Rapport BE Graph

Mise en place

Le lien du git : https://github.com/alexandregonzalvez/BE_graph.git

Version JDK : 1.8 (pom.xml)

Remarque : Le chemin vers le dossier maps dépend de la machine, il faut donc le modifié pour chacun des 5 tests :

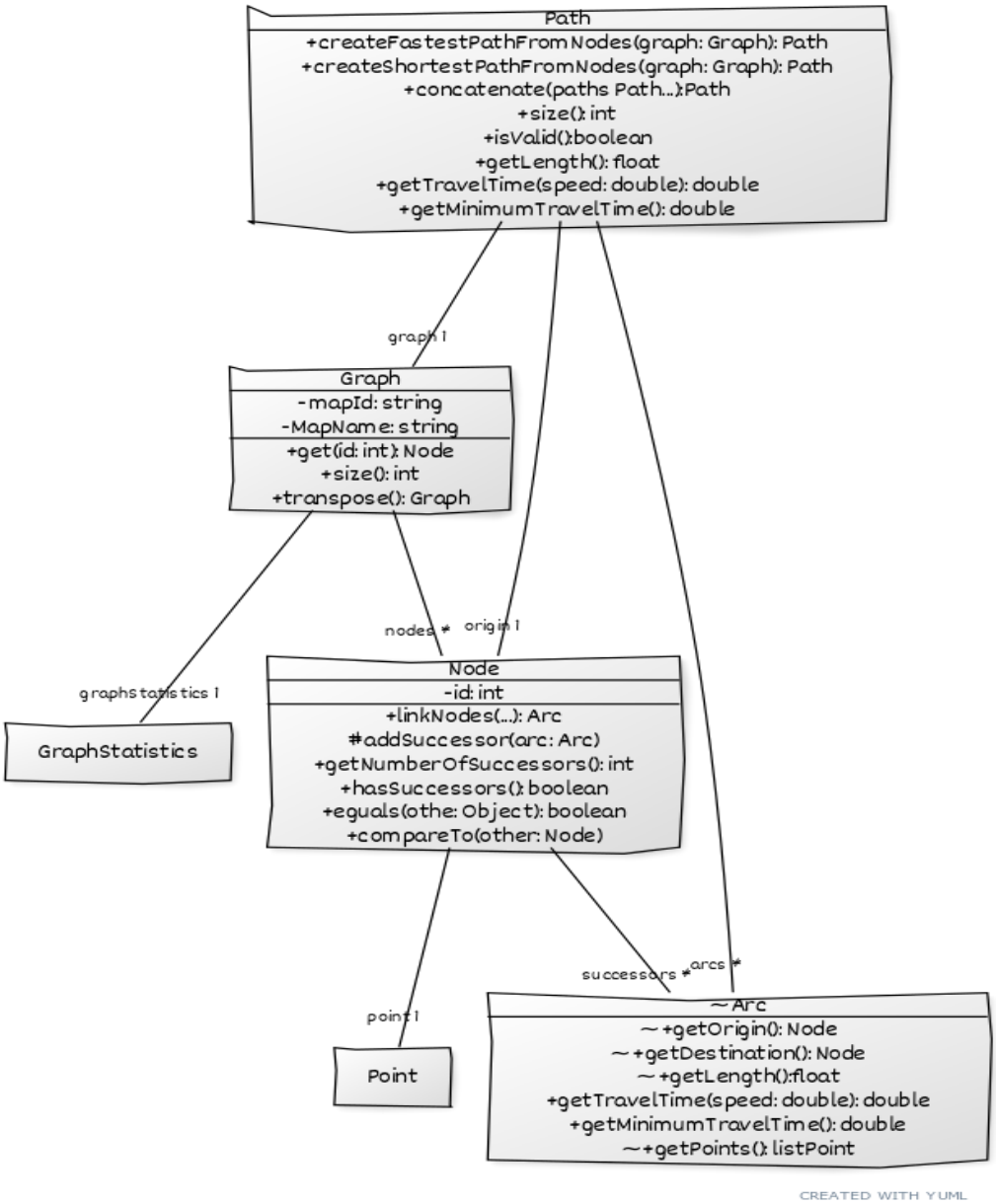
2 occurrences dans chacun les fichiers DijkstraTest.java et DijkstraAStar.java et 1 occurrences dans PerformanceDijkstraAStar.java (variable cheminVersMaps)

Vérification de chemins

Javadoc

On trouve la *javadoc* dans le répertoire DOC du git.

