Bureau d'Étude Graphes

-

Rapport

Cerise CHATELIER
Clarisse ERPELDINGER

3MIC Groupe C

31 mai 2019

Table des matières

Introduction **2**

1 Conception et implémentation des algorithmes de plus court chemin **3**
1.1 Modélisation du problème . . . 3
1.2 Algorithmes de recherche de plus court chemin . . . 4
1.2.1 Implémentation de Dijkstra et création de la classe Label . . . 4
1.2.2 Implémentation de A* et création de la classe LabelStar . . . 5

2 Tests de validité et d'optimalité des algorithmes conçus **6**
2.1 Vérification visuelle . . . 6
2.2 Tests en distance . . . 7
2.3 Tests en temps . . . 7
2.4 Tests d'optimalité sans oracle . . . 8

3 Tests de performance des algorithmes conçus **9**
3.1 Tests de performance visuels . . . 9
3.2 Analyse des résultats des tests automatisés . . . 10
3.2.1 Taille maximale du tas et nombre de sommets visités . . . 10
3.2.2 Temps d'exécution du CPU . . . 11

4 Réflexion sur le problème ouvert : problème de covoiturage **13**
4.1 Le problème . . . 13
4.2 La solution proposée . . . 13
4.3 Généralisation du problème . . . 14

Conclusion **15**

Introduction

Ce bureau d'étude a pour objectif de réaliser un algorithme de recherche de plus court chemin sous contraintes diverses, afin de concrétiser les connaissances acquises en théorie des graphes et programmation Java. Les algorithmes implémentés sont celui de Dijkstra et son extension A*.

Afin de veiller au bon fonctionnement des algorithmes, nous avons élaboré de tests de validité et de performance, dont une partie a été appliquée sur différentes cartes à l'aide d'une interface graphique. Nous avons aussi comparé les deux algorithmes pour souligner l'intérêt de l'utilisation de A* ou Dijstra dans certains cas.

Enfin, nous avons proposé une réflexion quant à l'adaptation de ces algorithmes pour résoudre un problème de covoiturage.

1 Conception et implémentation des algorithmes de plus court chemin

1.1 Modélisation du problème

Les cartes utilisées sont composées entre autres d'un ensemble de sommets (*Node*), d'un ensemble de chemins (*Arc*) et d'un descripteur de chemin fournissant des informations sur ces chemins. Nous avons résumé les liens principaux entre ces classes dans les diagrammes ci-dessous.

