INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Rapport

du Bureau d'Études de Graphes

Rédigé par :

Rémi GASCOU - Lina MHIRI

- Version du 2 juin 2018 -

Table des matières

Introduction

Le but de ce BE, est d'appréhender de façon concrète les algorithmes de plus court chemin.

Nous allons implémenter des algorithmes de plus court chemins en Java, puis nous allons les tester sur différentes cartes de différentes tailles.

Nous allons dans un premier temps détailler la conception des classes de notre projet, puis nous allons développer les différents tests que nous avons effectués sur nos algorithmes. Enfin, nous allons mettre en relations toutes ces compétences afin de résoudre un problème concret basé sur la recherche de plus court chemins.

1 Conception des classes

1.1 Structure du projet et classes importantes

Avant de concevoir les classes implémentant les algorithmes de plus courts chemins, nous avons tout d'abord identifier les classes utilisées pour représenter les graphes et les chemins. Ces classes de base se trouvent dans le package `org.insa.graph` et sont illustrées par le diagramme de classe suivant.

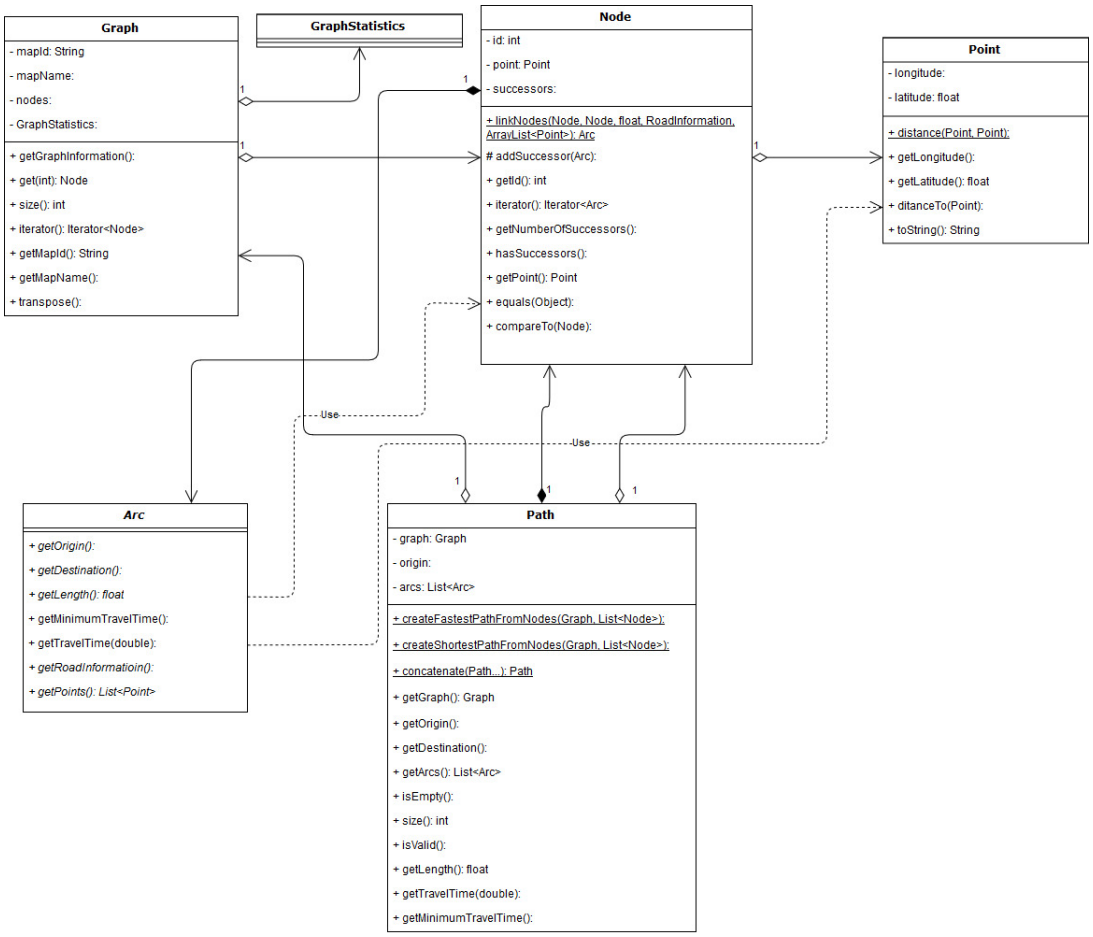


FIGURE 1 – Diagramme des classes du projet

Par la suite, nous avons fait le même travail pour comprendre comment les algorithmes de plus courts chemins devaient être développés, en fonction des classes qu'ils devaient implémenter et utiliser.

Pour cela, nous avons réalisé un diagramme de classe des packages `org.insa.algo` et `org.insa.algo.shortestpath`. En effet, le premier contient les filtres utilisés par les algorithmes ainsi que des classes abstraites nécessaires à la création d'algorithmes (classes pour les données d'entrées, pour les algorithmes et pour les données en sortie) et le second contient ces dernières implémentées pour les algorithmes de plus courts chemins. Le diagramme de classe de ces packages est donné ci-dessous.

