de Be Graphe**

# **Introduction**

Dans le cadre du cours de Graphes et de Programmation Objet, nous avons pu mettre en pratique nos connaissances acquises dans un bureau d’étude en java.

Nous avions comme objectif de comprendre et d’implémenter deux algorithmes de parcours de graphes ainsi que de réfléchir à un problème ouvert utilisant ces algorithmes de parcours. Les algorithmes de parcours sont l’algorithme de Dijkstra et l’algorithme A\*. Le problème ouvert que nous avons choisi est celui du covoiturage.

# **Implémentation**

Ce bureau d’étude se veut le plus proche possible d’un projet que l’on pourrait avoir à développer en entreprise. Afin de pouvoir coder sur le même projet en même temps, nous avons utilisé un logiciel de gestion de version, git.

La structure du programme étant déjà donnée, il ne nous restait qu’à implémenter les algorithmes et les quelques classes dont ils avaient besoin.

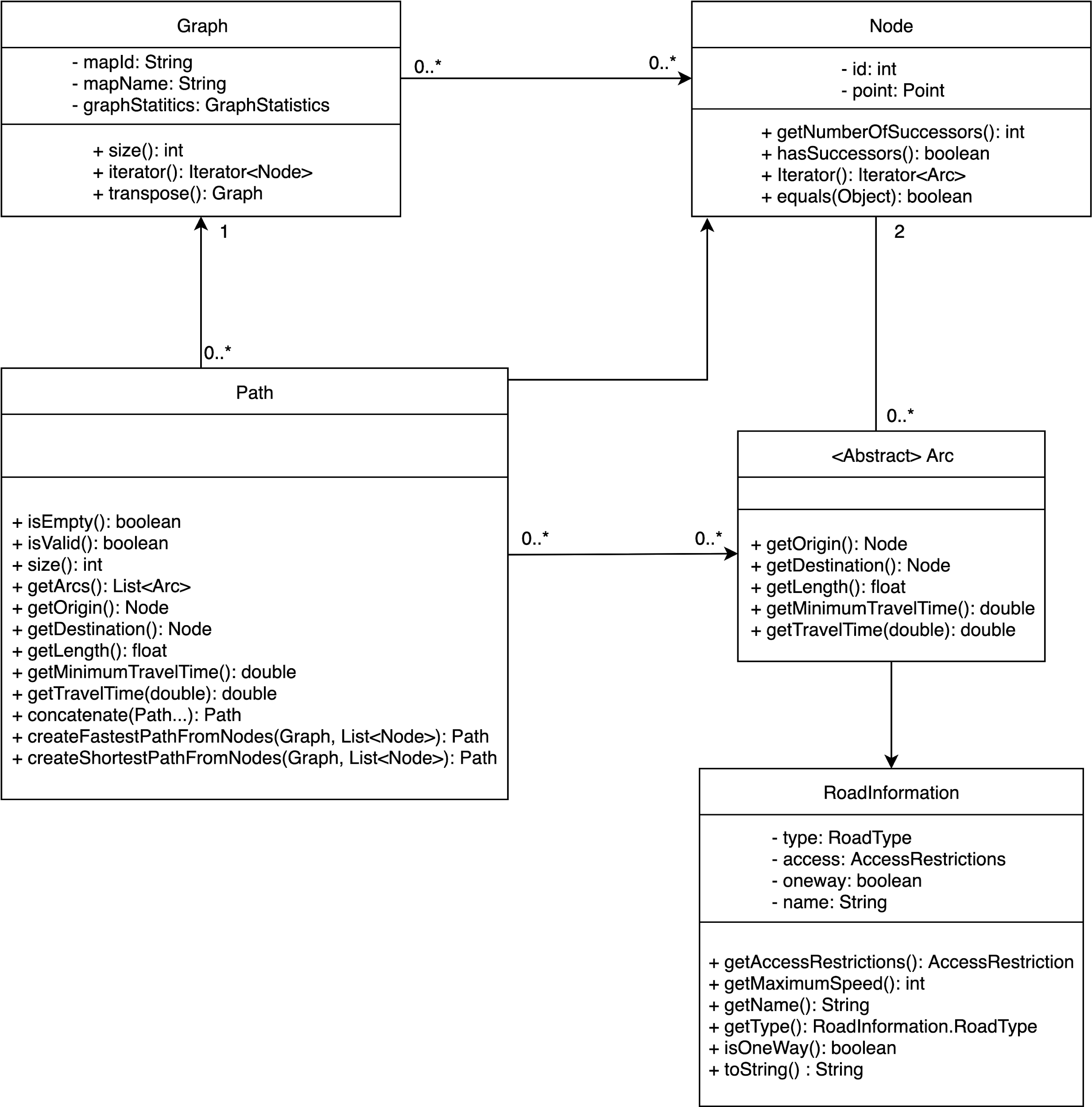
Nous avons implémenté la fonction “Remove” d’un tas pour ensuite pouvoir développer l’algorithme de Dijkstra.

Nous avons pu découvrir le concept de design pattern Observer qui permet de notifier des objets d’autres classes sans devoir multi-threader notre programme.

Lors des tests sur la carte « carré dense », nous pouvons voir le comportement de l’algorithme de Dijkstra qui se répand dans toutes les directions. A contrario, l’algorithme A\* suit une ligne directrice dans la direction de la destination.