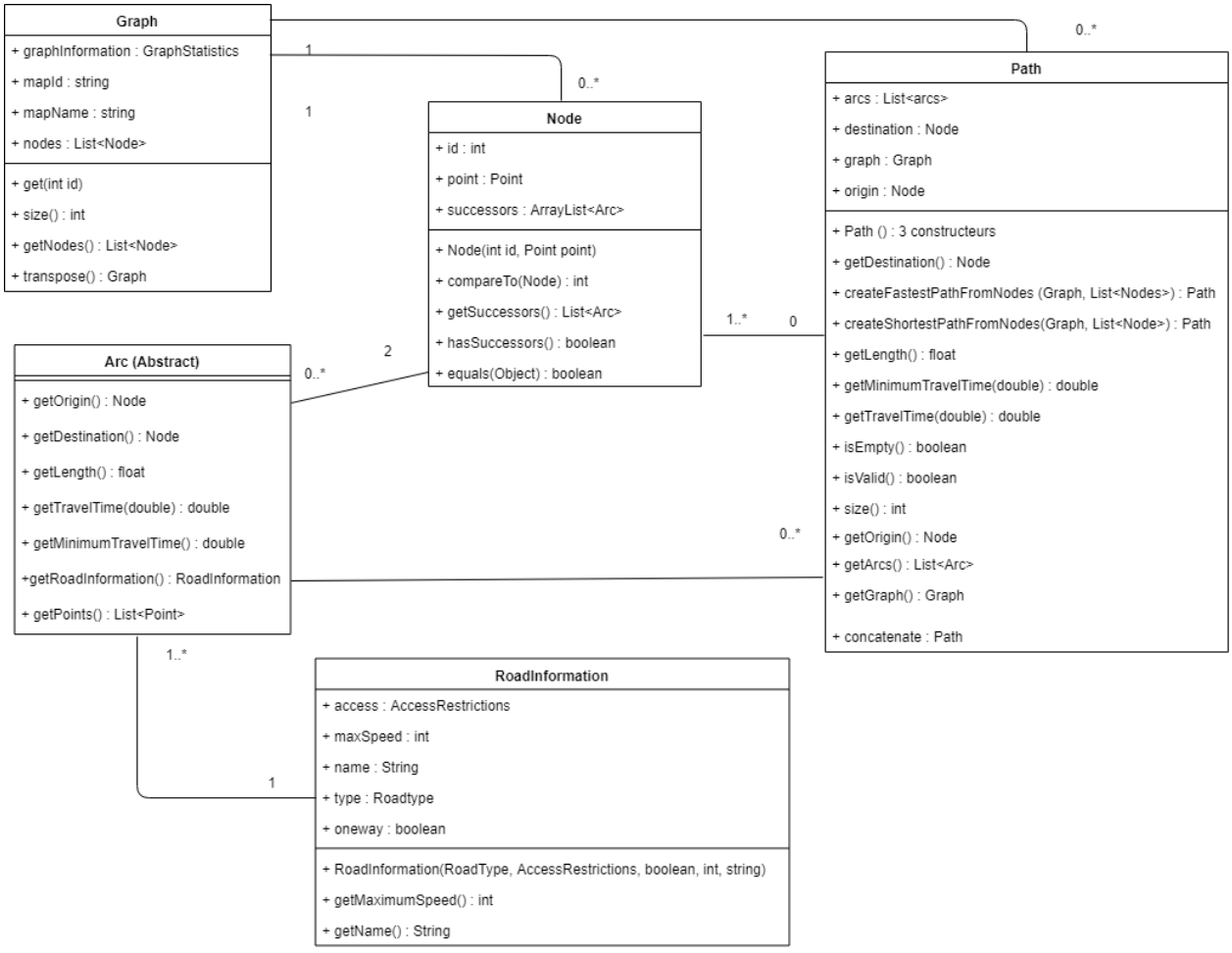


FIGURE 1.1 – Diagramme de classe du package graph

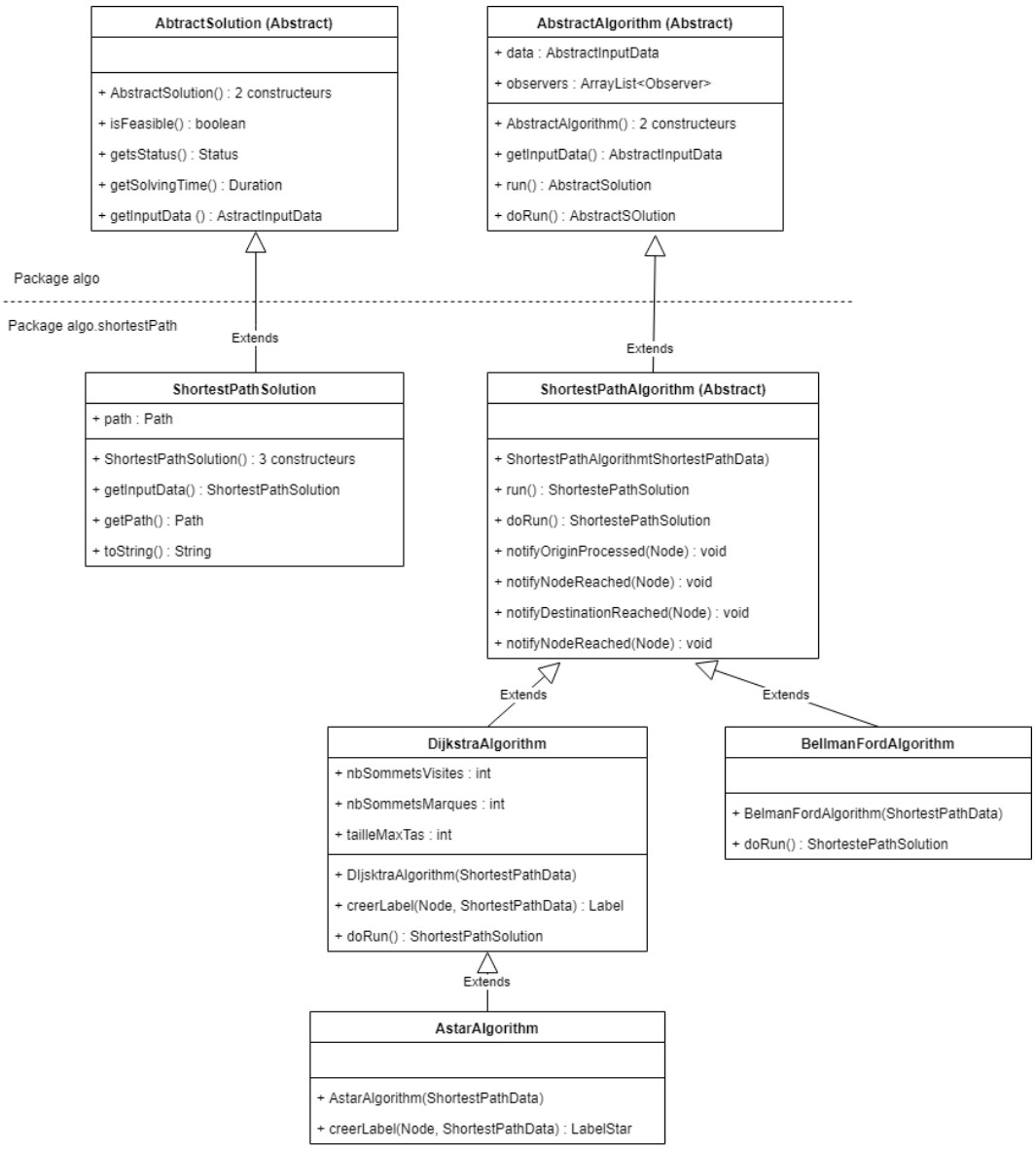


FIGURE 1.2 – Diagramme de classe des packages algo et algo.shortestpath

1.2 Algorithmes de recherche de plus court chemin

1.2.1 Implémentation de Dijkstra et création de la classe Label

L'algorithme de Dijkstra permet de rechercher un plus court chemin avec un coût de trajet optimal que ce soit en distance ou en temps. Sa complexité est logarithmique ($O((m+n) * log(n))$).

Nous avons d'abord créé une classe Label qui associe un label à un sommet (*Node*). Cette classe contient un sommet courant, son sommet prédécesseur, un coût indiquant la valeur du plus court chemin actuel et une marque permettant d'indiquer si le sommet est définitivement connu par l'algorithme.

La classe *Dijkstra* permet de lancer l'algorithme de Dijsktra. Celui-ci est contenu dans la méthode *doRun()*. Dans un premier temps, les éléments nécessaires au lancement sont créés et instanciés. On trouve notamment une liste de Label qui recense tous les labels associés aux sommets du graphe et le tas qui permet de récupérer le label des sommets que l'algorithme est en train de visiter et de retirer celui des sommets marqués. Les sommets marqués sont ceux dont on a visité tous les successeurs, ce qui n'est pas le cas des sommets visités.

