Rapport de Bureau d'Etude

Graphes

Milesi Laurie
Toutant Martin

3 MIC C

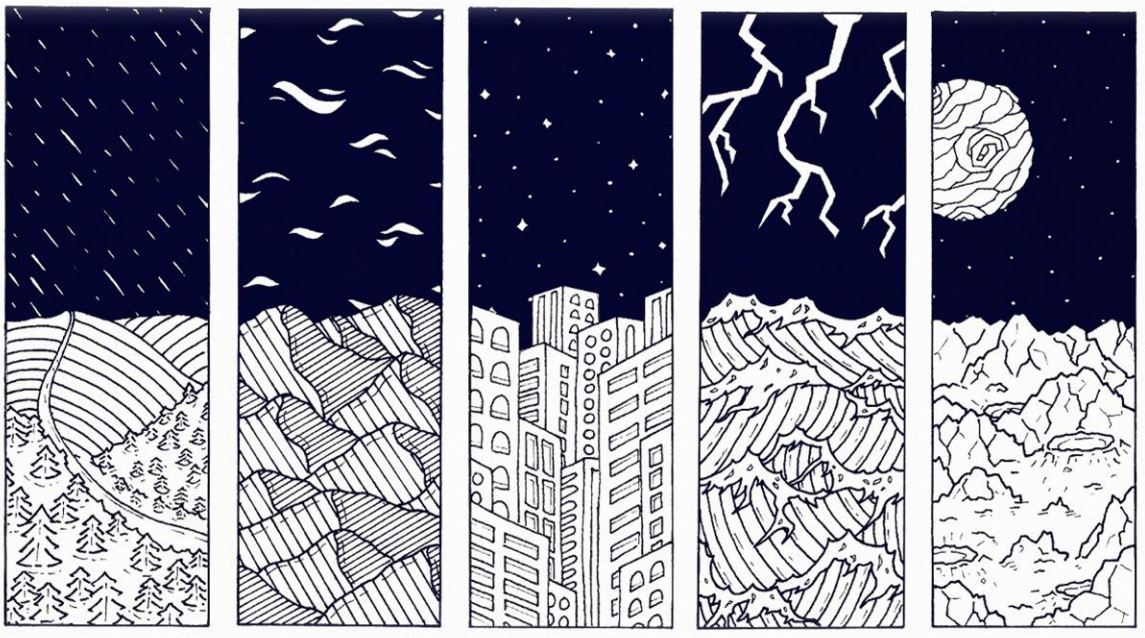


Table des matières

Introduction

L'objectif de ce bureau d'études est, à l'aide de nos connaissances en théorie des graphes et en programmation Java, de réaliser un algorithme de recherche de plus court chemin sous différentes contraintes. Pour se faire, nous avions quelques algorithmes comme celui de Bellman-Ford qui était déjà codés. Cela nous permettait au fur et à mesure de notre travail de comparer nos résultats aux résultats de ce dernier. Nous avons notamment codé les algorithmes de Dijkstra et A* (nouvelle version améliorée de Dijkstra).

Avec ce rapport, nous tentons de dépeindre l'exactitude de nos algorithmes, testés avec deux campagnes de tests ; les tests de validité et les tests de performance. Comme expliqué précédemment, nous nous servirons de l'algorithme de Bellman-Ford.

Dans une première partie, nous présenterons les préliminaires à l'implémentation des algorithmes de Dijkstra et A* que nous décrirons également. Ensuite, dans une seconde partie, nous nous occuperons des tests de validité sur ces deux algorithmes en distance et en temps puis nous ferons de même pour les tests de performance dans une troisième partie. Enfin, nous clôturerons notre rapport par la présentation et l'ébauche de réflexion sur un problème ouvert ; celui de covoiturage.

Voici le lien de notre dépôt GIT : https://github.com/Marteeh/BE_GRAPHE

I. Implémentation des algorithmes de plus courts chemins

A. Implémentation des classes Path et BinaryHeap

La classe Path :

Afin d'implémenter nos algorithmes de recherche de plus court chemin, nous avons dû auparavant compléter les classes Path et BinaryHeap.

Voici la liste des modifications apportées à la classe Path :

- **createFastestPathFromNodes ()** : à partir d'une liste de nœuds, on crée un nouveau chemin qui prend le moins de temps possible. On parcourt la liste des arcs à partir du nœud d'origine, on choisit celui qui mène le plus rapidement au nœud suivant et à chaque fois que l'on choisit un arc, on met à jour le temps de trajet. En cas de non connexion de deux nœuds consécutifs, on lève une exception : IllegalArgumentException ().

- **createShortestPathFromNodes () :** à partir d'une liste de nœuds, on crée le chemin le plus court. Cette fois-ci, c'est une variable tailleMAX que l'on met à jour à chaque fois qu'on ajoute un arc au chemin final. La même exception est levée si deux nœuds consécutifs ne sont pas reliés.

- **isValid () :** cette méthode a pour fonction de vérifier si le chemin fourni en argument est valide ou non. Un chemin est considéré valide s'il est vide, s'il ne contient qu'un nœud, et si le premier arc part du premier nœud et que pour deux arc consécutifs, l'arrivée de l'un est le départ de l'autre. On veille donc à traiter ces trois cas dans la méthode.