1. **Documents de conception: Diagramme UML**

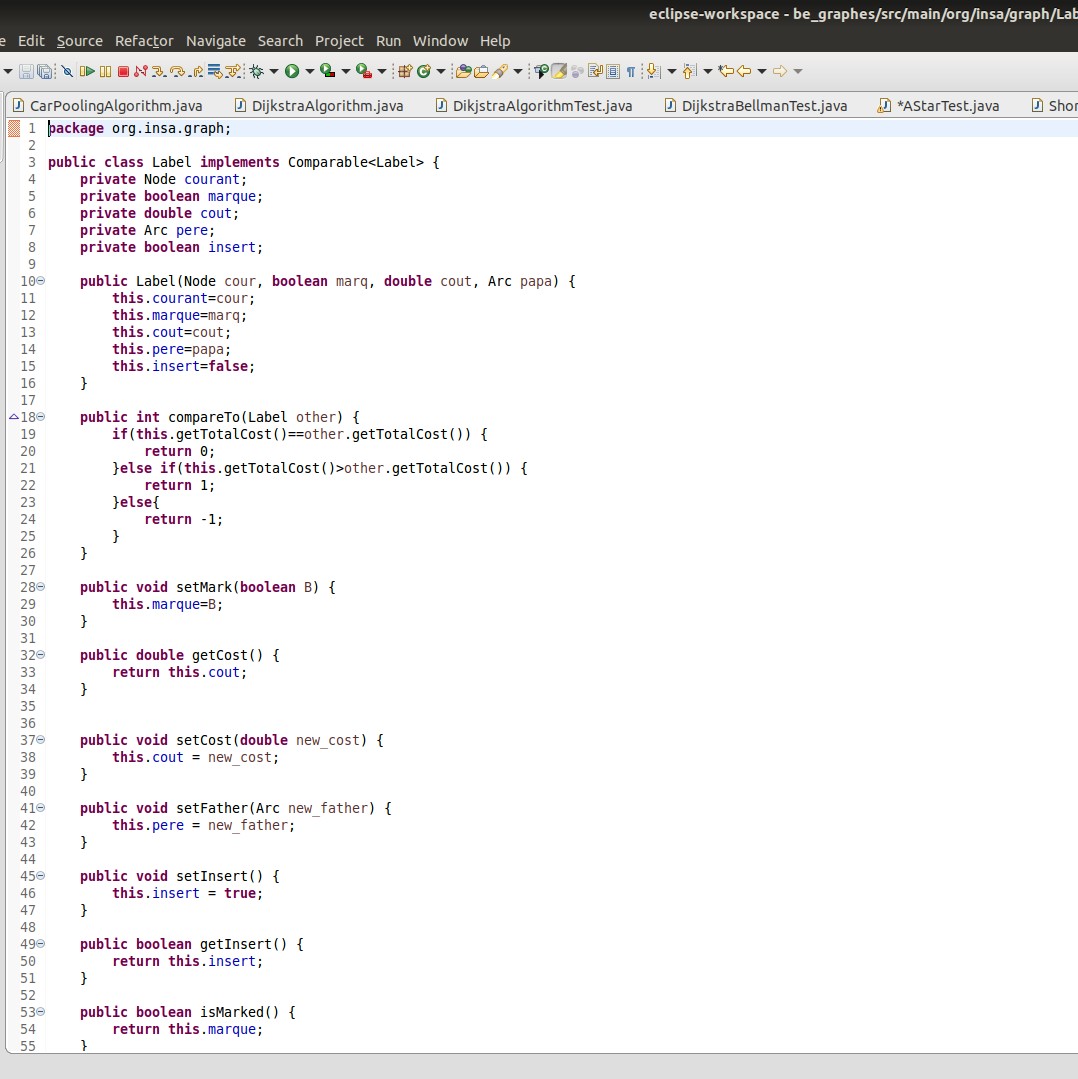
Afin de bien comprendre la documentation créée par Javadoc et le code qui nous a été fourni, nous avons dû concevoir un diagramme UML synthétique des classes Graph, Node, Arcs, Path et RoadInformation:

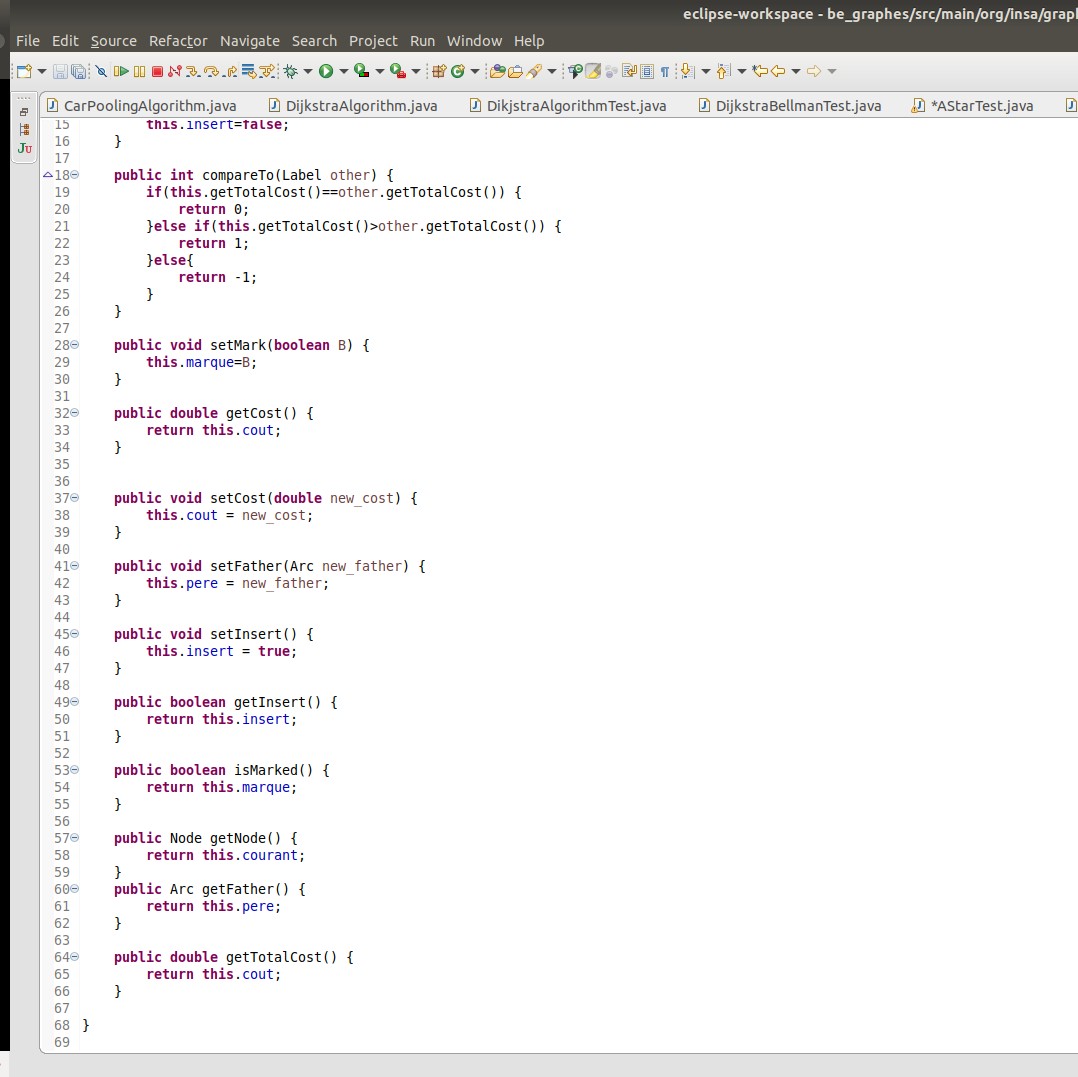
[](https://www.draw.io/?page-id=wdP3J9sE5_09kGqsjboZ&scale=auto#G1mdVTaEo9xJElqdw32MVCgE7sbYxQKJem)

Une fois ce diagramme fait, nous avons pu commencer à implémenter les différentes méthodes demandées en sachant à chaque fois où aller chercher les méthodes des différentes classes.