Ensuite se trouve le corps de l'algorithme. Il est constitué d'une boucle qui se termine si l'une des deux conditions suivante est remplie : soit le tas est vide, soit le sommet actuellement visité est le sommet de destination. Il faut faire attention à bien mettre à jour le tas dans le cas où le label y est déjà.

La dernière partie de l'algorithme permet de reconstituer la solution trouvée.

1.2.2 Implémentation de A* et création de la classe LabelStar

L'algorithme A* est une spécialisation de celui de Dijkstra, basé sur une heuristique minorante. En effet, celui-ci prend en compte une distance à vol d'oiseau, ce qui permet d'orienter la trajectoire du plus court chemin.

Au niveau de l'implémentation, nous avons fait une classe *AStar* qui est une classe fille de la classe *Dijkstra*. Nous avons également créé une classe *LabelStar* qui hérite de la classe *Label* et nous avons ajouté un attribut *TotalCost* dans la classe *Label*. Dans *LabelStar*, le coût total correspond à la distance (ou temps) à vol d'oiseau ajoutée à la valeur courante du plus court chemin depuis l'origine vers le sommet.

En revanche, dans la classe *Label* le coût total est égal au coût "normal" : la valeur courante du plus court chemin depuis l'origine vers le sommet. De plus, nous avons modifié la méthode *CompareTo()* de *Label* pour prendre en compte ce coût total. Afin de ne pas recopier le contenu de *doRun()* de *Dijkstra* et de changer les Labels en LabelStars, nous avons créé une méthode *CréerLabel()* dans la classe *Dijkstra* qui crée un *Label* ou un *LabelStar* en fonction de la manière dont on l'appelle, puisque cette méthode est redéfinie dans *LabelStar*.

Enfin, pour pouvoir utiliser A* en mode temps, nous avons converti l'estimation de distance à vol d'oiseau en temps en fonction de la vitesse maximale autorisée sur les chemins empruntés. Dans le cas où la vitesse n'est pas précisée, nous avons fixé la vitesse à 80 km/h.

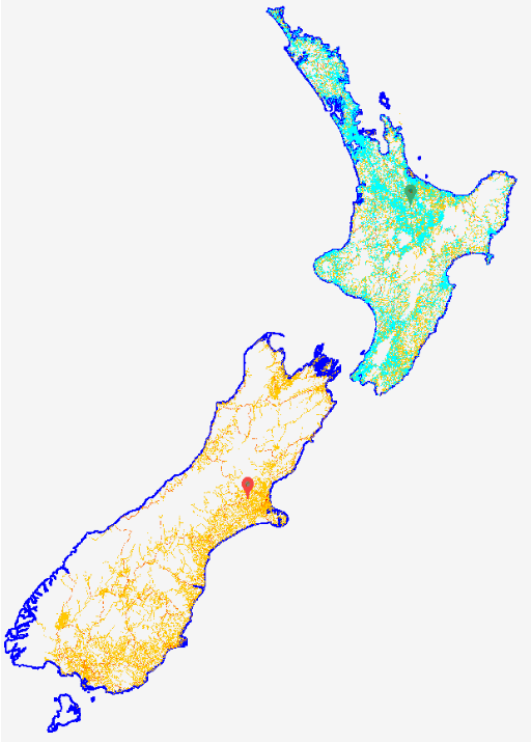
2 Tests de validité et d'optimalité des algorithmes conçus

Dans cette partie, seront étudiées les validations techniques des algorithmes de recherche des plus court chemin réalisées. Les tests ont été réalisés sur un PC portable Windows 10, équipé d'un processeur Intel Core i5, de 6GB de mémoire RAM et d'un disque dur SSD (128G).

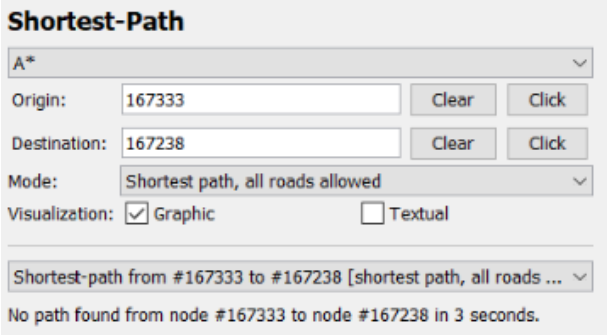
2.1 Vérification visuelle

Afin de vérifier le bon fonctionnement de nos algorithmes, nous avons réalisé des tests visuels via l'interface graphique. En particulier, nous avons créé un test sur la carte de Nouvelle-Zélande. En effet, ce graphe étant non-connexe, il permet de voir le comportement de nos algorithmes dans le cas de chemin impossible (ici, l'origine est sur la partie Nord et la destination sur la partie Sud).

Nos algorithmes fonctionnent bien puisqu'ils s'arrêtent en écrivant un message expliquant qu'il n'existe pas de chemin entre les deux sommets (cf image ci-dessous).



Cas d'un chemin non connexe, Nouvelle-Zélande



2.2 Tests en distance

Afin de tester nos algorithmes et d'éventuellement les corriger, nous avons élaboré des tests JUnit selon plusieurs scénarios :

— chemin dont l'origine et la destination constituent le même sommet ;
— sommet d'origine inexistant ;
— sommet de destination inexistant ;
— sommets d'origine et de destination inexistants ;
— chemin existant.

Ces scénarios ont été testés sur différentes cartes dont la carte "fractal", la Nouvelle-Zélande et le département Midi-Pyrénées. Les tests sur la carte Midi-Pyrénées ont été résumés dans le tableau ci-dessous.