- **getLength() :** getter pour la longueur d'un chemin (on ajoute les longueurs de tous les arcs qui composent le chemin).
- **getTravelTime () :** getter qui de la même manière retourne le temps de trajet.
- **getMinimumTravelTime() :** getter pour le temps minimal requis pour traverser le chemin.

Tous les tests junit de la classe Path passent.

La classe BinaryHeap :

Un tas binaire est une structure de données de type arbre qui permet d'implémenter une file de priorité. On peut retirer l'élément de priorité maximale d'un ensemble ou lui ajouter un élément. Ici, on implémente le tas dont nous aurons besoin dans Dijkstra dans la classe BinaryHeap.

Voici la liste des modifications apportées à la classe BinaryHeap :

- **remove () :** cette méthode permet de retirer un élément quelconque du tas. Dans le cas où le tas est vide ou que l'index est plus grand que la taille supposée du tas, on commence par lever une exception. En effet, si l'index est plus grand, l'algorithme risque de marcher quand même car on ne supprime pas vraiment les cases du tableau ; on lui demande plutôt de ne plus les lire. Sinon, on cherche l'élément x dans l'arbre.

On lève encore l'exception si l'index de x n'existe pas dans notre structure. Sinon, dans le cas où c'est le premier élément, on utilise *deleteMin(),* méthode déjà existante. Si c'est le dernier élément, c'est encore facile de le supprimer du tas, mais si l'élément est à une position "lambda" les choses se compliquent. Il faut inter changer la place de l'élément x avec le dernier élément du tas. Ensuite on peut utiliser la méthode *remove()* d'un array pour supprimer le dernier élément du tableau (qui est désormais x). Ensuite, il faut réorganiser la file de priorité. On vérifie si le nouvel élément à la place qui était anciennement celle de l'élément x est plus petit que son père et plus grand que son fils. Si ce n'est pas le cas, on utilise *percolateDown()* ou *percolateUp()*.

B. Création des classes Label et LabelStar

La classe Label :

Nous avons dû implémenter une nouvelle classe Label contenant le sommet courant où l'on se trouve (avec un numéro de 0 à n-1), une marque qui indique si on est déjà passé par ce sommet, un coût qui représente la valeur courante du plus court chemin depuis l'origine vers ce sommet et un père, le sommet précédent sur le chemin. On implémente aussi des getters et des setters pour les attributs de la classe.

Nous nous sommes également occupé des méthodes *compareTo*() et *equals()*. *CompareTo()* comme l'indique son nom sert à comparer deux nœuds en fonction de leurs coûts ; il retourne 1, 0 ou -1 en résultat à la comparaison. *Equals*() vérifie l'égalité de deux labels en prenant soin de vérifier que l'objet passé en argument est justement bien un label. Cette méthode sera utilisée pour vérifier que deux nœuds ne sont pas en doublon dans la file avec deux coûts différents.

La classe LabelStar :

Pour l'implémentation de la nouvelle version de l'algorithme de plus court chemin, nous avons dû implémenter une nouvelle classe. Cette dernière est chargée de prendre en compte la nouvelle information de coût.

C. Implémentation du Dijkstra

L'algorithme de Djisktra est censé renvoyer un ShortestPathData qui représente comme expliqué plus haut le plus court chemin. On peut résumer son fonctionnement de la manière suivante : on commence par choisir le premier sommet le plus près du sommet de départ. De là, on regarde le poids des arcs qui partent du sommet auquel on est arrivé mais aussi ceux qui partent du sommet de départ (donc en somme, on regarde les voisins de tous les nœuds sur lesquels on est déjà passé) et on choisit celui qui a le coût le plus faible. On a donc une nouvelle liste de voisins de tous les nœuds qu'on a visités ; on choisit le voisin avec le plus faible poids encore une fois. Quand la file de priorité c'est à dire la file contenant les nœuds à visité est vide ou que la destination est atteinte, l'algorithme peut s'arrêter.

On a besoin des structures suivantes pour le fonctionnement de l'algorithme :

- Une hashmap qui permette d'associer un noeud à un label (nouvelle classe qui nous permet d'avoir accès au coût, au prédécesseur et de savoir si le sommet a été visité ou non)
- Un tas : une BinaryHeap

Il nous faut une liste qui contienne les nœuds possibles où l'on peut se rendre ; c'est à dire la liste des voisins de tous les nœuds déjà visité. À chaque étape il faut supprimer de cette liste de possibilité le nœud sur lequel on choisit de se rendre et il faut ajouter ses voisins.
L'algorithme s'arrête quand notre filePriorite est vide ou que le sommet « destination » est atteint.

La première étape de l'algorithme consiste à ajouter à la Hasmap et à la file de priorité le sommet de départ avec un coût nul et une marque de visite. Ensuite, on vérifie que l'origine soit différente de la destination (avec la méthode compareTo() de la classe label). Si ce n'est pas le cas, on fait en sorte de renvoyer un chemin non valide.
Si on est dans le cas où les deux sommets sont bien différents, on peut marquer l'origine comme visitée (avec les observers).
On peut alors ajouter la destination dans la file de priorité et la hasmap.