Pour prendre en main l'implémentation de la construction des graphes et chemins, ci-dessous le diagramme UML (synthétique) des classes *Graph*, *Node*, *Path*, *Arcs* et *RoadInformation :*



Les chemins

Pour continuer à se familiarisé, on implémente des méthodes de la classe Path puis on vérifie que tous les tests de JUnit sont validés.

Création d'un programme from Scratch

On créer une alternative l'exécution avec « MainWindow », notamment pour pouvoir effectuer des tests.

On aurait pu créer un programme Graphe sans interface graphique et seulement des jeux de tests, cependant il parait plus agréable et simple de pouvoir visualiser l'exécution en temps réel de son programme. De plus une visualisation est souvent nécessaire dans ce type de projet.

Les types *GraphReader* et *PathReader* sont essentiel car ils nous permettent d'implémenter des fonctions d'interfaçages comme *AutoCloseable* et *Closeable.* Ainsi nos classes *BinaryGraphReader*/BinaryPathReader peuvent hérité de *BinaryReader* et implémenté *GraphReader/PathReader.* (Une classe ne peut pas hérité de deux autres)

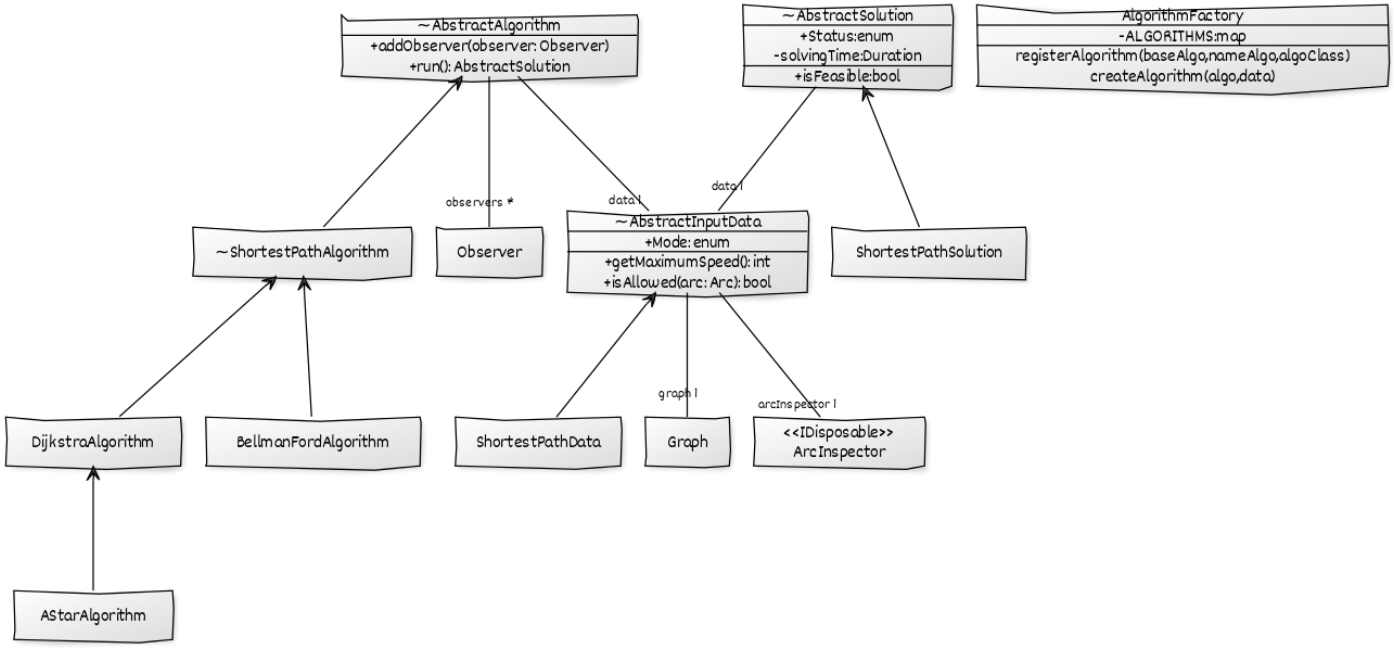
Remarque 1 : Le chemin vers ''3eme Annee MIC'' dans ''Launch.java'' est déclaré de façon volatile (par rapport au git. Donc en cas de clone du git, il faudra rééditer ce chemin si on souhaite l'exécuter ce programme avec Launch.