Carte	Algorithme	Mode	Origine	Destination	Coût (m)	Durée (s)
Midi-Pyrénées	Tous	Distance	0	0	0	0
			-1	0	Pas de solution	Pas de solution
			0	-1	Pas de solution	Pas de solution
			-1	-1	Pas de solution	Pas de solution
			2	53	5043.5546875	674
Nouvelle-Zélande	Tous	Distance	2	89858	Pas de solution	Pas de solution

Table 2.1 – Résultats des tests de validité en distance

Nous avons obtenus les mêmes résultats avec les algorithmes de Bellman-Ford, Dijkstra et A* à partir des mêmes données. Ceci est cohérent puisque ces trois algorithmes doivent fournir les mêmes chemins et donc un coût similaire.

2.3 Tests en temps

Pour les calculs en temps, nous avons obtenus les mêmes résultats que pour les tests en distance.

Carte	Algorithme	Mode	Origine	Destination	Coût (s)	Distance (m)
Midi-Pyrénées	Tous	Temps	0	0	0	0
			-1	0	Pas de solution	Pas de solution
			0	-1	Pas de solution	Pas de solution
			-1	-1	Pas de solution	Pas de solution
			2	53	456	7 392
Nouvelle-Zélande	Tous	Distance	2	89858	Pas de solution	Pas de solution

Table 2.2 – Résultats des tests de validité en temps

Nos algorithmes fonctionnent correctement puisque pour la recherche de chemin le plus rapide, le temps de trajet est inférieur à celui du chemin le plus court en distance (et inversement pour la distance du chemin le plus court en distance).

2.4 Tests d'optimalité sans oracle

Dans la partie précédente, nous avons utilisé les résultats obtenus par l'algorithme de Bellman-Ford comme oracle. Cependant, il n'est pas toujours possible de disposer de la valeur de la solution optimale. C'est pourquoi il est nécessaire de réfléchir à d'autres types de tests permettant de vérifier l'optimalité d'une solution obtenue avec Dijkstra et A*, par exemple nous pourrions :

— vérifier que la distance du chemin le plus rapide est supérieure ou égale à la distance du chemin le plus court ;
— vérifier que le temps du chemin le plus rapide est inférieur ou égal au temps du chemin le plus court ;
— dans le cas où il n'existe pas de route à sens unique, lancer l'algorithme en inversant l'origine et la destinations, les résultats devraient être identiques ;
— lancer l'algorithme entre l'origine et la destination, puis le relancer entre l'origine et un sommet du plus court chemin obtenu précédemment (point B de la figure ci-dessous) : les sommets du deuxième chemin doivent tous être des sommets du premiers chemin.

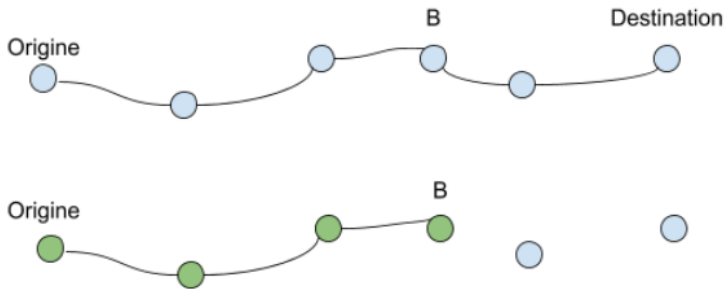


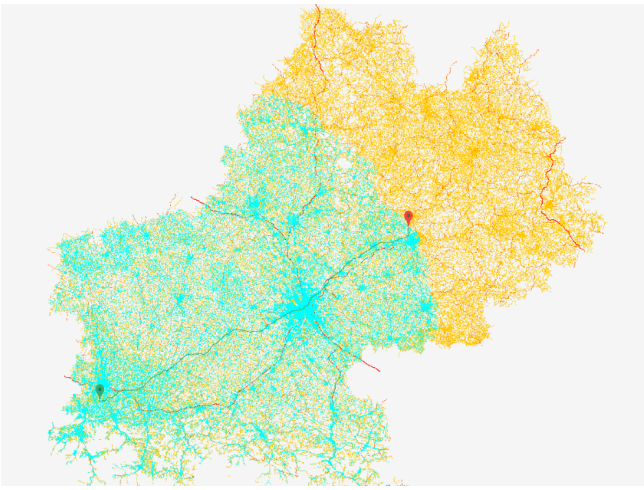
FIGURE 2.1 – Schéma explicatif du 4^{eme} type de test sans oracle

3 Tests de performance des algorithmes conçus

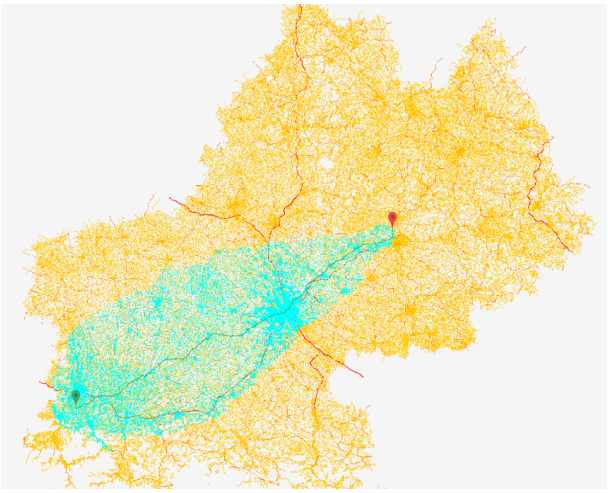
Cette partie de tests permet de comparer les performances des algorithmes Dijkstra et A*. Pour ce faire, nous avons mesuré le temps d'exécution de l'algorithme, le nombre de sommets visités et la taille maximale du tas au cours de l'exécution. Tous les tests ont été réalisés sur un PC portable Windows 10, équipé d'un processeur Intel Core i5, de 6GB de mémoire RAM et d'un disque dur SSD (128G), sauf pour les calculs du temps d'exécution du CPU. Ces derniers ont été réalisés sur un ordinateur fixe Intel core i7 de 8ème génération avec une mémoire RAM 16 GHz, sous Windows 10 et avec un disque dur DDR.