Notre boolean "*destReached*" sert à vérifier que la destination est atteinte ou non.

On peut désormais entrer dans une boucle dont on ne sortira que quand la file de priorité sera vide ou que la destination sera atteinte. Grâce à la fonction *deleteMin(),* on trouve l'arc au coût le plus faible ; on le supprime de la file de priorité. On compare ensuite l'élément qu'on vient de recevoir à la destination pour s'assurer qu'ils ne sont pas égaux. Si ce n'est pas le cas ; on parcourt la liste d'arcs menant aux successeurs du sommet où l'on est, on les ajoute à la hasmap et à la file de priorité s'ils n'y sont pas déjà.
Dans le cas où l'un d'eux y est déjà, on vérifie le coût qui lui est associé : s'il est supérieur à la somme du coût du sommet actuel et celui de l'arc on doit mettre le coût et le prédécesseur de ce sommet à jour (avec la somme justement).
Lorsque tous les arcs partant d'un sommet ont été étudiés, le sommet est marqué comme visité.

À l'issue de la boucle on a trouvé notre plus court chemin.

Les observers fonctionnent de la manière suivante ; au fil du code, nous leur envoyons des signaux. À la réception de ces signaux, ils sont chargé d'effectuer l'action qui leur incombe. Dans nos algorithmes, nous nous en servons pour notifier du fait que l'origine est marquée, qu'un nœud quelconque est marqué ou que nous avons atteint l'arrivée. Nous avons décidé de faire apparaître des messages de bon déroulement quand les notifications des observateurs sont envoyées.

D. Implémentation de A*

L'algorithme A* est une extension de Dijkstra où les sommets ne sont plus seulement ordonnés en fonction de leur coût par rapport à l'origine mais en fonction de deux coûts, le coût par rapport à l'origine et le coût estimé à la destination (basé sur un trajet à vol d'oiseau entre les deux sommets). Dans ce cas le déroulement de l'algorithme de plus court chemin est guidé par la destination du trajet.

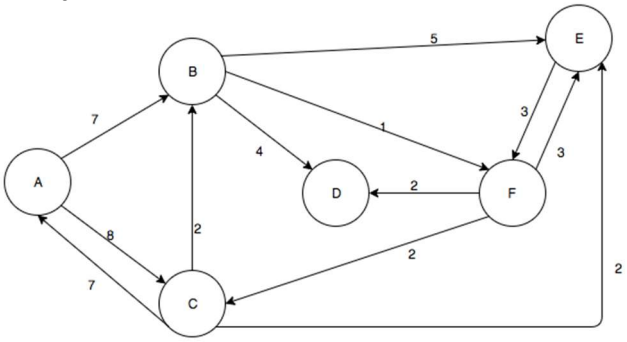
A* doit donner les mêmes résultats que Dijkstra.

II. Tests de validité

A. Tests avec oracle

Nous cherchons désormais à valider notre implémentation. Les tests vont nous permettre d'évaluer l'algorithme à travers les caractéristiques des solutions obtenues. On vérifie qu'elles sont valides et optimales (en comparant d'abord à un oracle puis sans).

Pour s'assurer de la validité, on se sert d'une carte simple : fractal spirale, et d'une autre un peu plus complexe mais de taille acceptable : celle de la Guyane. Ainsi, on évite que nos phases de tests soient trop longues (on utilisera des cartes plus volumineuses pour comparer les performances).
On trouve tout d'abord dans le fichier de test pour Dijkstra un test sur un graphe simple que nous avons créé nous même et visible sur la figure ci-après (ce test s'apelle ShortestPathTestGrapheTest). Ce test basique nous a surtout servi pour débugger et vérifier le fonctionnement de notre Dijkstra.



Pour les tests, on s'intéresse d'abord à des cas particuliers : origine égale à la destination CheminDistanceNull(), et destination inatteignable depuis l'origine OrigineNoFils() (pas de chemin valide). Puis on teste avec l'oracle si les plus courts chemins, trouvés par Dijkstra et A-Star, sont optimaux. L'oracle désigne le résultat de BellmanFord pour les chemins qu'on va tester. On considère que cet algorithme fournit une solution optimale pour le problème de plus court chemin, il s'agit de notre référence. On utilise les mêmes tests sur les algorithmes de Djikstra et A*, à la fois en distance (TestDistance()) et en Temps (TestTemps()). De plus, on peut appliquer à tous nos tests des contraintes grâce aux ArcInspectors :

On initialise un GraphReader, qui nous permet d'utiliser des variables pour stocker nos 2 cartes de tests :

```
GraphReader reader;

//Scenario map fractal spirale
String mapName = "/home/toutant/Bureau/Cours/3MIC/S2/BE-Graphe/Maps/fractal-spiral.mapgr";
reader = new BinaryGraphReader(new DataInputStream(new BufferedInputStream(new FileInputStream(mapName))));
fractoul = reader.read();

//Scenario map guyane
mapName = "/home/toutant/Bureau/Cours/3MIC/S2/BE-Graphe/Maps/guyane.mapgr";
reader = new BinaryGraphReader(new DataInputStream(new BufferedInputStream(new FileInputStream(mapName))));
guyane = reader.read();
```

Puis extraction des ArcsInspectors pour les contraintes de test :

```
lenghtallAllowed =  ArcInspectorFactory.getAllFilters().get(0);
lenghtCarRoadOnly = ArcInspectorFactory.getAllFilters().get(1);
timeAllAllowed = ArcInspectorFactory.getAllFilters().get(2);
timeCarRoadOnly = ArcInspectorFactory.getAllFilters().get(3);
timePedestrianRoad = ArcInspectorFactory.getAllFilters().get(4);
```

On s'interessera pour nos tests aux filtres time/lengthAllAllowed (toutes les routes sont bonnes), time/lengthCarRoadOnly (seulement les routes pour les voitures) et time/lengthPedestrianRoad (les routes accessibles aux piétons).