Nous avons finalement créé les classes `DikstraAlgorithm` et `AStarAlgorithm` pour implémenter les algorithme de Dijkstra et d'AStar. Ces deux algorithmes reposent sur l'utilisation d'un tas min (la racine est l'élément le plus petit) qui est implémenté par la classe BinaryHeap (classe dont générique), ainsi que du marquage et du coût des noeuds. Nous avons donc développé une classe Label permettant d'associer un marquage et un coût à un noeud. Cette classe a été implémentée dans le package `org.insa.algo.utils`.

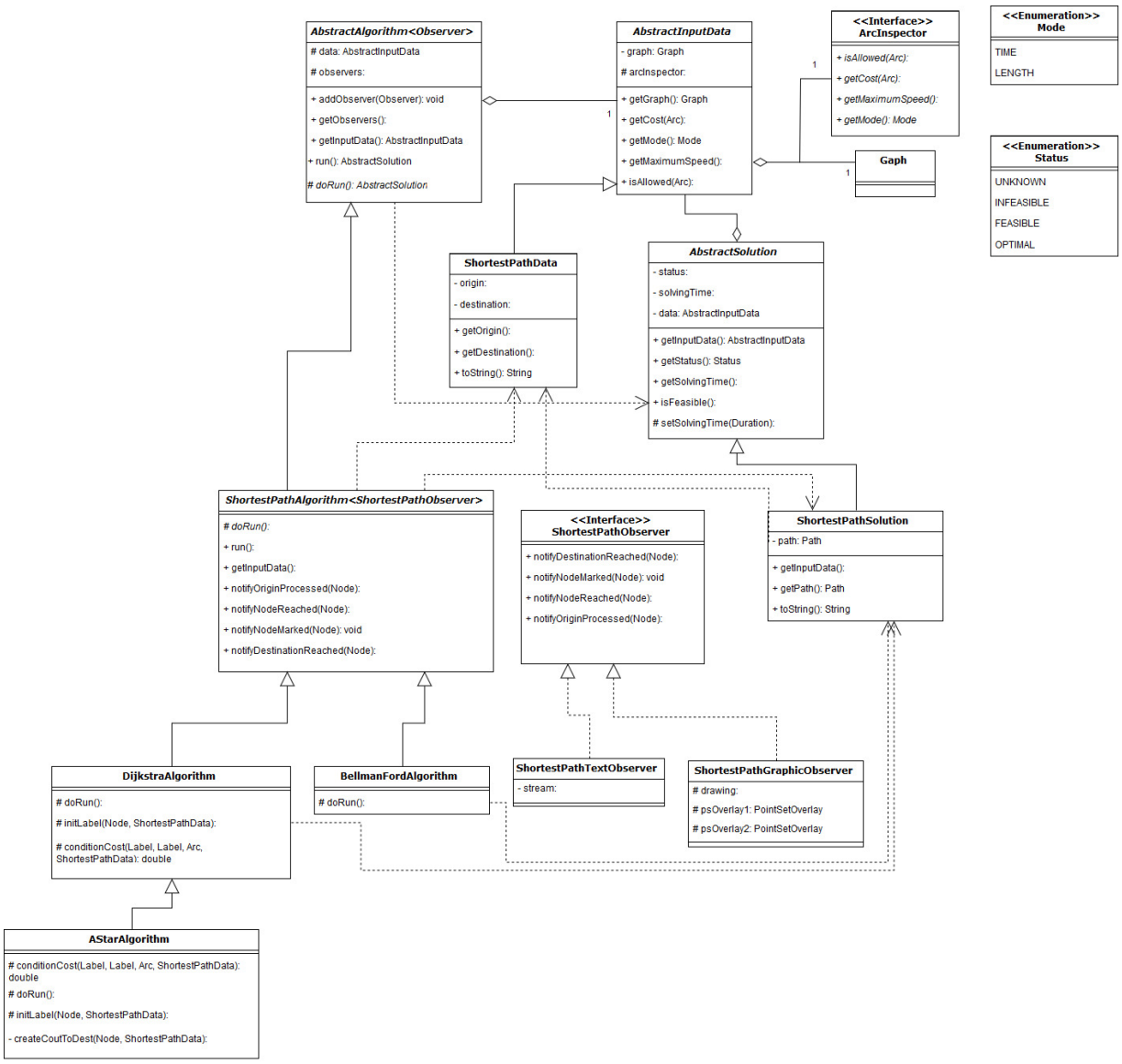


FIGURE 2 – Diagramme de classes des algorithmes

1.2 Les classes Label et LabelStar

Ces deux classes sont fondamentales pour les algorithmes de Dijkstra et AStar puisqu'elles représentent les éléments qui vont être comparés et placés dans le tas. En effet, la classe Label implémente l'interface `Comparable<Label>` en redéfinissant la méthode `compareTo()` ce qui permet de comparer des Labels par leurs coûts (donc des nodes). Ces deux classes permettent donc de marquer des noeuds et de les ordonner par leur coût, qui évolue au fil des itérations, dans le tas min (permettant de récupérer avec une complexité la plus faible possible le noeud de coût minimal pas encore marqué).