3.1 Tests de performance visuels

Par une simple analyse visuelle, nous avons pu constater les différences entre A* et Dijkstra :
— A* est plus rapide pour calculer le chemin (à partir du moment où il colorie les sommets) ;
— A* parcourt beaucoup moins de sommets.



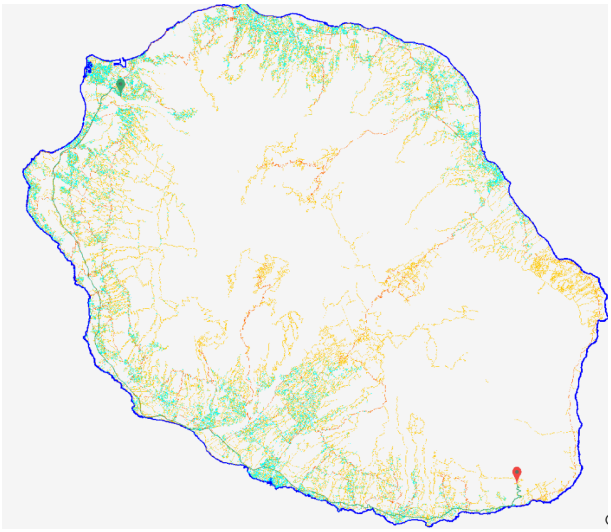
Algorithme de Dijkstra, chemin existant

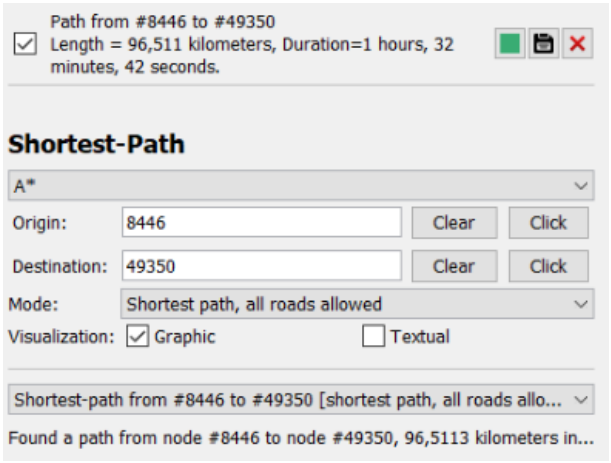


Algorithme A*, chemin existant

Ce sont des résultats auxquels nous nous attendions puisque l'algorithme A* est un type de Dijkstra guidé vers la destination grâce à la distance estimée à vol d'oiseau.

Cependant, on remarque que A* n'est pas optimal pour toutes sortes de cartes. Si on prend une carte comme la réunion par exemple, l'algorithme n'est pas vraiment meilleur que Dijkstra. La distance à vol d'oiseau est trompeuse car il y a très peu de routes au centre de la carte (cf image ci-dessous).





Algorithme A*, chemin existant à la Réunion

De manière générale, les cartes où les plus courts chemins s'éloignent du segment Origine-Destination (par manque de route, présence d'impasses etc.), l'algorithme de A* perd en efficacité.

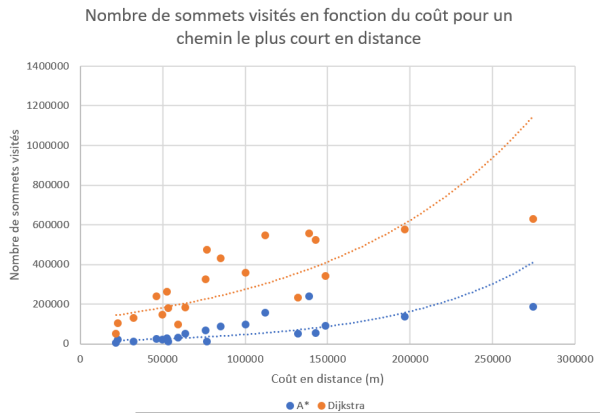
3.2 Analyse des résultats des tests automatisés

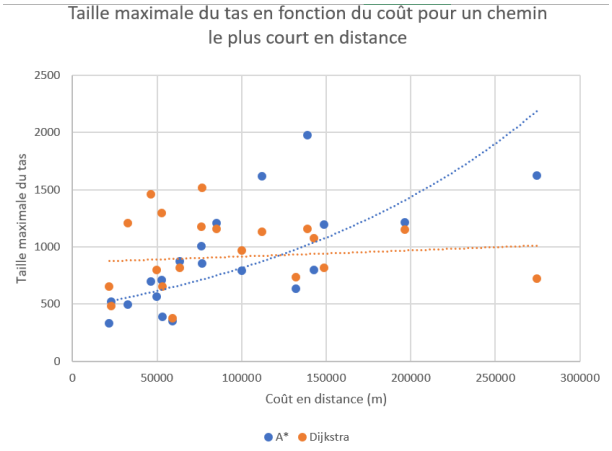
Les tests réalisés ont été automatisés de sorte à créer 100 couples origine-destination aléatoirement à partir des sommets d'une carte donnée et d'y appliquer les algorithmes. Les résultats obtenus sont recensés dans un fichier (un type d'évaluation sur une carte par fichier) puis transcrits dans un tableur Excel.

3.2.1 Taille maximale du tas et nombre de sommets visités

Tests en distance

Nous avons représenté ci-dessous le comportement de la taille maximale du tas et du nombre de sommets visités en fonction du coût du plus court chemin en distance. Les tests ont été réalisés sur les cartes "carré dense" et "Midi-Pyrénées", en utilisant 20 couples origine-destination.