Lors du code, nous avons gardé un esprit de généricité pour pouvoir effectuer de multiples tests en utilisant un minimum de fonctions. On code ces fonctions de tests de telle sorte qu'un test soit utilisable pour plusieurs cartes, et plusieurs filtres.

Ainsi avec 6 fonctions principales de test : TestTemps, TestDistance, OrigineNoFils, CheminDistanceNul, TestValiditeDistance et TestValiditeTemps (ces deux dernières utilisant 3 sous-fonctions pour alléger le code), on fait passer 17 tests à nos algorithmes.

Il nous semble important de noter que la fonction TestTemps pour la carte guyanne.mapgr, sur les routes piétonnes semble ne jamais se valider, et nous l'avons donc laissé de coté pour se focaliser sur les tests sans Oracle.

Une fois ces tests passés, on considère que le fonctionnement de nos deux algorithmes est correct et on se propose de vérifier visuellement leur fonctionnement.

B. Tests sans oracle

Parfois, il arrive qu'on ne dispose pas de la valeur de la solution optimale. On doit donc appliquer d'autres tests pour vérifier les solutions.
Pour cela, on s'appuie sur une propriété du plus court chemin : si on prend une partition du plus court chemin trouvé, alors le sous-chemin extrait est un plus court chemin.

Pour cela, on exécute D ou A et récupérons le path dans une variable de type ShortestPathSolution.
Ensuite, on utilise une fonction choisirDest qui prends un point environ à mi-chemin du parcours et calcule le plus court chemin entre l'origine et ce point. On teste ensuite si le chemin recalculé correspond au morceau de *path* calculé plus tôt (en distance et en temps) et s'ils correspondent, on affirme que l'algorithme passe le test. [1]

[1] *Note aux correcteurs : les chemins spécifiés sont des chemins "bruts" et non relatifs par rapport au dossier où on lance le programme. Pour que les programmes de test fonctionnent bien sur une autre session que celle de Martin, il faut changer les "paths" aux lignes 49 et 54 de AStarAlgorithmTest.java, 95 et 100 de DijkstraAlgorithmTest.java et la ligne 45 de PerformanceShortestPathTest.java*

III. Tests de performance

A. Fonctionnement des tests

Pour les tests de performances, on utilise des cartes de plus grand volume en variant la disposition des routes.

On se base sur la conception des JUnit.
Tout d'abord, les tests sont initialisés avec une fonction AllInit() qui initialise 4 variables : la carte grâce à un GraphReader, le nombre de tests à effectuer, une borne supérieur et une borne inférieur des chemins à générer. Afin d'être surs que cette fonction s'exécute en premier on utilise la mention @BeforeClass.
De plus, on utilise un PrintWriter, qui nous permet de générer un fichier et d'écrire dedans. On l'utilise pour stocker les données de nos tests sous le format .csv. En utilisant un séparateur bien choisi, on pourra exploiter ces données sur l'outil Excel, de la suite Office de Microsoft.

Le test appelle ensuite la fonction principal FaireTest : on génère les points de départ et d'arrivée au hasard sur la carte, puis on exécute les algorithmes de Dijkstra et A-Star avec ces deux points. Afin d'améliorer le temps d'exécution du test, on lance d'abord Dijkstra et on teste si le chemin est réalisable et sa longueur est comprise entre les deux bornes demandées, puis A-Star est appelé seulement si l'exécution de Dijkstra a réussit. A chaque test on rajoute une ligne dans le fichier csv pour avoir le résultat suivant (ouverture du fichier dans Excel en précisant que le séparateur est « ; ») :

Algo	ID origine	ID destinatic	distance_vol	distance_Alg	Nombre noe	temps(ms)	nombreSommetVisite
Dijkstra	4984201	111768	74900,5118	87014,2	678	1241,76846	976476
AStar	4984201	111768	74900,5118	87014,2	678	181,666742	52192
Dijkstra	2742397	3253860	82269,0979	97914,9	925	2646,59009	403788
AStar	2742397	3253860	82269,0979	97914,9	925	297,928332	153945
Dijkstra	1316790	5376129	110637,592	122081,234	752	3049,00791	805725
AStar	1316790	5376129	110637,592	122081,234	752	258,664233	138087

Les deux fonctions pour les calculs des algorithmes suivent le même schéma :