La différence entre l'algorithme de Disjkstra et d'Astar repose dans le classement des noeuds : pour Dijkstra, les coûts sont calculés en sommant les coûts des arcs alors que pour Astar le coût du chemin à vol d'oiseau du noeud à la destination est ajouté au coût précédent. Par conséquent, nous avons créé la classe LabelStar qui prend en compte ce coût à vol d'oiseau dans la comparaison de deux noeuds (i.e. deux LabelStar). Les attributs et méthodes de ces deux classes sont illustrées par le diagramme ci-après.

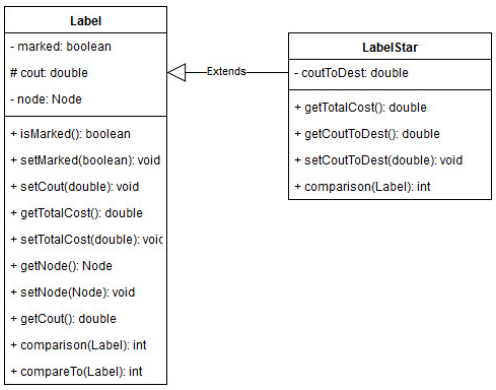


FIGURE 3 – Diagramme de classes du projet

1.3 L'algorithme Dijkstra

Notre implémentation de l'algorithme de Dijkstra hérite de la classe `ShortestPathAlgorithm`. La partie principale de l'algorithme est contenue dans une méthode appelée doRun(). Nous avons également créé les méthodes `initTas()`, `initLabel(Node node, ShortestPathData data)` et `conditionCost(Label ly, Label lx, Arc arc, ShortestPathData data)` afin de pouvoir par la suite implémenter l'algorithme A* par héritage de Dijkstra en ne redéfinissant que certaines méthodes.

L'algorithme que nous avons développé n'est pas exactement celui de Dijkstra dans le sens où nous arrêtons l'algorithme dès que nous marquons le noeud de destination (alors que Dijkstra ne s'arrête que lorsque tous les noeuds sont marqués). Cette condition peut être facilement supprimée pour trouver les plus courts chemins de un vers tous.

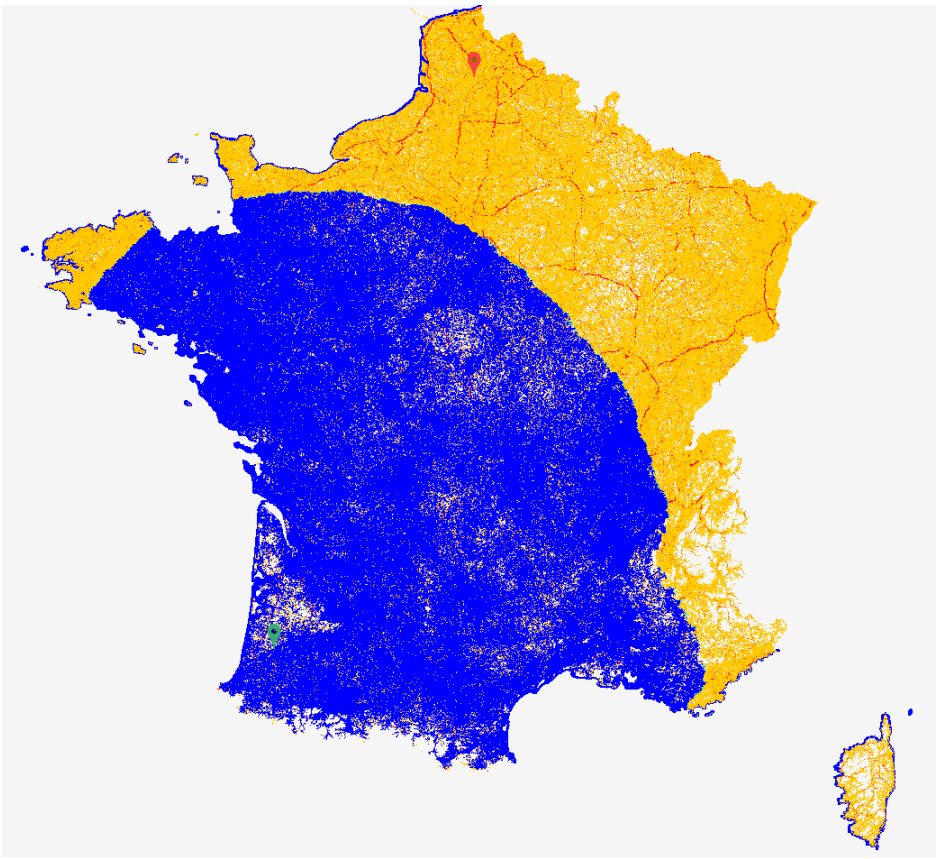


FIGURE 4 – Progression de l'algorithme de Dijkstra sur la carte de France

1.4 L'algorithme A*

L'algorithme A* est un algorithme de recherche de plus court chemin ayant un fonctionnement analogue à celui de Dijkstra. Dans notre implémentation, il est hérité directement de l'algorithme Dijkstra. Il est nécessaire de redéfinir les méthodes suivantes :