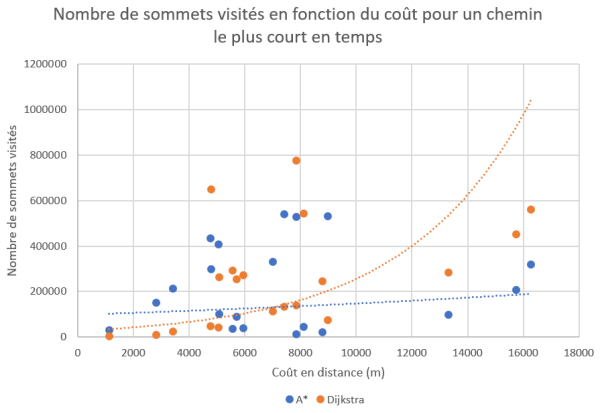


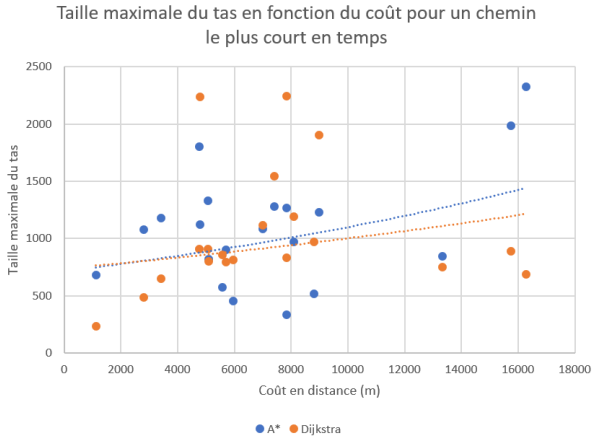


On remarque que A* parcourt moins de sommets que Dijkstra. De plus, l'écart de performance est d'autant plus marqué que la distance augmente. En moyenne, A* visite 8 fois moins de sommets que Dijkstra.

Aussi, on remarque qu'au delà d'une distance de 100 km environ, la taille maximale du tas pour l'algorithme de Dijsktra est inférieure à celle du tas de A*. L'évolution de la taille du tas de Dijkstra évolue de manière assez constante. En effet, pour Dijkstra, la taille du maximale du tas augmente très légèrement quand celle de A* croît beaucoup plus vite. Ce résultat cohérent puisque A* est un Dijkstra orienté vers la destination.

Tests en temps





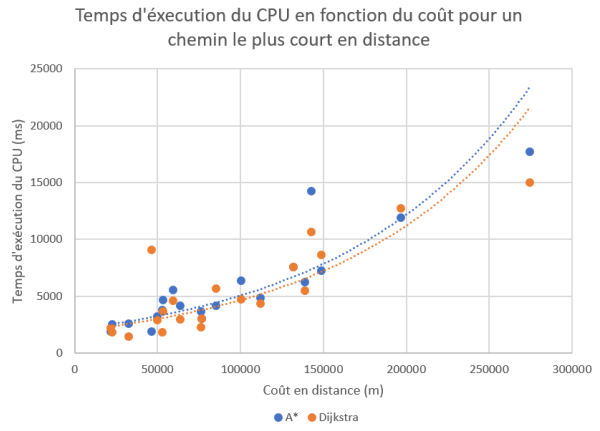
Contrairement aux tests en distance, on remarque l'algorithme A* visite plus de sommets que Dijkstra jusqu'à un certain point où la tendance est inversée . A partir d'un chemin d'environ 7 km, Dijkstra visite plus de sommets. En moyenne, A* visite 3,5 fois moins de sommets que Disjkstra. La taille maximale du tas de Dijkstra et de A* est quasiment similaire en deça d'une distance de 5,5 km. Pour une distance supérieure, le tas de A* est supérieur à celui de Dijsktra.

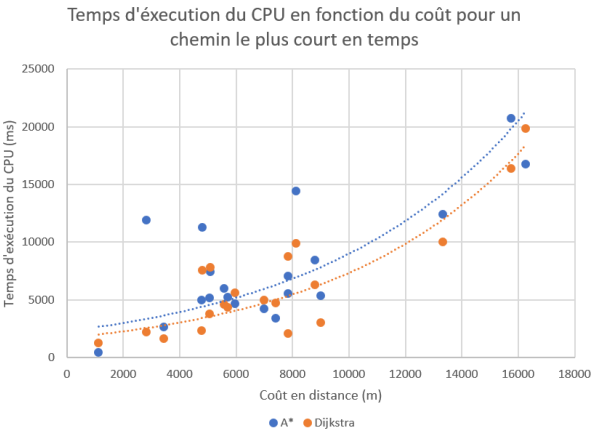
Pour conclure, l'algorithme de A* est plus performant que celui de Dijkstra. Néanmoins, l'écart de performance diminue avec l'augmentation de la distance d'un plus court chemin (distance ou temps). Les analyses précédentes nous permettent d'affirmer que l'utilisation de notre implémentation de l'algorithme de A* est plus intéressante dans le cas de recherche d'un plus court chemin en mode distance. Ceci est expliqué par le fait que A* est basé sur une estimation de longueur à vol d'oiseau, qui est bien moins précise lorsqu'elle est convertie en temps (cf explication 1.2.2).

3.2.2 Temps d'exécution du CPU

Résultats inattendus

Dans un premier temps, nous avons calculé les temps d'exécution des algorithmes de manière très simple : récupération du temps en millisecondes (*System.currentMillis()*) juste avant et après avoir lancé l'algorithme, puis nous avons fait la différence entre les deux valeurs. Les résultats obtenus figurent ci-dessous.