- Initialisation des structures nécessaires au calcul & utilisation des variables début et fin de type double pour calculer le temps d'exécution : fonction System.nanoTime() ;

```
ShortestPathData data = new ShortestPathData(carte, origine, dest, ArcInspectorFactory.getAllFilters().get(0));
ShortestPathAlgorithm dijkstraAlgo = new DijkstraAlgorithm(data);

debut = System.nanoTime();
ShortestPathSolution solution = dijkstraAlgo.doRun();
fin = System.nanoTime();

double time=((double)(fin - debut)/1000000.0);
```

- Ecriture des données dans le fichier .csv quand le chemin respecte les conditions. On utilise un StringBuilder en tant que buffer pour le PrintWriter précédemment instancié dans InitAll().

```
if ((solution.isFeasible()) && (solution.getPath().getLength() < (float) borneSup*1000.0) && (solution.getPath().getLength() > (float) borneInf*1000.0)){

    System.out.println(time);
    int size = solution.getPath().size()+1;

    ecriture.append(solution.getPath().getLength() + separateur + size + separateur + time + separateur + solution.getnombreSommetVisite());
    //on remplace tous les points par des virgules pour le traitement excel
    writer.println(ecriture.toString().replaceAll("\\.",","));
    System.out.println(ecriture.toString().replaceAll("\\.",","));
```

Cette procédure est répétée autant de fois qu'il y a de tests demandés.

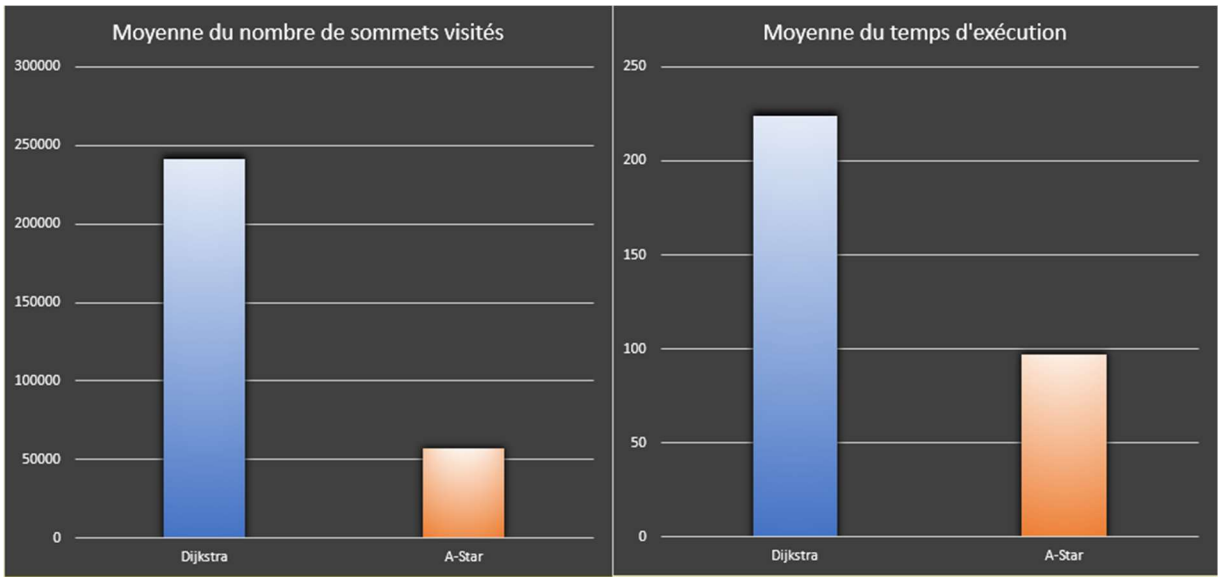
B. Analyse des résultats.

On envisage trois cartes relativement différentes pour nos tests : languedoc-roussillon.mapgr, denmark.mapgr et british-isles.mapgr. Pour chaque carte, on génère des données pour une série de chemins courts et une de chemins longs. On prend un grand nombre d'échantillons : 100, pour avoir des résultats concluants. Les résultats que nous présentons par la suite sont des moyennes que nous calculons après avoir exporté le fichier .csv sur Microsoft Excel.