Remarque 2 : Le fichier Launch.java se trouve dans le répertoire à l'adresse suivante : be_graphes-gui/src/main/java/org/insa/graphs/gui/simple

Plus court chemin

Algorithmes et classes java

Ci-dessous le diagramme de classe des packages algo et algo.shortestpath :



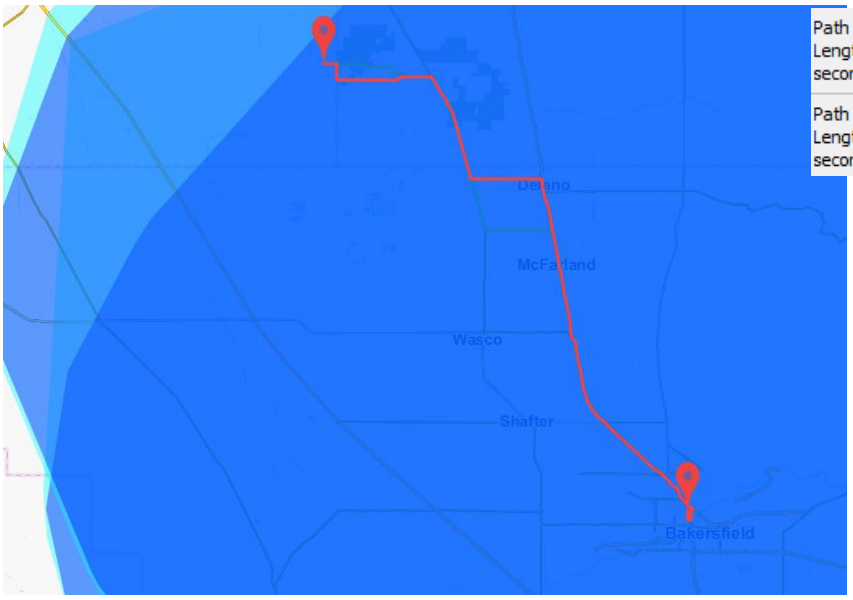
Algorithme de Dijkstra

On commence par créer une classe label dans le package *org.insa.graphs.algorithm.utils*, celle-ci implémentera Comparable afin d'utiliser les opérateurs de comparaison. Pour utiliser *Node* pour le sommet courant il faut également importer *org.insa.graphs.model.Node* .

Remarque : Après certains tests, l'algorithme semble fonctionner avec ''peu'' d'arcs, par exemple en chargeant la carte Californie, avec certains trajets (+600km entre les deux points) contenant beaucoup d'arcs, l'algorithme semble fonctionner au départ puis finit de façon inattendu en choisissant des arcs approximatifs et ne peut donner la distance et le temps. Sans doute que l'espace mémoire alloué n'est pas assez important.

Campagne de tests de correction

I. Vérification visuelle



Sur cet exemple, on voit que les points explorés au fur et à mesure du temps (ajout dans le tas). Il s'agit d'un dijkstra ''normal'' donc il explore dans toutes les directions, l'affichage correspond. On peut voir sur la capture d'écran que pour le calcul de durée, l'algorithme explore moins de sommets.

II. Caractéristiques des solutions obtenues et optimalité de l'algorithme avec un algorithme de référence

Deux classes *JUnit* ont été créée pour effectuer ces tests *DijktraTest* et *solutionDijkstraTest*, la seconde correspond à une fonction testant un cas avec pour référence l'algorithme de Bellman-Ford, par conséquent certains tests peuvent s'avérer long (dû à l'utilisation de cette algorithme). Pour des questions de temps de tests, seulement des zones de taille petite/moyenne ont été testés.