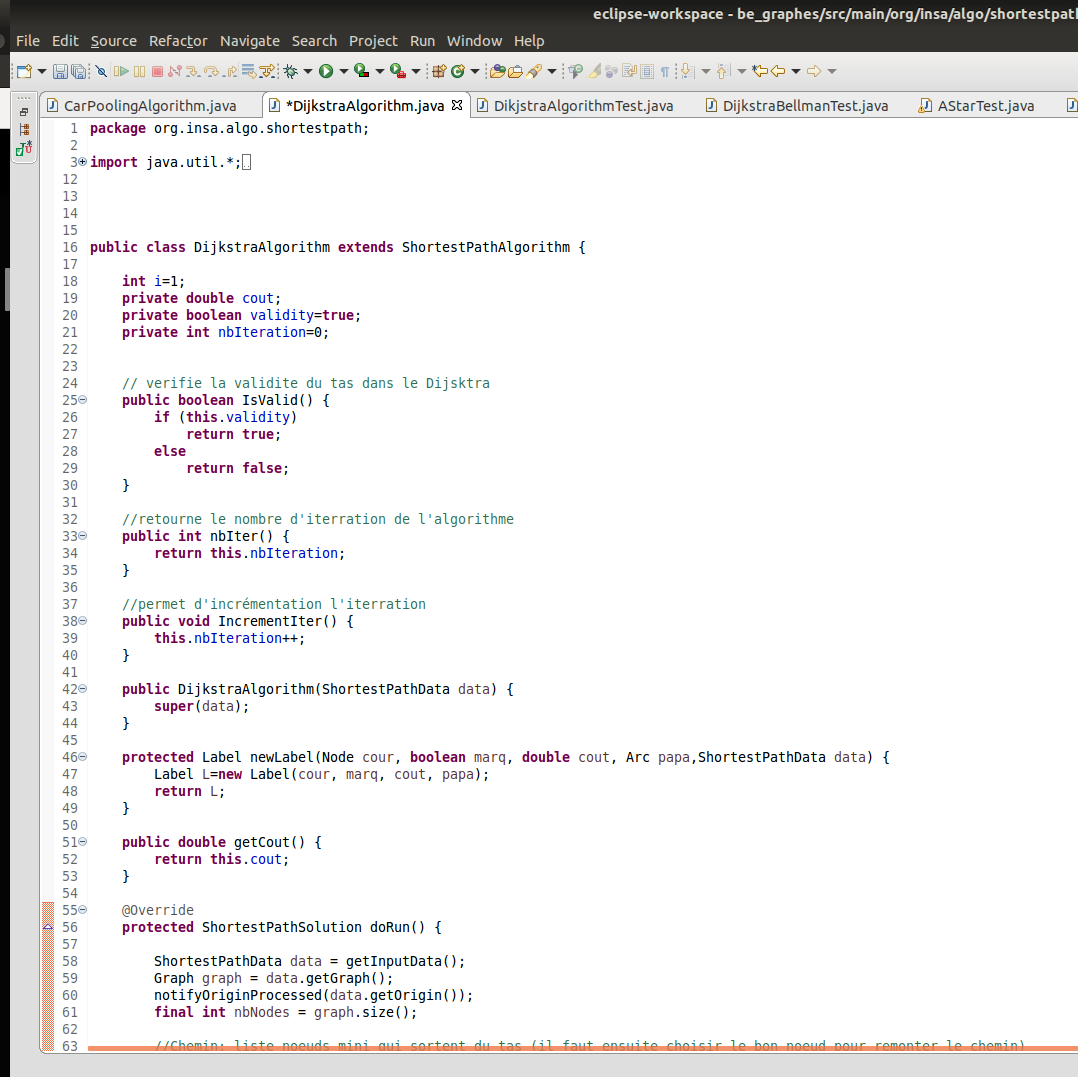
Suite à cela, nous avons eu à coder toutes les méthodes marquées “deprecated” de la classe Path ainsi que la méthode Remove de la classe BinaryHeap. Ces méthodes sont essentielles pour pouvoir implémenter l’algorithme de Dijkstra.

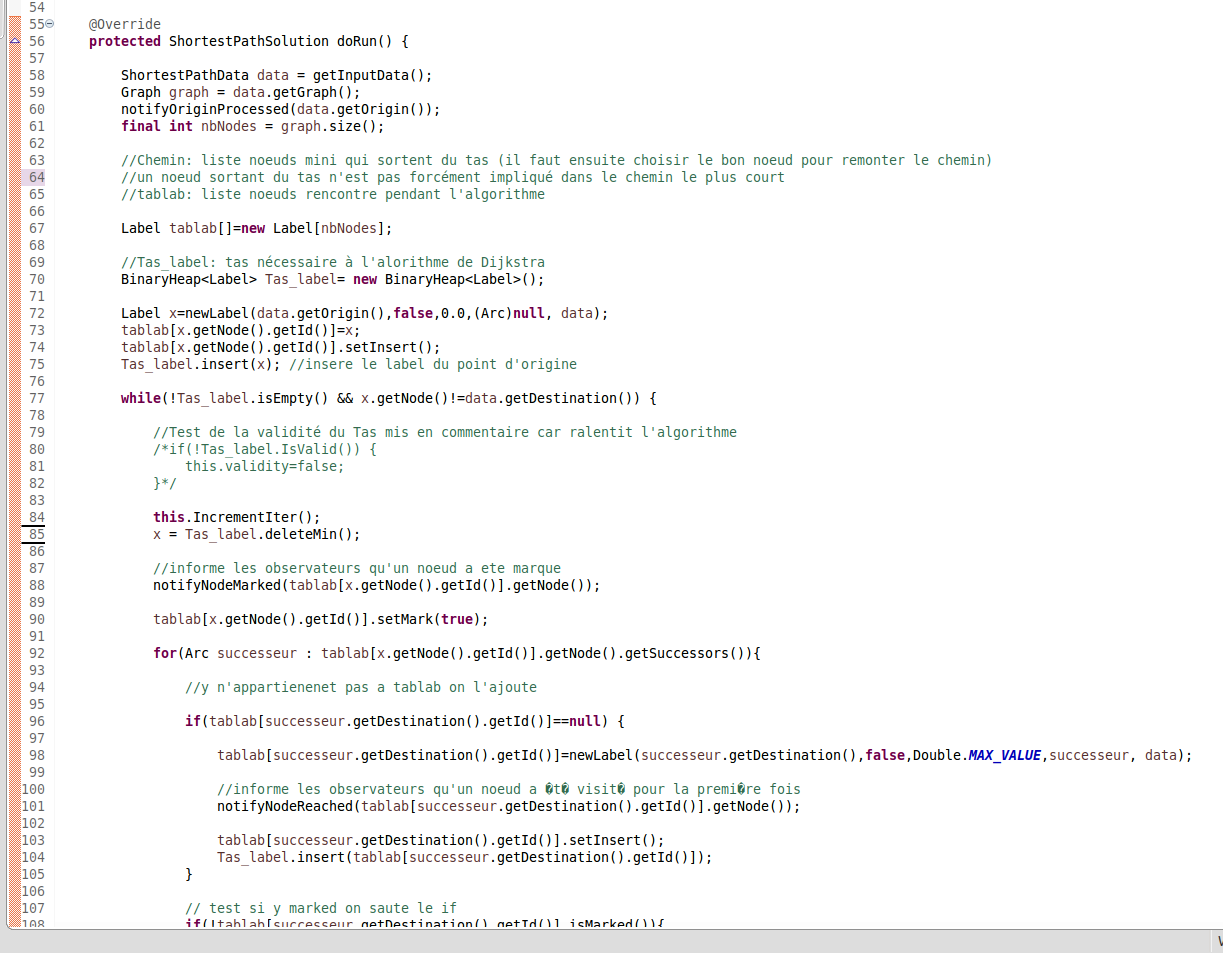
1. **Algorithme de Dijkstra:**
2. **Code de Label**