- `initTas()` :
  Méthode utilisée pour créer le BinaryHeap contenant les labels des noeuds parcourus par l'algorithme.
- `initLabel(Node node, ShortestPathData data)` :
  Méthode utilisée pour créer le label associé au noeud.
- `conditionCost(Label ly, Label lx, Arc arc, ShortestPathData data)` :
  Méthode permettant de calculer le coût à mettre dans le Label.

L'algorithme A* est une variante de l'algorithme de Dijkstra dirigé vers la destination. Pour cela, la distance à la destination entre en jeu dans le coût des noeuds calculé à chaque étape.

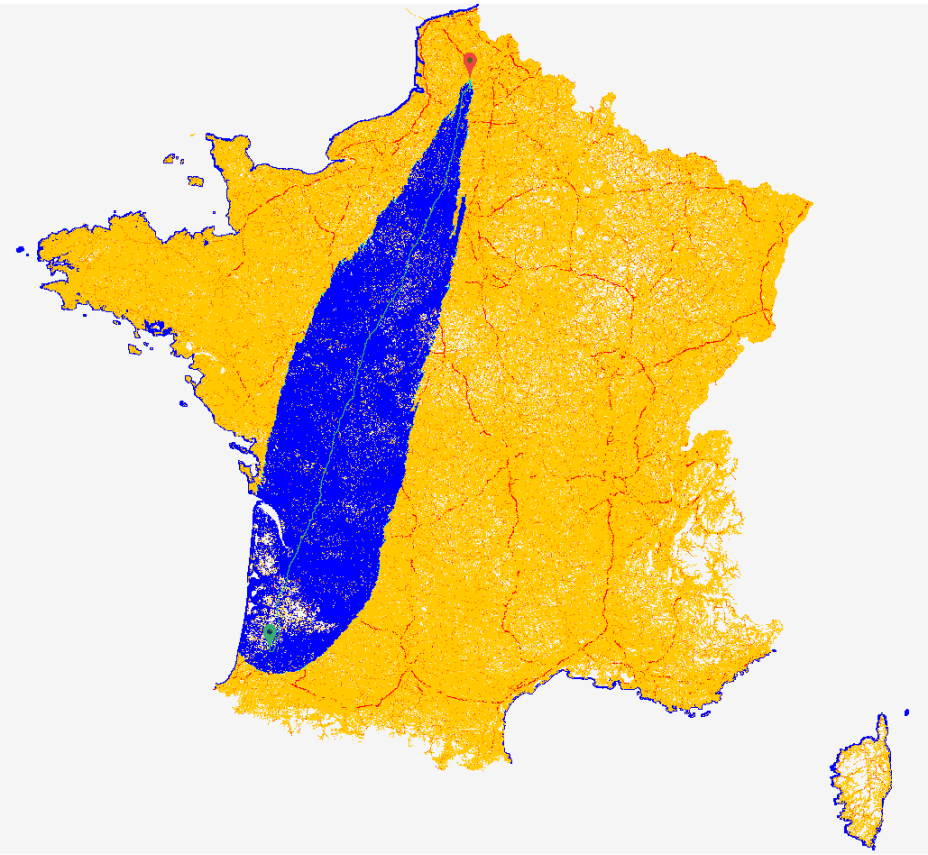
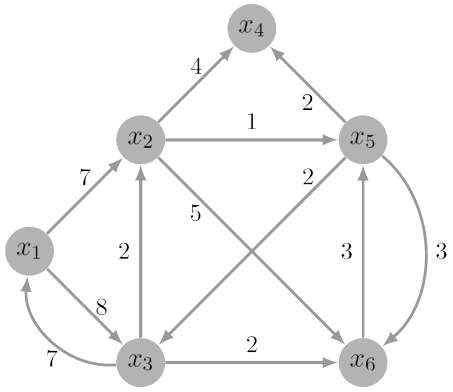


FIGURE 5 – Progression dirigée de l'algorithme A* sur la carte de France

2 Tests de validité sur Dijkstra et A* en distance et en temps

Pour démontrer l'exactitude des résultats obtenus avec Dijkstra et AStar, nous avons mis en place trois types de tests : un test JUnit qui vérifie leur fonctionnement sur un graphe simple, des tests de validité qui le vérifie sur des chemins variés et des tests de plus court chemin de sous chemins.