Deux tests généraux sont effectués, le premier effectue différents cas avec pour mode la distance :

- Sommets identique
- Origine ou destination invalide
- 9 cas ''normaux'' d'utilisation
- 1 cas où l'algorithme de Bellman-Ford ne trouve pas de solution

Pour chacun de ces tests, différentes cartes avec différents sommets (origine/destination) sont testés.

Le second effectue ces mêmes tests avec différentes cartes et d'autres sommets.

L'algorithme codé dans l'étape précédente renvoi la même solution que celui de Bellman-Ford dans tous ces cas, donc la solution trouvée par l'algorithme est optimal pour les cas testés.

III. Optimalité de l'algorithme sans algorithme de référence

Sans algorithme de référence comme celui de Bellman-Ford, il est nécessaire de vérifié l'optimalité de la solution de notre algorithme.

La première idée, est de tester cet algorithme dans des cas simple, ou la solution est évidente. On pourrait prendre la carte carre.mapgr pour tester notre algorithme. Cependant, on souhaitera le testé pour des cas plus complexe. Une autre solution serait de calculé tous les chemins possible et de prendre la meilleure solution, le problème étant la complexité d'un tel algorithme. Une autre idée serait de légèrement surestimer la longueur du plus court chemin (ou le temps minimum pour le parcourir) en fonction de la distance entre le point d'origine et celui de destination, de façon arbitraire ou dans un meilleur cas avec un jeu de données. On pourra également vérifier d'autres

paramètres comme que la solution soit inférieure à la distance à vol d'oiseau (inférieur au temps à vitesse maximum pour parcourir cette distance), que la solution en distance pour un chemin entre 2 points soit inférieur ou égale à la solution en en minimisant le temps (et inversement en prenant le temps comme indice).

Dijkstra en version A*

On souhaite modifier légèrement notre implémentation de Dijkstra pour favorisé la recherche dans les chemins allant dans la bonne direction (plus précisément on souhaite ordonné les sommets visité selon leur coût depuis l'origine et l'estimation de leur coût jusqu'à l'arrivée, cette estimation correspond à la vitesse en vol d'oiseau dans notre algorithme).

Pour ceci, une méthode getTotalCost() à été rajouté dans Label et modifié dans LabelStar pour considérer cette estimation jusqu'à l'arrivé.

Note : Après divers tests et modifications, je n'ai pas réussi à trouver l'erreur qui mène à la même solution (en termes de recherche) que l'algorithme de Dijkstra classique. L'algorithme donne donc une solution optimale mais n'apporte aucune amélioration par rapport à celui de Dijkstra.

Campagne de tests de correction

Comme pour l'algorithme de Dijkstra classique on créée 2 class JUnit pour effectuer ces tests *DijktraAStarTest* et *solutionDijkstraAStarTest.* L'algorithme *Dijkstra* classique a été ''validé'' à l'étape précédente donc il est possible de vérifier l'algorithme A* avec ce dernier, par conséquent on remarque que les tests sont effectués bien plus rapidement dans les cas normaux. En ce qui concerne les tests, il s'agit des mêmes tests. L'algorithme *Dijkstra* classique a été vérifié avec ces valeurs, donc nous pouvons être certains qu'il retourne la bonne solution pour ces derniers.

Campagne de tests de performance

Pour effectuer ces tests, une nouvelle classe JUnit a été créé PerformanceDijkstraAStar.java, 4 tests vont être effectué pour chaque test, une carte est choisi avec un mode à choisir (LENGTH ou TIME). Sur chaque test 20 valeurs pour l'origine et la destination vont être ''aléatoirement'' choisi parmi les différents sommets de cette carte.

Comme attendu après les tests effectué après l'implémentation de Dijkstra en version A*, les deux algorithmes sont sensiblement aussi performant, et donc l'implémentation de Dijkstra en version A* n'est pas validé.

Remarque 1 : Sans effectué 4 fois cette manipulation, la probabilité que le test soit validé était trop grande

Remarque 2 : Pour une question de lisibilité un sleep du thread principal de 3 secondes s'effectue entre chacun de ces tests.

Problème ouvert

Problème choisi : Eviter des lieux fréquentés (voiture)

Description du problème : On souhaite se rendre d'un point A à un point B en voiture en évitant les lieux fréquentés, sans que le trajet soit trop long par rapport à la solution optimal.