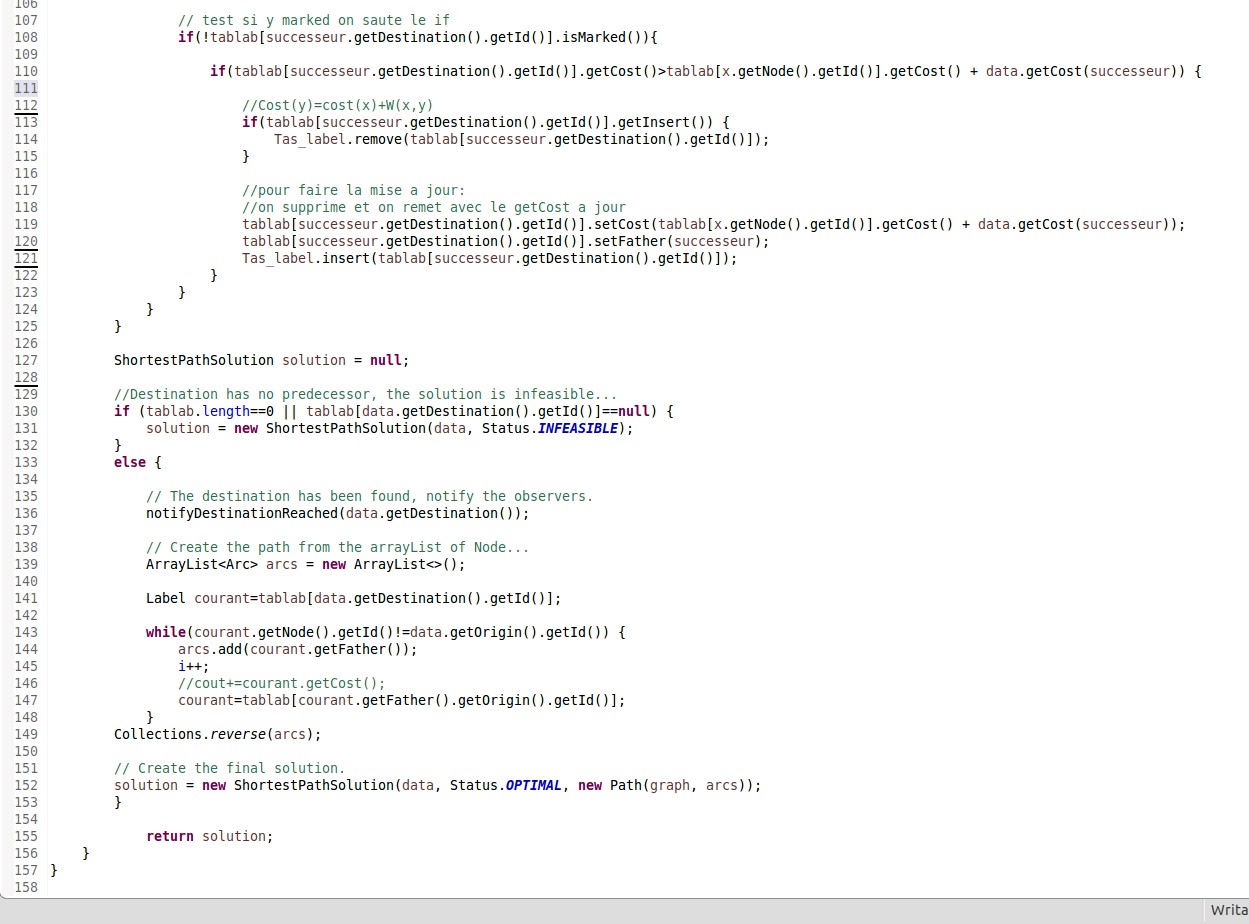
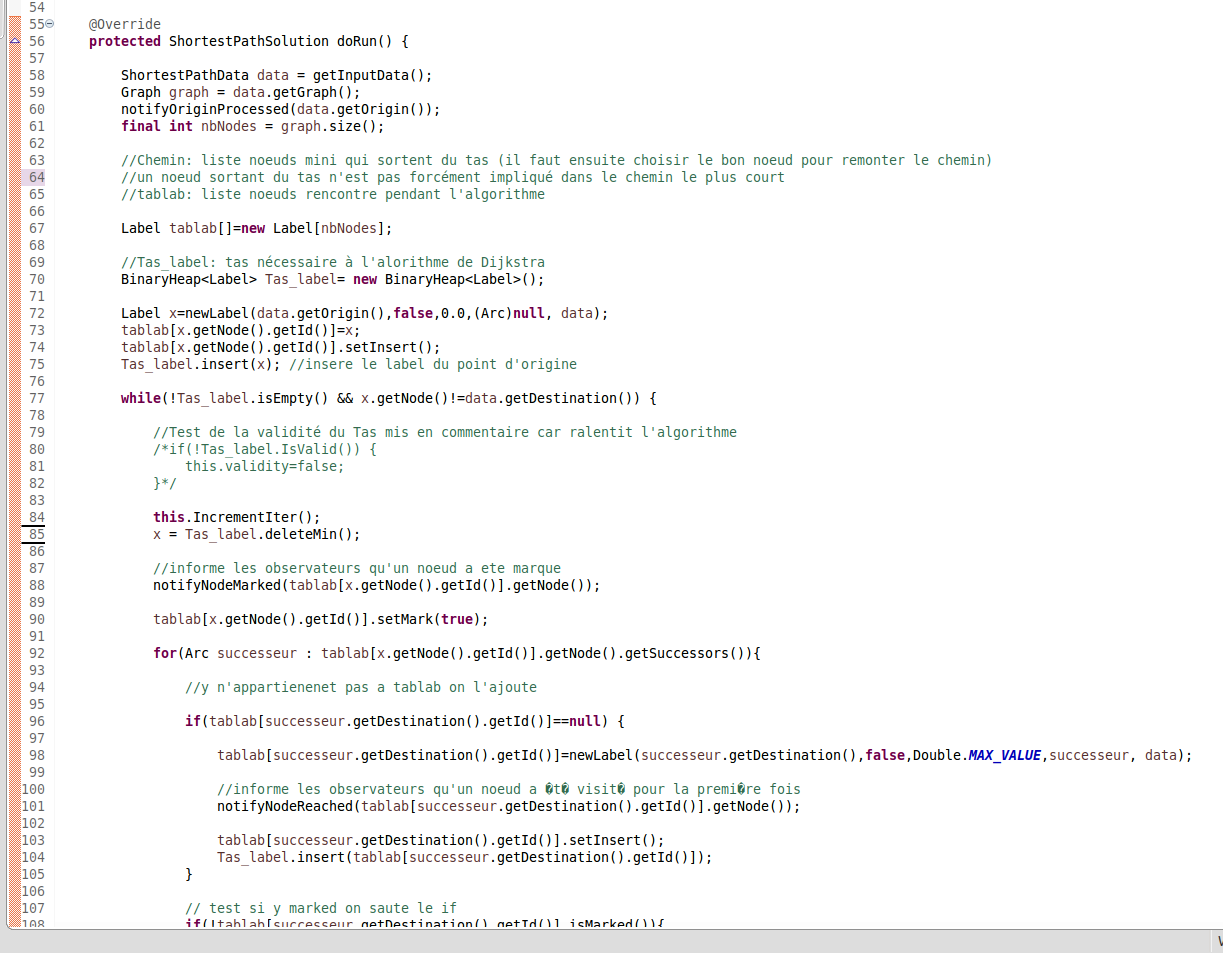