2.1 JUnit test



Ce premier type de test consiste à vérifier la fonctionnalité des algorithmes de Dijkstra et A* en comparant leur résultat avec celui de Bellman-Ford sur un graphe simplifié illustré ci-contre. Nous recherchons les plus courts chemins de tous les noeuds vers tous les noeuds. Les résultats attendus sont les données dans le tableau ci-dessous. Ces tests sont réalisés dans la classe `DijkstraAlgorithmTest` et `AStarAlgorithmTest` (qui hérite de `DijkstraAlgorithmTest` car les tests sont les mêmes, seule l'initialisation varie).

Ces tests sont réalisés dans la classe `DijkstraAlgorithmTest` et `AStarAlgorithmTest` (qui hérite de `DijkstraAlgorithmTest` car les tests sont les mêmes, seule l'initialisation varie). Dans cette classe, nous définissons un méthode qui est exécutée avant le test (`@BeforeClass`) et qui construit le graphe en créant des noeuds et les liant avec la méthode statique `Node.linkNodes()`. Dans le cas d'AStar, il est nécessaire de créer des points pour chaque noeud (inutile pour Dijkstra) pour le calcul du coût à vol d'oiseau, c'est pourquoi cette méthode est redéfinie dans la classe de test d'AStar. Les tests sont ensuite réalisés en exécutant Bellman-Ford et Dijkstra ou AStar (la méthode retournant l'algorithme à tester est redéfinie dans la classe pour les tests d'AStar) d'un noeud vers un autre pour tous les noeuds du graphe. Nous testons pour chaque chemin s'il est valide et si la distance, la taille et la durée sont égales à celles obtenues avec Bellman-Ford. Nous obtenons un tableau similaire avec l'algorithme de Dijkstra, d'AStar et de Bellman-Ford.

	x_1	x_2	x_3	x_4	x_5	x_6
x_1		7 (x_1)	8 (x_1)	10 (x_5)	8 (x_2)	10 (x_3)
x_2	10		3 (x_5)	3 (x_5)	1 (x_2)	4 (x_5)
x_3	7 (x_3)	2 (x_3)		5 (x_5)	3 (x_2)	2 (x_3)
x_4	$+\infty$	$+\infty$	$+\infty$		$+\infty$	$+\infty$
x_5	9 (x_3)	4 (x_3)	2 (x_5)	2 (x_5)		3 (x_5)
x_6	12 (x_3)	7 (x_3)	5 (x_5)	5 (x_5)	3 (x_6)	

FIGURE 6 – Résultats obtenus par application de l'algorithme de Bellman-Ford

2.2 Les tests de validité

Les tests de validité que nous avons mis en place comparent les résultats de nos algorithmes (Dijkstra et AStar) avec ceux obtenus avec Bellman-Ford. Pour garantir que les algorithmes soient corrects sur le maximum de cas possibles, nous testons des chemins différents en temps et en distance et avec des modes de transport différents (voiture, piéton, vélo). Les trajets que nous avons choisis sont :

- Trajets courts avec les 3 modes de transports.
- Trajets réservés aux piétons.
- Trajets réservés aux voitures.
- Trajet dans les deux sens avec les 3 modes de transports.
- Trajet ABC et comparaison avec $AB + BC$ avec les 3 modes de transports.

Nous couvrons ainsi, en prenant les cartes de l'INSA, de la Haute-Garonne et de Midi-Pyrénées, des chemins de tailles variables (allant de 400m à 270km). Les trajets réservés à un seul mode de transport permettent de vérifier que les algorithmes n'empruntent pas des routes interdites mais aussi qu'ils ne trouvent pas de chemins lorsqu'il n'y en a pas. Les tests sur des chemins dans les deux sens (i.e. test de A à B puis de B à A) donnent des résultats sensiblement différents à cause de routes à sens unique mais que l'on peut considérer comme assez proche pour être validés. La comparaison entre les chemins ABC et $AB + BC$ permet de vérifier l'inégalité triangulaire mais aussi que les deux sous-chemins sont des plus courts chemins. Nous avons voulu réaliser des tests de validité sur la carte de la France pour des chemins plus long mais le temps d'exécution de Bellman-Ford ne le permettait pas. Par conséquent, des trajets de distance de l'ordre 900km ont été effectué pour les tests de performance et leur comparaison manuelle permet de valider ce cas de chemin.

Pour réaliser les tests de validité, nous avons implémenté une classe `Scenario` qui comprend le graphe (la carte), l'origine, la destination, le mode d'évaluation (temps ou distance) et le mode de transport (voiture, vélo, piéton) pour un trajet. Nous avons ensuite développé une classe abstraite `ValidityTest` qui compare les résultats obtenus avec Bellman-Ford et un autre algorithme sur 22 scénarios (i.e. 22 trajets). Pour chaque test, nous vérifions qu'un chemin a été trouvé et si c'est le cas, nous comparons selon le mode temps ou distance le temps de trajet minimal ou la distance du chemin trouvé. Les algorithmes étant similaires mais pas identiques, des petites différences dans le temps ou la distance peuvent apparaître. Nous avons donc considéré comme validé un test dont le résultat varie de 1% par rapport à la valeur attendue.