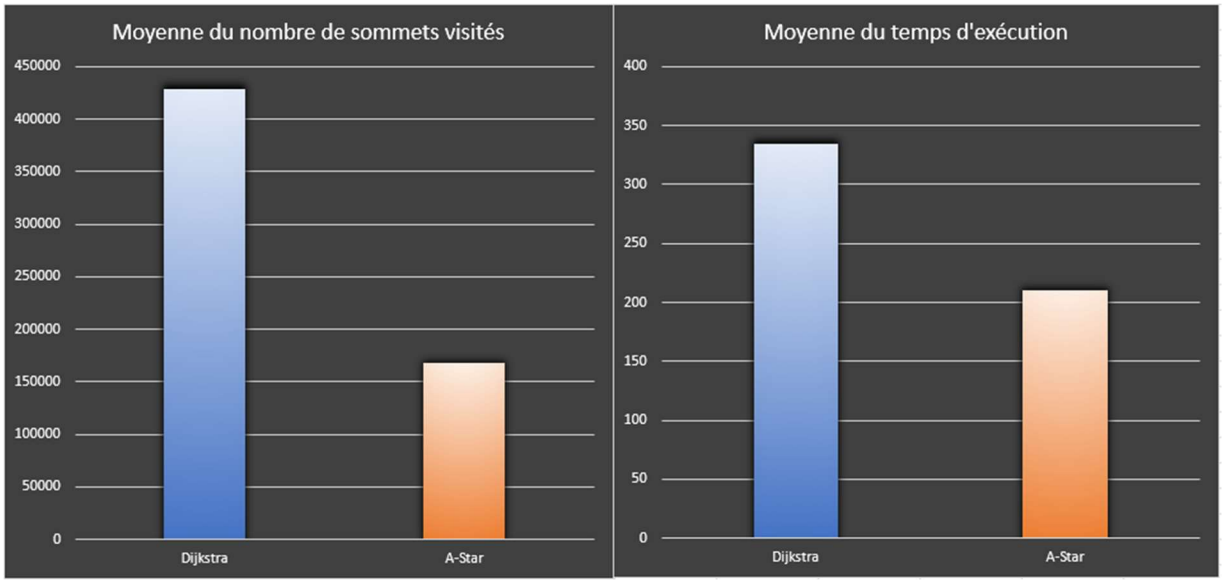
<u>Languedoc – Roussillon</u>

On remarque une différence notable d'efficacité entre les deux algorithmes : A-Star est plus de deux fois plus rapide que l'algorithme de Dijsktra classique, en considérant presque 5 fois moins de sommets. Ces résultats sont cohérents avec le principe de fonctionnement de A-Star qui oriente sa recherche vers le sommet de destination. Il permet d'économiser du temps et de la mémoire sachant qu'on stocke ces sommets dans une hashmap.

Pour des chemins courts :



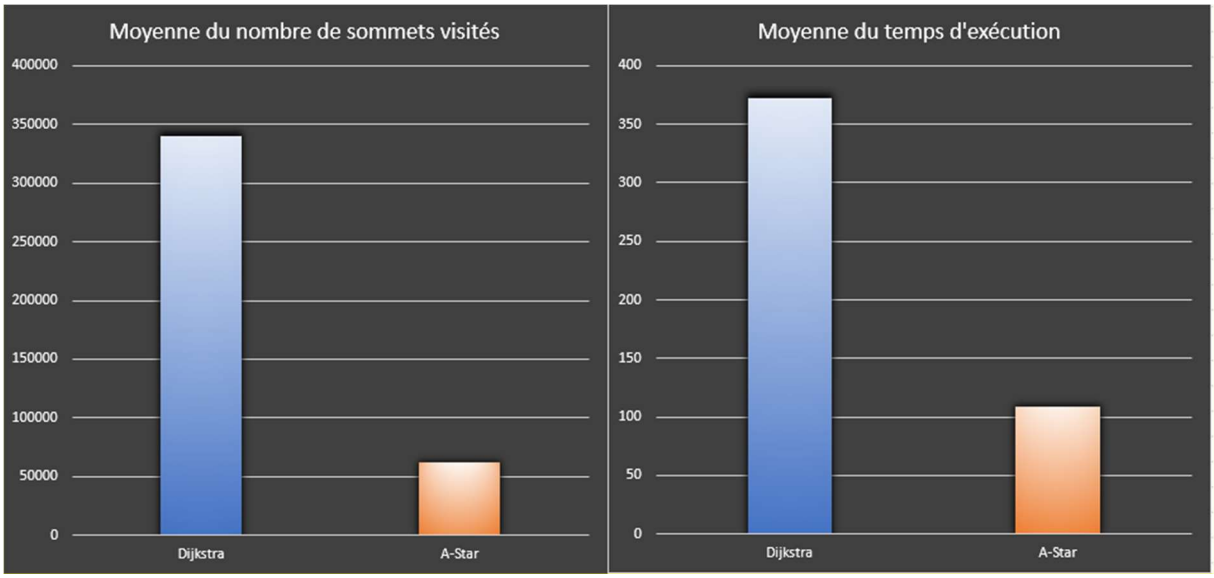
Pour des chemins longs :