1. **Code de Dijkstra**







1. **Explication du code**

Pour implémenter l’algorithme de Dijkstra, nous avons eu besoin d’une nouvelle classe “Label” permettant de sauvegarder les caractéristiques de chaque sommet. Un Label contient donc:

* Le sommet courant qui sera le sommet associé à ce Label
* Un attribut marque qui est un boolean indiquant si le sommet a été marqué ou non
* Son coût
* Le père qui correspondra au sommet précédent sur le chemin correspondant au plus court chemin.

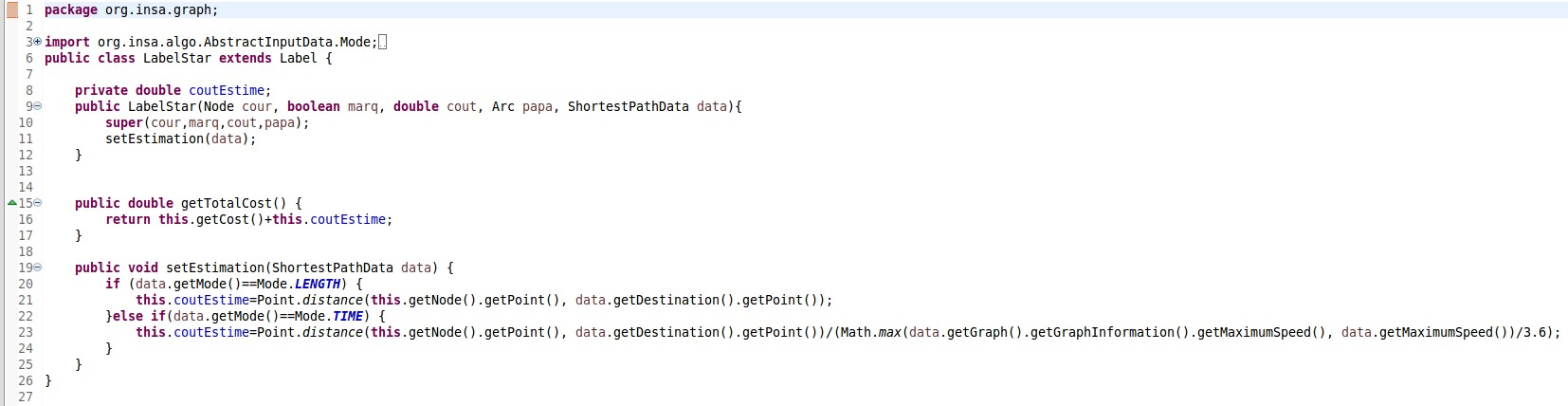
Une fois la classe Label créée, nous avons pu coder Dijkstra. Nous nous sommes inspirés du pseudo-code présent dans le polycopié du cours. Néanmoins, nous avons choisi de ne pas initialiser en début d’algorithme le coût de tous les sommets à l’infinie. Nous initialisons les sommets seulement lorsque l’algorithme les rencontres.

Nous disposons de plusieurs structures de données. Un tas dans lequel les Labels sont rangés par ordre de coût ainsi qu’un tableau qui recense tous les Labels qui ont été rencontrés pendant l’algorithme.

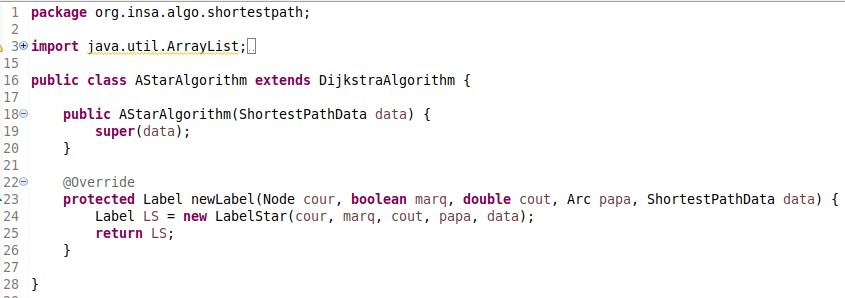
Tant que le tas n’est pas vide ou que le noeud minimum du tas n’est pas le noeud de Destination, alors, on sort le noeud de coût minimum, on le marque, on met à jour tous ses successeurs si l’ancien coût est supérieur au nouveau.

Lorsque l’algorithme se termine, on cherche le prédécesseur du Label de Destination et ainsi de suite jusqu’à arriver au Label d’Origine. On a plus qu’à inverser la liste obtenue pour avoir le chemin optimal.

1. **Algorithme de A\***
2. **Code de LabelStar**



1. **Code de A\***



1. **Explication de A\***

A\* est une extension de l’algorithme de Dijkstra ce qui explique que son code est court. C’est pourquoi la classe de A\* hérite de la classe DijsktraAlgorithm. Dans A\*, nous ne rangeons plus les sommets par seul ordre de coût mais avec une fonction de deux coûts: coût par rapport à l’origine plus coût estimé jusqu’à la destination.

Tout comme pour Dijkstra, nous avons dû créer une classe LabelStar pour sauvegarder nos informations sur les noeuds visités. LabelStar contient en plus des propriétées héritées de Label, un attribut “Coût estimé”. Il correspond à la distance entre le noeud courant et la destination finale à vol d’oiseau dans le cas d’une recherche de chemin le plus court. Dans le cas de la recherche du chemin le plus rapide, c’est la distance restante à vol d’oiseau divisé par la la vitesse maximum autorisée. Ces deux estimations, appelés heuristiques, permettent de guider l’algorithme vers la destination.

Pour que l’algorithme de Dijkstra puisse s’adapter en celui de A\* en fonction des besoins, nous créons une méthode newLabel dans l’algorithme de Dijkstra que nous redéfinissons dans celui de A\*. Cette méthode nous permet ainsi de créer un Label si on est dans Dijsktra et un LabelStar si nous sommes dans A\*.