Les résultats de nos tests sont positifs puisque pour Dijkstra et AStar tous les tests sont validés. En effet, les seuls tests pour lesquels les résultats ne sont pas exactement les mêmes avec l'algorithme d'AStar sont pour les chemins ABC, AB et BC. Sur le chemin ABC en voiture, nous observons une différence de 26m sur une distance de plus de 234km ce qui acceptable. De plus, la somme des distances de AB et BC en voiture est égale à la distance de ABC déterminée par Bellman-Ford, ce qui permet de valider le test. Les autres tests de ce type sont concluants.

Grâce à ces tests, nous pouvons affirmer que les algorithmes d'AStar et Dijkstra fonctionnent correctement sur de nombreux cas. Une autre vérification de leur fonctionnement est sans oracle et consiste à tester que des sous chemins de plus courts chemins sont bien des plus courts chemins.

2.3 Test des sous chemins

Les tests précédents sont basés sur une comparaison des résultats obtenus avec Dijkstra ou AStar et Bellman-Ford, et reposent donc sur l'exactitude de Bellman-Ford. De plus, Bellman-Ford ayant un temps d'exécution particulièrement long, cela limite les chemins de test possibles. Une autre façon de tester nos algorithmes est de vérifier qu'ils satisfassent bien les propriétés des plus courts chemins, en particulier celle affirmant que les sous chemins de plus courts chemins sont des plus courts chemins.

Nous avons donc développé une classe abstraite `SousCheminTest` étendue par deux classes (`Dijkstra SousCheminTest` et `AStarSousCheminTest`) pour tester cette propriété sur 24 chemins et sous chemins avec les modes de comparaison en temps et en distance et les trois modes de transport cités précédemment. Les tests consistent à trouver le plus court chemin entre deux noeuds A et B, d'identifier deux noeuds C et D appartenant à ce chemin et de lancer le même algorithme pour trouver le plus court chemin entre C et D (illustration ci dessous). Nous parcourons ensuite ce dernier en vérifiant que les noeuds sont les mêmes que ceux entre C et D dans le plus court chemin AB. Cette vérification noeud peut induire une erreur dans le cas où il y a plusieurs plus courts chemins entre C et D avec la même distance ou le même temps. Pour éviter cela, nous considérons que le test est valide si tous les noeuds sont les mêmes ou si la distance (respectivement la durée) est égale dans les deux cas (plus court chemin entre C et D et chemin entre C et D sur le plus court chemin entre A et B).

FIGURE 7 – Diagramme de classes des algorithmes

Les tests de plus court chemin des sous chemins étant tous validés pour les algorithmes de Dijkstra et d'AStar, nous sommes convaincus du fonctionnement de ces deux algorithmes.

3 Tests de performance sur Dijkstra et A* en distance et en temps

L'intérêt de développer l'algorithme d'A* est son temps d'exécution qui doit être inférieur à celui de Dijkstra et de Bellman-Ford. Pour prouver qu'A* trouve bien le plus court chemin plus rapidement que les deux autres algorithmes, nous développons des tests de performance qui chronomètrent l'exécution des algorithmes sur des chemins de distance allant de 900m à 900 km. Nous souhaitons aussi estimer l'impact du type de coût (distance ou temps) et du mode de transport (voiture, vélo ou piéton) sur les temps d'exécution, c'est pourquoi chaque algorithme est exécuté 6 fois pour chacun des 5 chemins. Les chemins choisis sont accessibles par les trois modes de transport (excepté un) ce qui permet de couvrir des cas de trajets différents.

La classe regroupant les tests est la classe abstraite `PerformanceTest` qui est étendue par les classes `DijkstraPerformanceTest`, `AStarPerformanceTest` et `BellmanFordPerformanceTest`, chacune exécutant les tests avec un algorithme. Pour éviter que le ramasse miette ne se produise pendant l'exécution d'un algorithme, nous utilisons la méthode `System.gc()` pour que cette opération soit effectuée avant l'exécution de l'algorithme (pour éviter que le temps d'exécution de cette fonction doit pris en compte dans la mesure).

Nous avons choisi d'exécuter les tests de performance sur la carte de la France pour visualiser nettement les variations de temps d'exécution dues aux différences dans les implémentations. L'inconvénient de ce choix est l'impossibilité d'obtenir des résultats avec l'algorithme de Bellman-Ford puisque son temps d'exécution dépasse les 7 heures pour le plus court de nos chemins. Nous faisons donc la comparaison entre les temps d'exécution de Dijkstra et d'A*. Pour comparer de façon visuelle, nous avons réalisé des graphiques ci-dessous du temps d'exécution en fonction de la distance et du temps pour le mode de transport voiture et piétons.