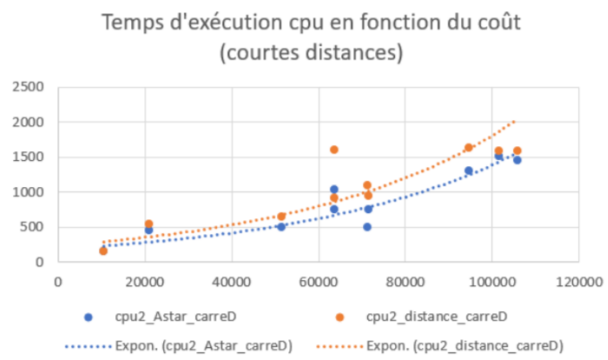


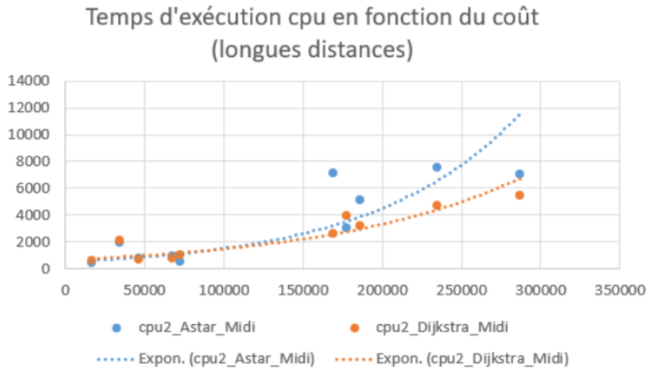
Nous ne nous attendions pas à ces résultats étant donné que l'algorithme A* nous semblait être plus rapide et qu'il visite beaucoup moins de sommets que Dijkstra. Nous avons donc cherché à comprendre à quoi étaient dus ces résultats, la réponse figure ci-après.

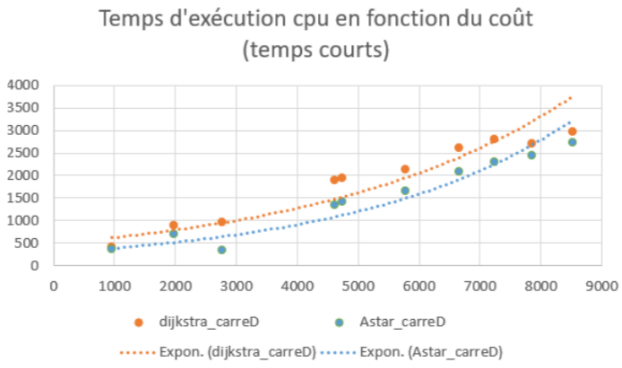
Amélioration proposée et tests corrigés

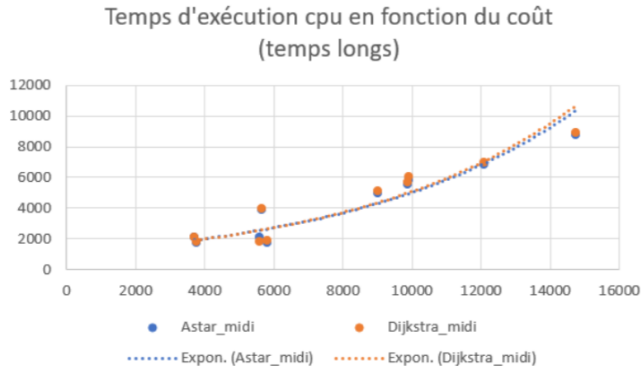
Après plusieurs tests, nous avons compris ce qui ralentissait notre algorithme A* : la création des labels et en particulier celle des LabelStar. En effet, lors de l'initialisation des algorithmes, nous créons une liste de Labels en l'instanciant avec tous les Labels correspondant aux sommets d'un graphe. Autrement dit, nous créons des Labels pour tous les sommets.

Les graphiques ci-dessous montrent que le calcul du plus court chemin (sans prendre en compte la création des Labels) est plus rapide avec A* qu'avec Dikjstra, surtout pour des chemins de courtes distances. Cependant, pour des chemins assez longs (au delà de 150 km) Dijkstra devient plus rapide que A*. Nous supposons que cela est dû à l'augmentation de la taille maximale du tas observée précédemment (celle de A* est supérieure à celle de Dijkstra).









Une solution pour réduire le temps de calculs total de notre algorithme serait non pas de créer tous les labels au début mais seulement lors de la visite des sommets. Cela permettrait d'économiser le nombre de labels créés puisque l'algorithme A* ne parcourt pas beaucoup de sommets comparés à Dijkstra.

Remarque : nous n'avons pas eu le temps d'implémenter cette correction.

4 Réflexion sur le problème ouvert : problème de covoiturage

4.1 Le problème

Deux automobilistes U1 et U2 ont chacun leur origine respective (O1 et O2) et souhaitent atteindre une même destination D.

L'objectif est de proposer à ces deux usagers un trajet de covoiturage leur permettant d'arriver le plus tôt possible à destination. Les deux usagers se rencontrent en un point à déterminer (R), parquent l'un des deux véhicules, et continuent le trajet ensemble.On cherche à minimiser la somme des temps de trajet des deux véhicules.