# **Tests de correction**

Nous avons pu prendre en main le Framework JUnit qui permet de tester notre programme de manière automatique et non pas de seulement faire des affichages de nos résultats. Nous avons réalisé tous nos tests grâce à ce Framework.

Nous avons mené divers tests de corrections portants sur des cartes fictives et réelles. Ces tests ont été validés. Les tests portaient sur des chemins les plus courts en distance ainsi que sur la recherche de chemin les plus rapides en temps. Nous les avons faits pour l’algorithme de Dijkstra et A\*.

Pour réaliser nos tests de correction nous avons utilisés le graphe (cf fig.1) proposés dans l’énoncé du BE (il a ensuite été retiré mais nous l’avions fini avant):

# 

fig. 1: Graphe de test

**Résultat attendu:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | x1 | x2 | x3 | x4 | x5 | x6 |
| x1 | - | 7,x1 | 8,x1 | 10,x5 | 8,x2 | 10,x3 |
| x2 | 10,x3 | - | 3,x5 | 3,x5 | 1,x2 | 4,x5 |
| x3 | 7,x3 | 2,x3 | - | 5,x5 | 3,x2 | 2,x3 |
| x4 | ∞ | ∞ | ∞ | - | ∞ | ∞ |
| x5 | 9,x3 | 4,x3 | 2,x5 | 2,x5 | - | 3,x5 |
| x6 | 12,x3 | 7,x3 | 5,x5 | 5,x5 | 3,x6 | - |

Ci-dessus le tableau des plus courts chemins depuis un point de départ (colonne de gauche) jusqu’à la destination (ligne du haut). Nous avons ainsi fait les tests de notre Dijkstra sur ce graphe et tous les résultats ont été justes (le fichier des résultats édité est en annexe p.16).

Ensuite nous avons comparé les résultats avec ceux de l’algorithme de Bellman-Ford fourni pour vérifier que nous obtenions bien exactement le même chemin, voici quelques exemples:

**Chemin nul : Carré dense**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithme** | **Mode** | **Origine** | **Destination** | **Coût** |
| Dijkstra | Shortest | 133444 | 133444 | 0 |
| A\* | Shortest | 133444 | 133444 | 0 |
| Dijkstra | Fastest | 12970 | 12970 | 0 |
| A\* | Fastest | 12970 | 12970 | 0 |

**Chemin inexistant : Carte INSA**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithme** | **Mode** | **Origine** | **Destination** | **Coût** |
| Dijkstra | Shortest | 1037 | 955 | No path found |
| A\* | Shortest | 1037 | 955 | No path found |
| Dijkstra | Fastest | 1037 | 955 | No path found |
| A\* | Fastest | 1037 | 955 | No path found |

**Carte midi-pyrénées:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithme** | **Mode** | **Origine** | **Destination** | **Coût** |
| Bellman-Ford | Shortest | 50911 | 526981 | 157,240km |
| Dijkstra | Shortest | 50911 | 526981 | 157,240 km |
| Bellman-Ford | Fastest | 50911 | 526981 | 1h48min27s |
| Dijkstra | Fastest | 50911 | 526981 | 1h48min27s |
| Bellman-Ford | Shortest | 514015 | 459008 | 203,568 km |
| Dijkstra | Shortest | 514015 | 459008 | 203,569 km |
| Bellman-Ford | Fasttest | 514015 | 459008 | 2h39min0s |
| Dijkstra | Fasttest | 514015 | 459008 | 2h39min0s |

**Carte Carré-dense:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithme** | **Mode** | **Origine** | **Destination** | **Coût** |
| Bellman-Ford | Shortest | 201949 | 324703 | 41,754 km |
| Dijkstra | Shortest | 201949 | 324703 | 41,754 km |
| Bellman-Ford | Fastest | 33185 | 306613 | 1h21min56s |
| Dijkstra | Fastest | 33185 | 306613 | 1h21min56s |

Suite à ces différents tests nous pouvons être sur que notre algorithme de Dijkstra fonctionne correctement et trouve le bon chemin à chaque fois.

# **Tests de performances**

Nous avons repris les tests de corrections auxquels nous avons ajouté deux appels système pour connaître le temps d’exécution des tests pour chaque algorithme. Nous avons pu remarquer que l’algorithme de Bellman-Ford est le plus lent suivi de l’algorithme de Dijkstra, enfin l’algorithme A\* est le plus rapide. Nous avons aussi vérifié en même temps que les chemins soient égaux entre Dijkstra et A\*.

Voici quelques uns des résultats obtenus:

**Carte Bretagne:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Algorithme** | **Mode** | **Origine** | **Destination** | **Coût** | **Temps exécution** |
| Dijkstra | Shortest | 1504 | 450393 | 189,1934 km | 655 ms |
| A\* | Shortest | 1504 | 450393 | 189,1934 km | 595 ms |
| Dijkstra | Shortest | 185614 | 85086 | 128.3032 km | 540 ms |
| A\* | Shortest | 185614 | 850393 | 128.3032 km | 372 ms |
| Dijkstra | Fastest | 185614 | 85086 | 84.6632 min | 747 ms |
| A\* | Fastest | 185614 | 85086 | 84.6632 min | 629 ms |

**Carte Midi-Pyrénées:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Algorithme** | **Mode** | **Origine** | **Destination** | **Coût** | **Temps exécution** |
| Dijkstra | Fastest | 185614 | 85086 | 2h6min14s | 1289 ms |
| A\* | Fastest | 185614 | 85086 | 2h6min14s | 1279 ms |
| Dijkstra | Fastest | 18 | 85086 | 1h36min54s | 1380 ms |
| A\* | Fastest | 18 | 85086 | 1h36min54s | 1279 ms |
| Dijkstra | Shortest | 18 | 886 | 4.1474 km | 171 ms |
| A\* | Shortest | 18 | 886 | 4.1474 km | 44 ms |
| Dijkstra | Shortest | 12000 | 15641 | 13.8563 km | 112 ms |
| A\* | Shortest | 12000 | 15641 | 13.8563 km | 51 ms |
| Dijkstra | Shortest | 4219 | 15 | 28.4625 km | 159 ms |
| A\* | Shortest | 4219 | 15 | 28.4625 km | 79 ms |

**Carte Belgique:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithme** | **Mode** | **Origine** | **Destination** | **Temps exécution** |
| Dijkstra | Fastest | 966454 | 8836 | 1430 ms |
| A\* | Fastest | 966454 | 8836 | 1454 ms |
| Dijkstra | Shortest | 538885 | 12354 | 714 ms |
| A\* | Shortest | 538885 | 12354 | 204 ms |
| Dijkstra | Shortest | 15641 | 124 | 579 ms |
| A\* | Shortest | 15641 | 124 | 133 ms |

Nous remarquons que l’algorithme A\* est bien plus rapide que l’algorithme de Dijkstra, cependant l’écart se réduit lorsqu’il s’agit du mode fastest, A\* semble donc plus adapté pour les plus courts chemins (mode Shortest).