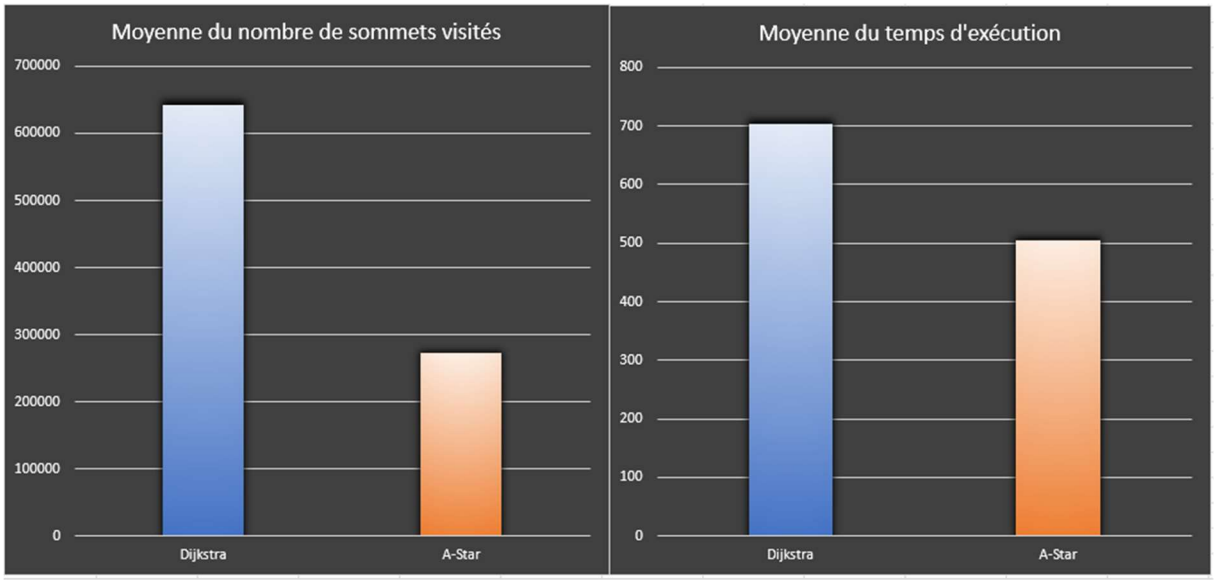
Danemark et Royaume – Uni

Pour les petites distances, on remarque généralement les mêmes résultats. Cependant pour des chemins plus longs, on remarque que les temps d'exécution se rapprochent. Ces cartes ont en effet des contraintes liées à l'eau : des fois le chemin "direct" n'est pas possible et il faut contourner le point d'eau. Dans ce cas là, A-Star va pouvoir contourner mais en cherchant toujours à aller tout droit, ce qui va augmenter le temps d'exécution et celui-ci va se rapprocher du temps d'exécution de Dijkstra.

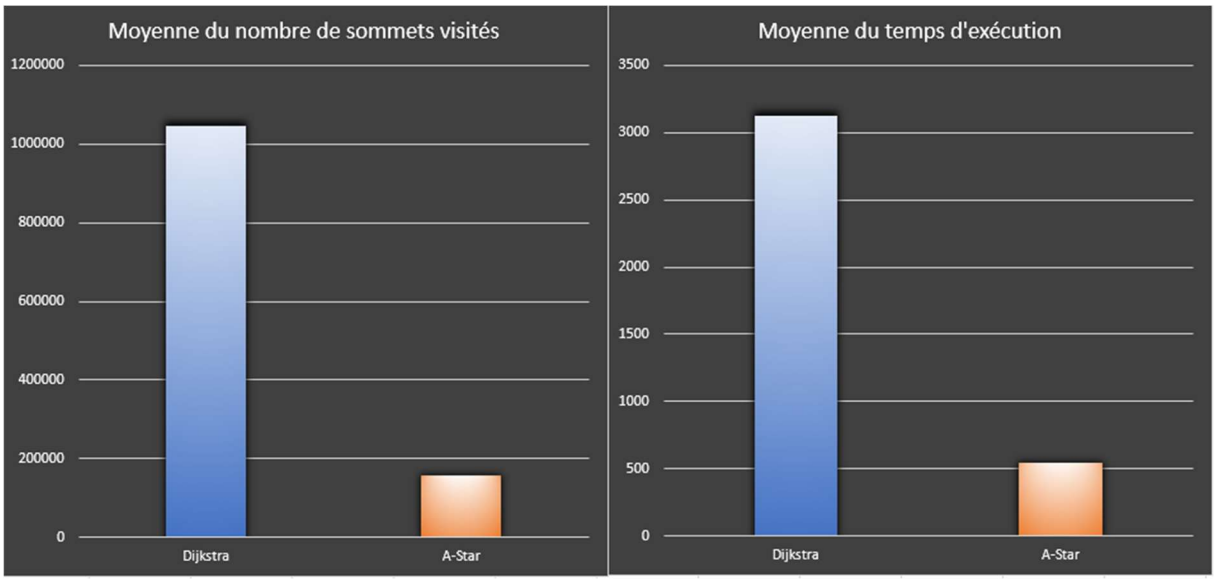
Danemark pour des chemins courts :