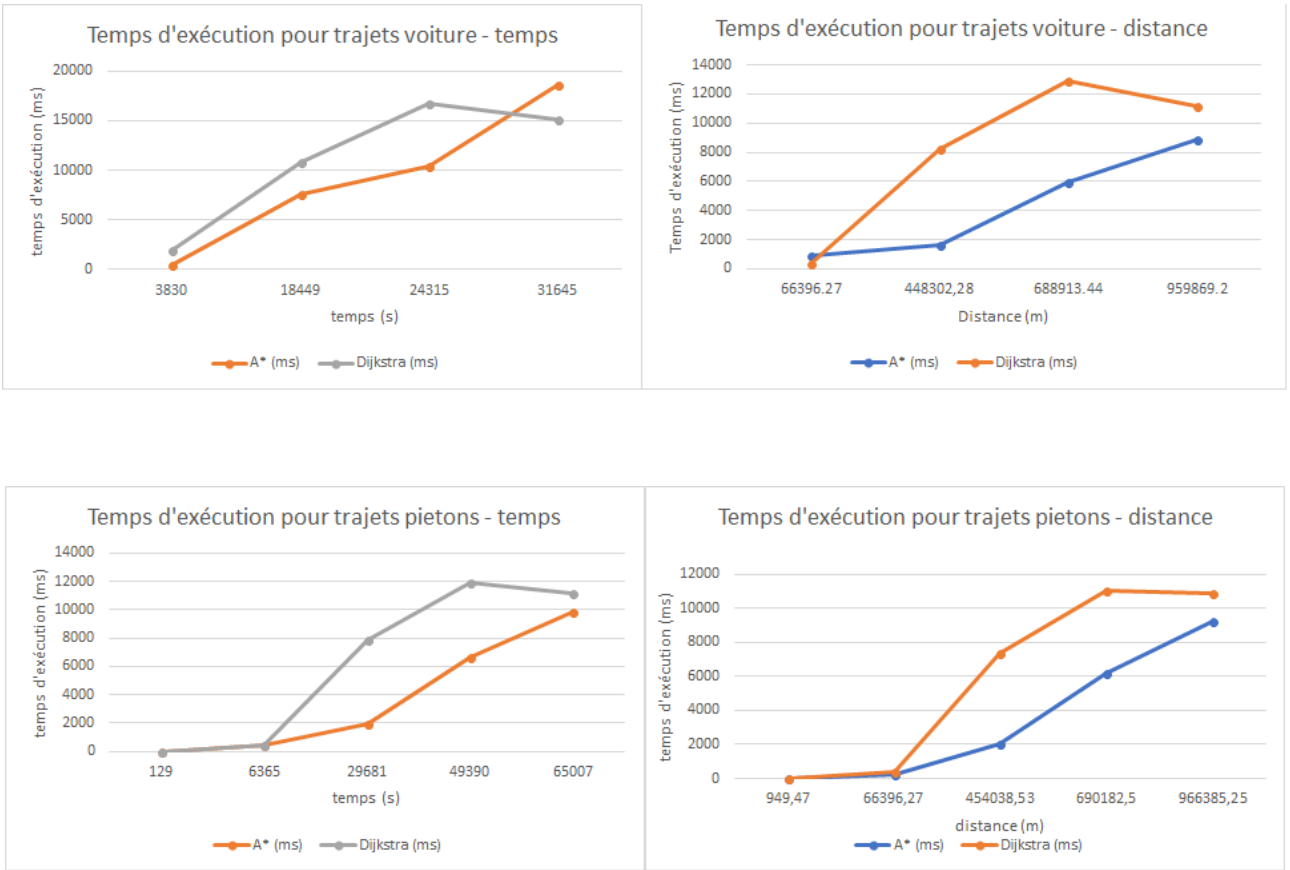


FIGURE 8 – Comparaison des temps d'exécution des algorithmes

Nous pouvons constater que le temps d'exécution d'A* est en moyenne plus faible que celui de Dijkstra. Cependant, la différence est assez faible pour des courts ou des trés long chemins mais elle est élevée pour des chemins de distance moyenne. Lorsque les chemins ont une courte distance, les algorithmes sont équivalents puisque les noeuds explorés par Dijkstra éloignés de la destination sont peu nombreux et le temps utilisé à l'exploration de ces noeuds revient approximativement au temps mis par A* pour calculer les coûts à destination à vol d'oiseau des noeuds visités. La faible différence pour des très long chemins est plus difficilement explicable mais doit être due aux calculs des coûts à destination à vol d'oiseau. Une large différence est visible au niveau du troisième point sur les graphes des trajets en voiture et le quatrième sur celui piéton, puisque le point de départ se situe au centre de la France (illustré sur la figure suivante). Dans ce cas, l'avantage de la directivité d'A* est conséquente car la différence entre le nombre de noeuds à explorer pour Dijkstra et A* est significative.