Pour commencer il faut identifier les lieux potentiellement fréquentés, 3 approches ont été identifiées :

- Avoir des statistiques sur la fréquentation de chacun de ces lieux (par exemple une application comme Waze à la base de données nécessaire).
  =>Meilleur solution, non retenu car données manquantes
- Regarder le nombre de sommets présent autour du sommet visité. Plus ce nombre est grand plus la zone est sensible d'être fréquenté.
- Supposé que plus le nombre d'arc partant d'un sommet plus le sommet est « fréquenté ». (solution proche de la précédente mais plus simple à exécuter)
  => Solution retenu, une amélioration possible serait de regardé également le nombre d'arc des plus proches sommets voisin.

Il faut donc définir un nombre de successeur MAX.

Le deuxième point, consiste à définir une méthode pour que le trajet ne soit pas trop long par rapport au plus court chemin. Et le cas échéant utiliser certains sommets ''fréquentés'', dans l'idéal ceux réduisant le plus possible la durée du trajet.

Solution simple (résultat moyen mais rapide) :
En utilisant un algorithme de Dijkstra A* (complexité en O((nbSommets+nbSuccesseurs)*log(nbSommets)), pour chaque sommet qui a plus de MAX successeurs, on ne le met pas dans le tas. Une fois l'algorithme terminé on regarde si le coût est inférieur à un algorithme de Dijkstra A*(sans cette condition) multiplié par le paramètre de tolérance. Si c'est le cas on choisit cette route. Sinon on prend la route de donnée par le second algorithme.

La complexité de cet algorithme est celle de Dijkstra étant donné que l'on effectue deux fois un algorithme de Dijkstra. En réalité, la première exécution sera sans doute plus rapide car on enlève des sommets et que les sommets restant possèdes moins de successeurs (m est donc plus petit en général).
Complexité = en O((nbSommets+nbSuccesseurs)*log(nbSommets))

Dans le cas où le premier algorithme nous donne un trajet acceptable, la solution est correcte et optimale car on aura le meilleur trajet sans passer par un seul sommet fréquenté. Dans le cas contraire, le résultat est simplement le plus court trajet mais passera par des lieux fréquentés.

Cette solution n'est donc pas satisfaisante, on souhaiterai minimiser le nombre de lieux fréquenté.

Solution 2 (résultat bon mais lent) :
Cette solution serait de tester d'exécuter l'algorithme précédant en mettant dans un tas les sommets ''fréquenté'' et dans le cas où le résultat excède le paramètre de tolérance. Relancer l'algorithme en sortant les sommets du tas (FIFO ou LIFO).
Complexité = en O((nbSommets+nbSuccesseurs)*log(nbSommets)*(nbSommets))

Remarque : Le résultat sera souvent meilleur mais le temps d'exécution plus long si on utilise la méthode FIFO pour le tas.

La solution sera correcte mais ne sera pas optimal, en effet on inclue au fur et à mesure de nouveaux sommets ''fréquentés'' cependant rien ne nous assure que la combinaison retourné est la meilleur (minimisant le nombre de sommet ''fréquentés'').

En réalité parmi ces sommets ''fréquenté'' certains le sont plus que d'autre, on pourrait donc affecter une valeur de fréquentation à chaque sommet (par exemple : un sommet ayant 6 successeurs aurait pour cout de fréquentation 6). Avec cette façon de procéder on pourrait dans un monde idéal, calculer chaque trajet possible (passant une seul fois maximum par sommet) et pour chacun d'entre eux donné le coût en distance (ou durée) et le cout en fréquentation totale. La solution optimale serait le trajet ayant le cout de fréquentation le plus bas parmi ceux qui ont une distance (ou durée) inférieur au meilleur trajet multiplié par le paramètre de tolérance.
Un tel algorithme nous donne le résultat optimal mais à une complexité trop grande pour être fonctionnel.
Complexité : O(nbSommets !)
Remarque : Dans ce cas on pourrait limiter de façon importante le nombre de sommet avec certaine condition cependant la complexité dans le pire cas reste la même.

Ouverture : On pourrait également souhaiter vouloir minimisé le ratio coût de fréquentation * coût en durée/distance.