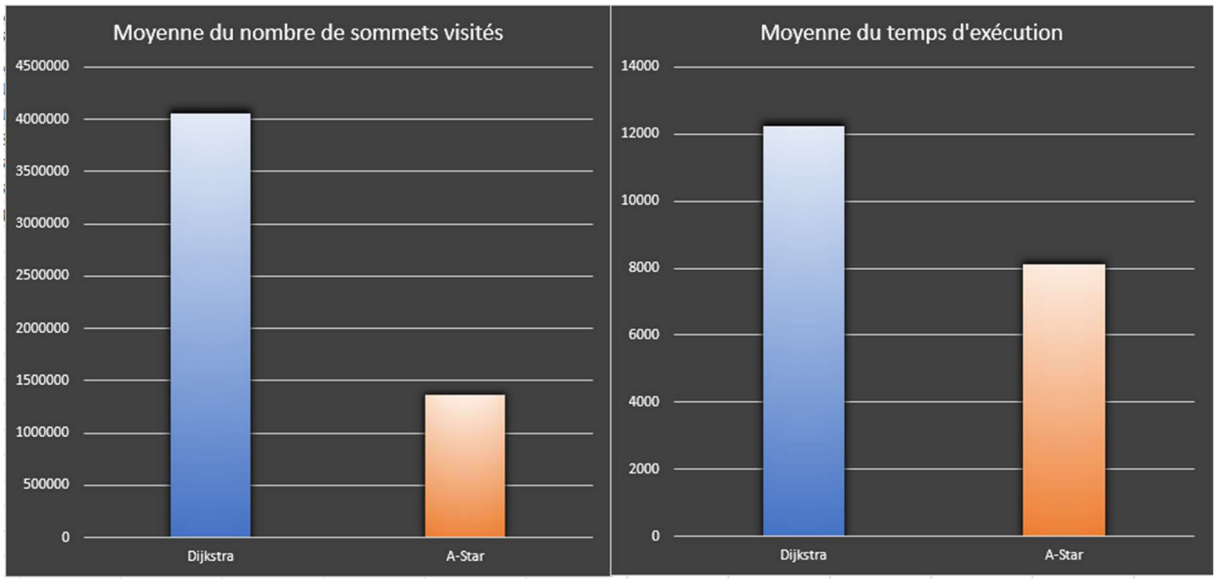
Danemark pour des chemins longs :



Royaume-Uni pour des chemins courts :



Royaume-Uni pour des chemins longs :



IV. Problème ouvert

Nous avons choisi le problème de covoiturage.

Le problème à résoudre est le suivant : à partir de deux origines distinctes, O1 et O2, deux usages U1 et U2 cherchent à rejoindre une même destination D en un temps minimal. Pour se faire, ils doivent se rencontrer en un point C (pour covoiturage) et continuer ensemble vers D. On cherche alors à minimiser la somme des temps de trajet des deux véhicules, que ce soit avec ou sans covoiturage (solution non bla-bla). C'est à dire que si la somme des temps mis par les deux automobilistes seuls depuis leurs origines respectives vers la destination est inférieure au coût de n'importe quel chemin avec du covoiturage, le chemin non bla-bla est à privilégier.
On ne peut prendre d'autoroute s'il y a moins de deux personnes à bord de la voiture.
Cela signifie qu'on cherche à minimiser [O1C]+[O2C]+[CD] qui doit rester inférieur à [O1D]+[O2D].

On propose comme solution de lancer 3 algorithmes de Dijkstra. Les deux premiers partiraient des deux origines O1 et O2 vers le sommet de destination D. Comme on ne peut prendre l'autoroute quand on est seul dans la voiture, il faudrait utiliser pour ces algorithmes des ArcInspector pour les chemins autorisés pour 1 seul voyageur. Le troisième Dijkstra tournerait sur le graphe inverse, de la destination à l'une des origines avec un ArcInspector autorisant les autoroutes.
Il faudrait que les 3 algorithmes complètent des tableaux récapitulatifs avec les sommets communs qu'ils ont visités et le coût de chacun pour y parvenir. Il faudrait ensuite faire la somme de ces 3 coûts afin de sélectionner le sommet avec le coût le plus faible.

Pour être certain qu'on peut utiliser ce chemin, il faut toujours veiller à vérifier la condition d'infériorité au temps mis si les deux automobilistes faisaient le chemin seuls.

On obtient ainsi un chemin optimal en terme de temps.

On ne peut pas utiliser l'algorithme A* pour résoudre ce problème car il ne parcourt pas tous les sommets et il se pourrait alors qu'on ne voit pas un sommet qui aurait pu minimiser le trajet.

Conclusion

À travers ce projet, nous avons pu davantage visualiser le rendu et le fonctionnement des algorithmes de plus courts chemins que nous avions pu voir en Graphes. Niveau code, nous avons eu une application directe de la programmation orientée objet avec l'utilisation de diverses classes pour tout le projet et l'utilisation de java.

Grâce à l'affichage graphique et nos tests de performance, nous avons pu nous rendre compte que l'algorithme de Dijkstra explorait les possibilités de chemin de manière circulaire autour du point de départ, et ce jusqu'au point d'arrivée. L'adaptation A-Star se focalise sur la destination et "guide" ses recherches dans cette direction. De manière générale, on trouvera le plus souvent l'algorithme de A-Star bien plus performant, notamment pour des chemins courts. Cependant, son efficacité se rapproche, voir peut être dépassée, par l'efficacité de Dijkstra, sur une carte qui présente un obstacle naturel majeur (point d'eau tel un lac ou la mer, ou encore une chaine de montagne) qu'il est nécessaire de contourner. Cette différence est d'autant plus notable que les chemins sont longs.

D'éventuelles améliorations pour notre code porteraient sur l'appel des tests. En effet, à chaque fois pour tester une carte différente, on doit changer le nom de la carte dans l'algorithme (pour chacun des algorithmes). On pourrait plutôt invoquer un menu de sélection en se calquant sur le fichier mainWindow.java

Nous aurions également pu, éventuellement, au lieu des deux fichiers de tests pour A* et Dijkstra, essayé de faire une classe de tests abstraite et nous aurions simplement dû instancier des classes avec le nom de l'algorithme à tester pour éviter les répétitions.