# **Problème ouvert**

Nous avons choisi le problème du covoiturage. Deux usagers veulent covoiturer, nous devons trouver le point de rencontre optimal pour qu’ils effectuent le moins de kilomètres. Ils peuvent faire le trajet chacun de leur côté dans le pire des cas. Nous avons choisi le problème dans sa version la plus simple lorsqu’il n’y a que deux covoitureurs qui ont la même destination.

Voici l’algorithme utilisé:

Faire un A\* entre U1 et D: A\*1, et entre U2 et D: A\*2

Faire un A\* entre U1 et U2: A\*3

Trouver le milieu du trajet U1-U2 : I

Faire un A\* entre I et D :A\*4

Trouver le point situé à ⅓ de la distance en partant de I : G

Faire un A\* entre U1 et G, A\*5, et entre U2 et G, A\*6

Faire un A\* entre G et D: A\*7

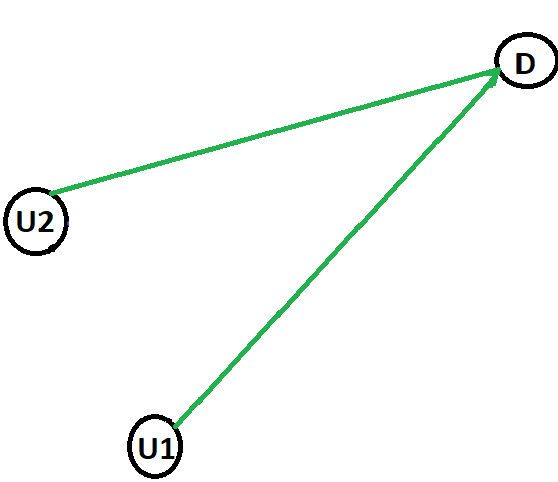
If((A\*1+A\*2)<(A\*5+A\*6+A\*7){

Path=A\*1+A\*2

}else{Path=A\*5+A\*6+A\*7}

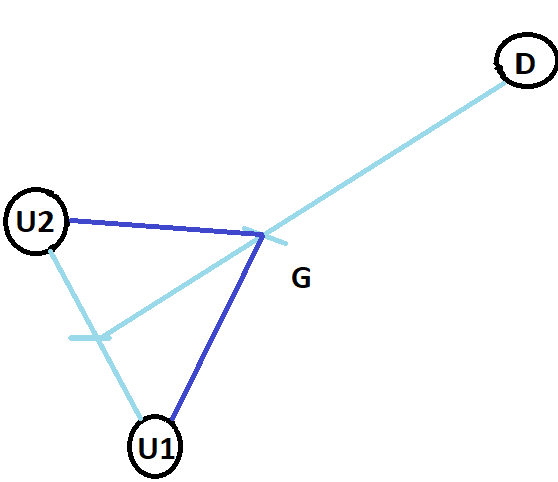
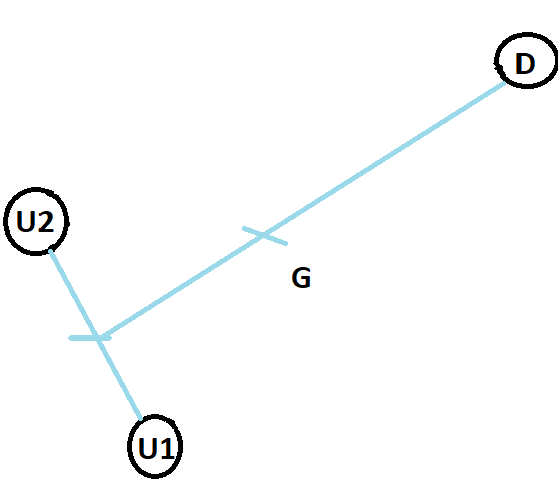
On a représenté ce problème sous la forme d’un triangle, avec pour sommets les usagers et la destination. Nous avons trouvé que le point de rencontre optimal s’il existe est le centre de gravité de ce triangle.

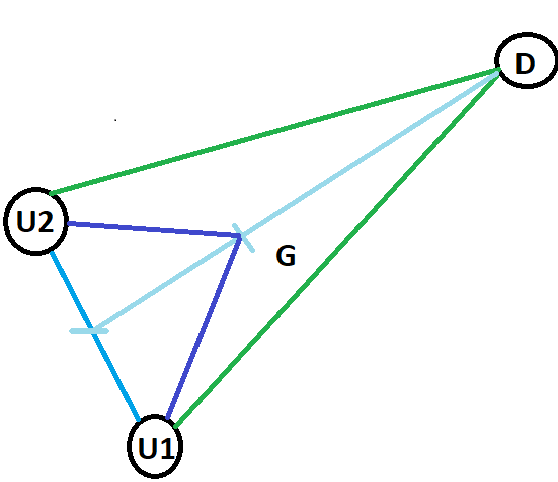
Nous trouvons donc le chemin le plus rapide entre les deux usagers. S’il est plus long que le chemin de la somme des trajets entre les usagers et la destination alors nous décidons que le covoiturage n’est pas une solution optimale.



Ensuite nous calculons le point qui est le plus proche de la moitié du coût du chemin le plus court entre les deux usagers, appelé Intersection.