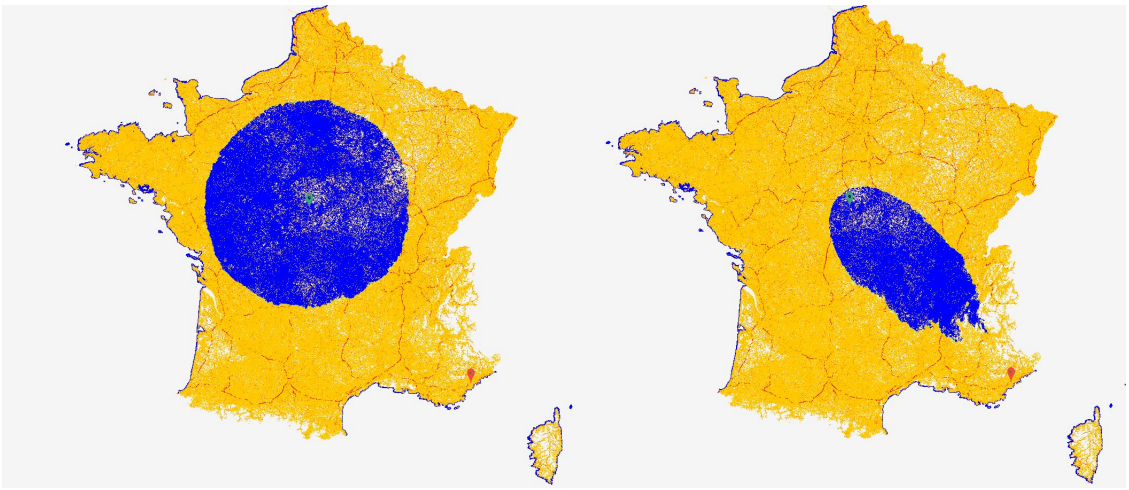


FIGURE 9 – Progression des algorithmes Dijkstra et A*

Nous pouvons aussi observer une variation dans l'ordre de grandeur des temps d'exécution lorsque le mode de comparaison est la distance ou le temps. Une raison peut être que les coûts des arcs sont des distances ce qui pourrait engendrer un nombre de calculs plus importants lorsque des durées sont comparées. Nous pouvons donc déduire que pour des chemins de courte distance ou durée, il n'y a pas un choix qui donne un résultat nettement plus efficace entre Dijkstra et AStar. Pour des chemins plus longs en distance ou en temps et/ou dont le nombre de noeuds autour du point de départ est très important (point au centre d'une carte), il est préférable d'utiliser A* pour avoir des résultats plus rapidement.

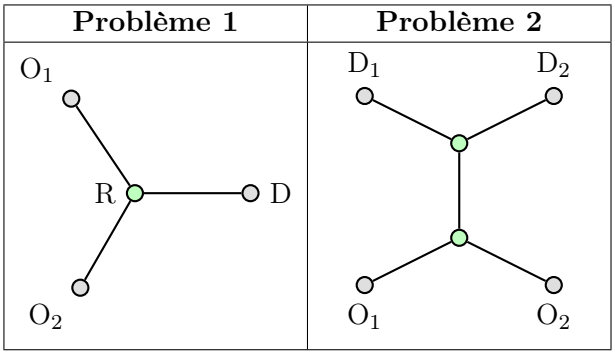
4 Problème ouvert

Trois problèmes nous ont été proposés :

— **Problème 1 :** *Covoiturage*
— **Problème 2 :** *Échange de colis entre deux robots*
— **Problème 3 :** *Point de rencontre*

Nous avons choisi le problème 1. Cependant, grâce à une fine analyse du problème, on remarque que les solutions aux problèmes 1 et 2 peuvent être obtenus grâce à la même méthode.

En effet, la solution aux problèmes 1 et 2 consiste en la recherche d'un arbre de Steiner. Ci-contre sont représentées les arbres de Steiner théoriques que nous obtenons pour les deux problèmes considérés. Dans notre cas, nous nous plaçons dans le triangle O_1O_2D. L'arbre de Steiner ne contient donc qu'un seul point, cela revient à la recherche du point de Toricelli dans le triangle O_1O_2D.



Dans le cas du problème 1, nous ne nous basons que sur 3 points. Ainsi nous proposons l'algorithme suivant :

Algorithme pour 3 points :

- Calcul des coordonnées cartésiennes théoriques du point de Steiner S.
- Recherche d'un node A de notre graphe approximant la solution S du point de Steiner trouvé en (3).
- Recherche des plus courts chemins AO_1, AO_2, et AD grâce à un algorithme de plus courts chemins.
- Tracé du parcours.

Conclusion

Nous avons étudié les algorithmes de plus court chemins ainsi que leur implémentation en Java.

Nous avons ensuite soumis nos algorithmes à des tests de performance afin de comparer l'efficacité de ses algorithmes dans différents cas.

Pour finir nous avons appliqué ces connaissances des algorithmes de plus courts chemins à un problème concret, le covoiturage de deux usagers.

INSA Toulouse
135, Avenue de Rangueil
31077 Toulouse Cedex 4 - France
www.insa-toulouse.fr

Université Fédérale Toulouse Midi-Pyrénées

Liberté • Égalité • Fraternité
RÉPUBLIQUE FRANÇAISE

MINISTÈRE
DE L'ÉDUCATION NATIONALE,
DE L'ENSEIGNEMENT SUPÉRIEUR
ET DE LA RECHERCHE