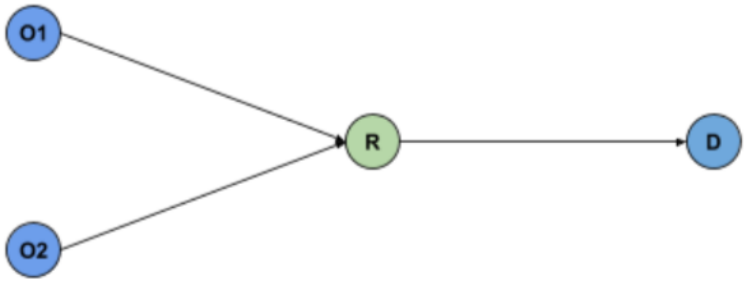


FIGURE 4.1 – Modélisation du problème de covoiturage

4.2 La solution proposée

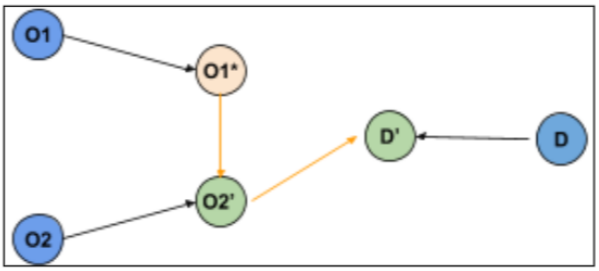
Une solution serait de lancer trois algorithmes de Dijkstra :

1. Le premier de O1 vers tous les sommets
2. Le second de O2 vers tous les sommets
3. Le dernier de D vers tous les sommets

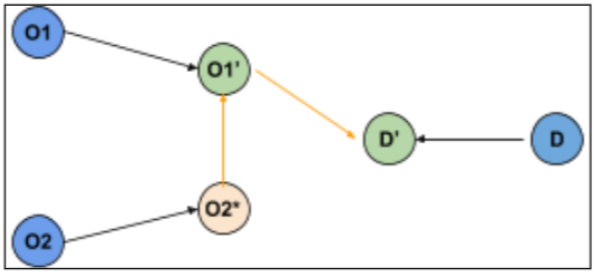
A chaque itération, il faudrait calculer un coût qui serait la somme des coûts des chemins de O1 vers R, de O2 vers R et de R vers D (C= O1->R + O2->R + R->D), et sélectionner l'algorithme qui a le plus petit coût. Pour cela, on peut utiliser un label et mettre le coût à jour à chaque itération.

La condition d'arrêt de l'algorithme serait qu'un sommet soit marqué dans deux algorithmes différents. Ensuite il faudra laisser le troisième algorithme rejoindre ce sommet.

Remarque : dans le calcul du coût, on devra utiliser les coûts à vol d'oiseau (en orange et en rouge ci-dessous).



Calcul du coût C pour l'algorithme 1
(Ici, R correspond à O2')



Calcul du coût C pour l'algorithme 2
(Ici, R correspond à O1')

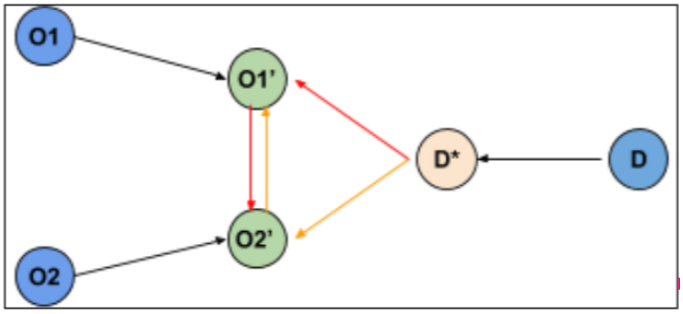


FIGURE 4.2 – Calcul du coût C pour l'algorithme 3 (minimum entre orange et rouge)

Légende : O1 et O2 sont les origines et D la destination. Les points intermédiaires notés *X'* correspondent à l'itération k-1 et les points *X** à l'itération k.

Remarque : dans le calcul du coût, on pourrait ajouter le temps d'attente. En effet, O1->R et O2->R ne mettront très probablement pas autant de temps (en fonction de la limitation de vitesse, du type d'usager etc.) et le premier usager arrivé devra donc attendre le second usager.

4.3 Généralisation du problème

Si on a deux origines et deux destinations :
— il faut lancer 5 fois Dijkstra au lieu de 3 (on fait une symétrie de ce qu'on a fait précédemment).

Si on a n origines et une destination :
— soit faire un seul point de rencontre, donc $n+1$ Dijkstra si on généralise la méthode ;
— soit faire un point de rencontre pour chaque origine. Dans ce cas on effectuera n fois la méthode précédente : une première fois pour le triplet O1, O2 et D puis une seconde fois pour R (le point de rencontre entre O1 et O2), O3, D etc.

Si on a une origine et n destination :
— il s'agit du même problème que n origines et une destination.

Si on a n origine et m destination :
— on va effectuer $n+m$ fois la première méthode : n fois pour trouver les points de rencontre de départ et m fois pour trouver les points de rencontre de destination.

Remarque : faire un seul point de rencontre est utile quand deux usagers sont proches. En revanche, ce n'est pas du tout performant lorsque deux usagers sont éloignés ou qu'il y a beaucoup d'usagers (certains proches, certains loins). Le mieux serait donc de faire un mélange : en fonction de la distance entre deux usagers, faire un seul ou plusieurs point(s) de rencontre.

Conclusion

Ce Bureau d'Etude Graphes a été l'occasion pour nous de mieux appréhender les problématiques liées à la théorie des graphes et d'appliquer les connaissances acquises sur des sujets concrets.

Le fait de travailler sur un projet dont une partie du code est déjà fournie nous a initié au monde professionnel. Il a fallu relever les informations pertinentes et faire abstraction des autres afin d'optimiser le temps de travail.

Ce projet nous a permis de développer nos compétences en Java, notamment en implémentant la problématique de recherche de plus court chemin sous forme de classes orientées objet. Nous avons également appris à réaliser des tests JUnit et donc à réfléchir à toutes les situations problématiques pour les algorithmes. La partie ludique de cette étude est d'avoir eu la possibilité d'observer les comportements de nos algorithmes via une interface graphique, afin d'analyser leurs forces et faiblesses.

Enfin, grâce au problème ouvert nous avons pu réfléchir à une adaptation de nos algorithmes pour résoudre une problématique de la vie courante : le covoiturage. Lors de cette réflexion, nous avons soumis une vision d'ensemble afin de généraliser les algorithmes, ce qui est important pour le futur.