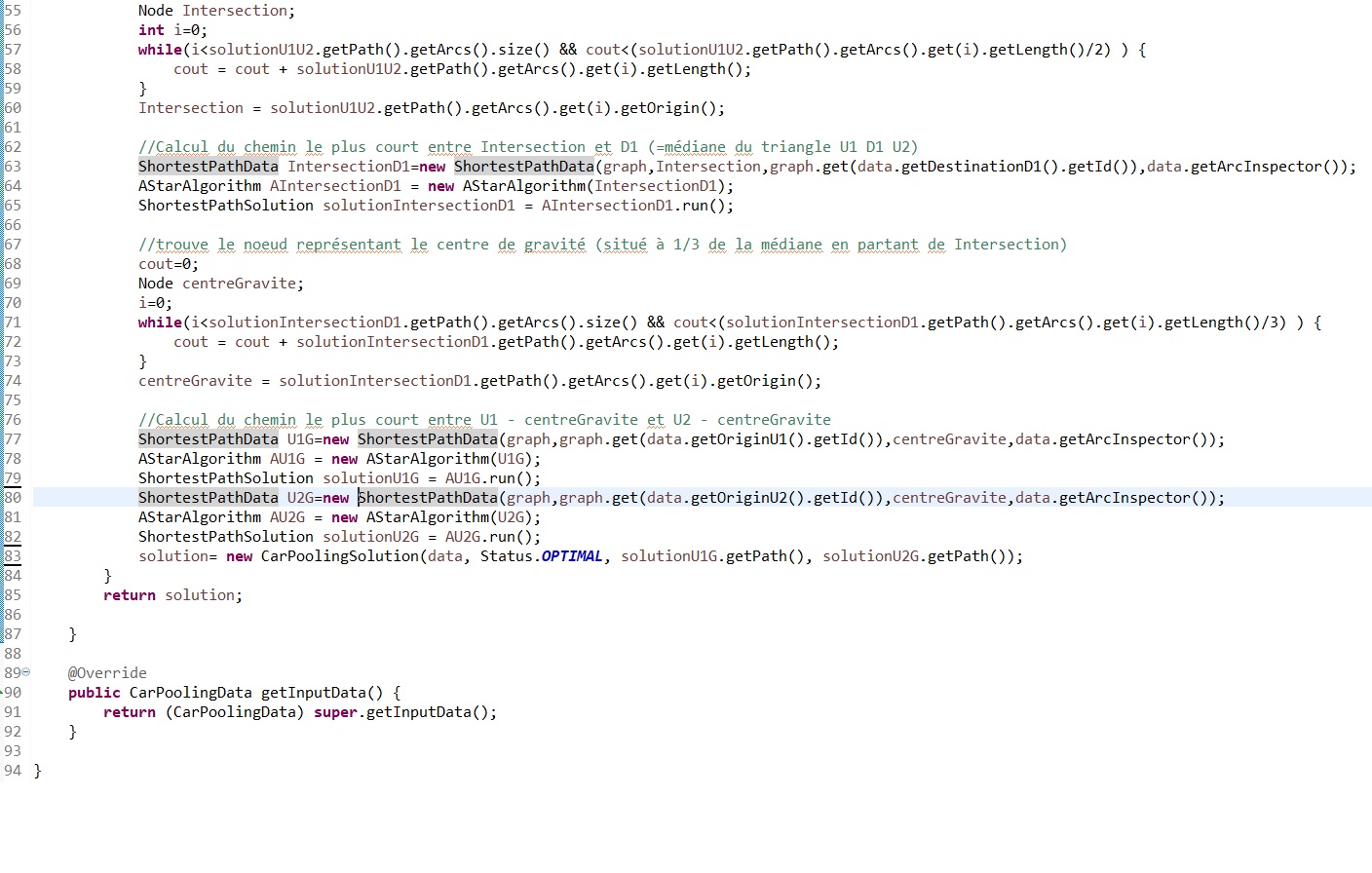
Nous trouvons le plus court chemin entre intersection et la destination. Le centre de gravité se trouve à 1/3 de distance à partir de Intersection.





Le chemin optimal pour chaque usager est donc le trajet jusqu’au centre de gravité puis jusqu’à leur destination.





Nous avons commencé l’implémentation du problème ouvert mais nous n’avons pas eu le temps de la finir. Les observateurs ne fonctionnent pas lorsque nous lançons un test de Car-Pooling. De plus, nous avons plusieurs paths, nous ne savons pas s’il faut les concaténer ou alors si les résultats peuvent être plusieurs paths différents.

**Conclusion**

Pour conclure, nous avons appris au cours de ce bureau d’étude à mettre en pratique nos compétences en programmation Java et en graphe sur des cas concrets. Nous avons codé des algorithmes de plus courts chemins: Dijkstra et A\* et ainsi observé leurs différences de comportement que ce soit sur les performances ou le fonctionnement de l’algorithme.

De plus, nous avons appris à réaliser une campagne de tests pour vérifier que les algorithmes fonctionnent dans tous les cas possibles: chemins vides, inexistants, court ou encore long tout en s’assurant que les chemins trouvés soient bien les plus courts.

Enfin nous avons pu commencer à travailler sur le problème ouvert pour proposer une solution malgré le fait que nous n’ayons pas eu le temps de finir de coder l’algorithme complètement.

**ANNEXE:**

Le document créé lors des tests du graphe proposé dans le BE.

sol: Found a path from node #0 to node #1, 0.0080 kilometers in 0 seconds.

Nombre d'iteration : 2

sol: Found a path from node #0 to node #2, 0.0080 kilometers in 0 seconds.

Nombre d'iteration : 3

sol: Found a path from node #0 to node #3, 0.0110 kilometers in 0 seconds.

Nombre d'iteration : 6

sol: Found a path from node #0 to node #4, 0.0090 kilometers in 0 seconds.

Nombre d'iteration : 4

sol: Found a path from node #0 to node #5, 0.0100 kilometers in 0 seconds.

Nombre d'iteration : 5

sol: Found a path from node #1 to node #4, 0.0010 kilometers in 0 seconds.

Nombre d'iteration : 2

sol: Found a path from node #2 to node #4, 0.0030 kilometers in 0 seconds.

Nombre d'iteration : 4

sol: No path found from node #3 to node #4 in 0 seconds.

Nombre d'iteration : 1

sol: org.insa.algo.shortestpath.DijkstraAlgorithm@2d127a61

Nombre d'iteration : 5

sol: Found a path from node #2 to node #0, 0.0080 kilometers in 0 seconds.

Nombre d'iteration : 6

sol: Found a path from node #5 to node #4, 0.0030 kilometers in 0 seconds.

Nombre d'iteration : 2

sol: Found a path from node #1 to node #0, 0.0110 kilometers in 0 seconds.

Nombre d'iteration : 6

sol: Found a path from node #1 to node #5, 0.0040 kilometers in 0 seconds.

Nombre d'iteration : 5

sol: Found a path from node #1 to node #4, 0.0010 kilometers in 0 seconds.

Nombre d'iteration : 0

sol: No path found from node #3 to node #5 in 0 seconds.

Nombre d'iteration : 1

sol: org.insa.algo.shortestpath.DijkstraAlgorithm@2bbaf4f0

Nombre d'iteration : 4

sol: No path found from node #3 to node #0 in 0 seconds.

Nombre d'iteration : 1

sol: Found a path from node #5 to node #1, 0.0070 kilometers in 0 seconds.

Nombre d'iteration : 5

sol: Found a path from node #1 to node #2, 0.0030 kilometers in 0 seconds.

Nombre d'iteration : 4

sol: Found a path from node #2 to node #1, 0.0020 kilometers in 0 seconds.

Nombre d'iteration : 2

sol: No path found from node #3 to node #2 in 0 seconds.

Nombre d'iteration : 1

sol: Found a path from node #4 to node #2, 0.0020 kilometers in 0 seconds.

Nombre d'iteration : 3

sol: org.insa.algo.shortestpath.DijkstraAlgorithm@11c20519

Nombre d'iteration : 4

sol: Found a path from node #4 to node #0, 0.0100 kilometers in 0 seconds.

Nombre d'iteration : 6

sol: Found a path from node #1 to node #3, 0.0030 kilometers in 0 seconds.

Nombre d'iteration : 3

sol: Found a path from node #2 to node #3, 0.0050 kilometers in 0 seconds.

Nombre d'iteration : 5

sol: No path found from node #3 to node #1 in 0 seconds.

Nombre d'iteration : 1

sol: Found a path from node #4 to node #3, 0.0020 kilometers in 0 seconds.

Nombre d'iteration : 2

sol: org.insa.algo.shortestpath.DijkstraAlgorithm@70beb599

Nombre d'iteration : 3

sol: Found a path from node #5 to node #0, 0.0130 kilometers in 0 seconds.

Nombre d'iteration : 6