INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

RAPPORT
BUREAU ETUDE DE GRAPHES

BINOME:

- **TRAN Trong Hieu**
- **TRAN Le Minh**

GROUPE : 3 MIC C

Sommaire

I. Introduction :

Il s'agit d'un bureau d'étude où on va appliquer les compétences de programmation en Java et des connaissances théoriques en Graphe pour implémenter les recherches du plus court chemin sur une carte donnée, au lieu des simples graphes créées avec des noeuds et des arcs.
Dans ce bureau d'études, nous avons réalisé :

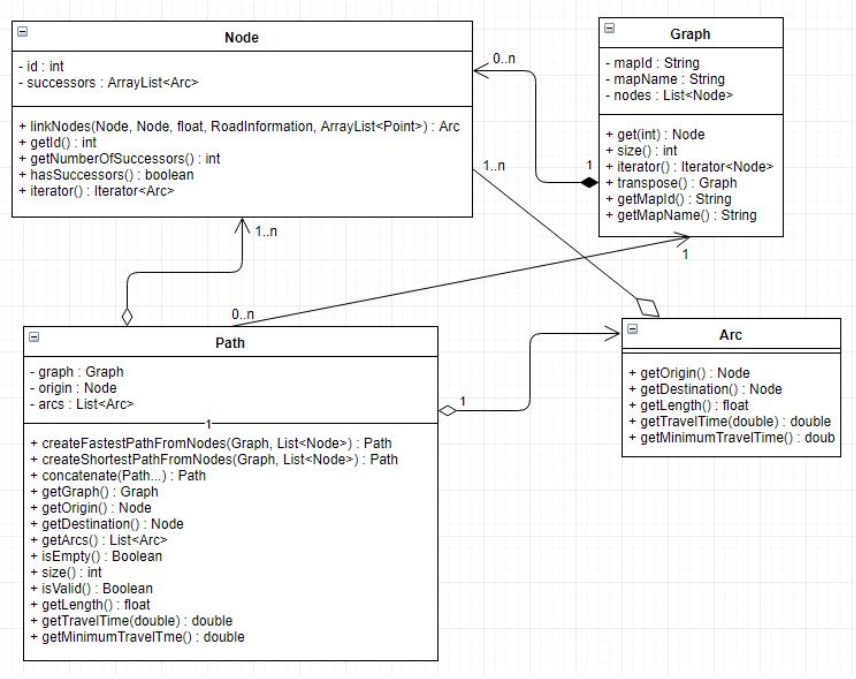
- L'implémentation des classes de base, à savoir la classe *Path* basant sur les classes *Graphe, noeud* et *Arc.*
- L'implémentation de l'algorithme de Dijkstra
- La mise en place des tests de validité
- L'implémentation de l'algorithme A* - version améliorée de l'algorithme de Dijkstra
- La mise en place des tests de performances
- L'implémentation du problème ouvert : Le covoiturage

Vous trouverez dans ce rapport les livrables à rendre, quelques explications sur le travail effectué, et une partie dédiée au problème ouvert.

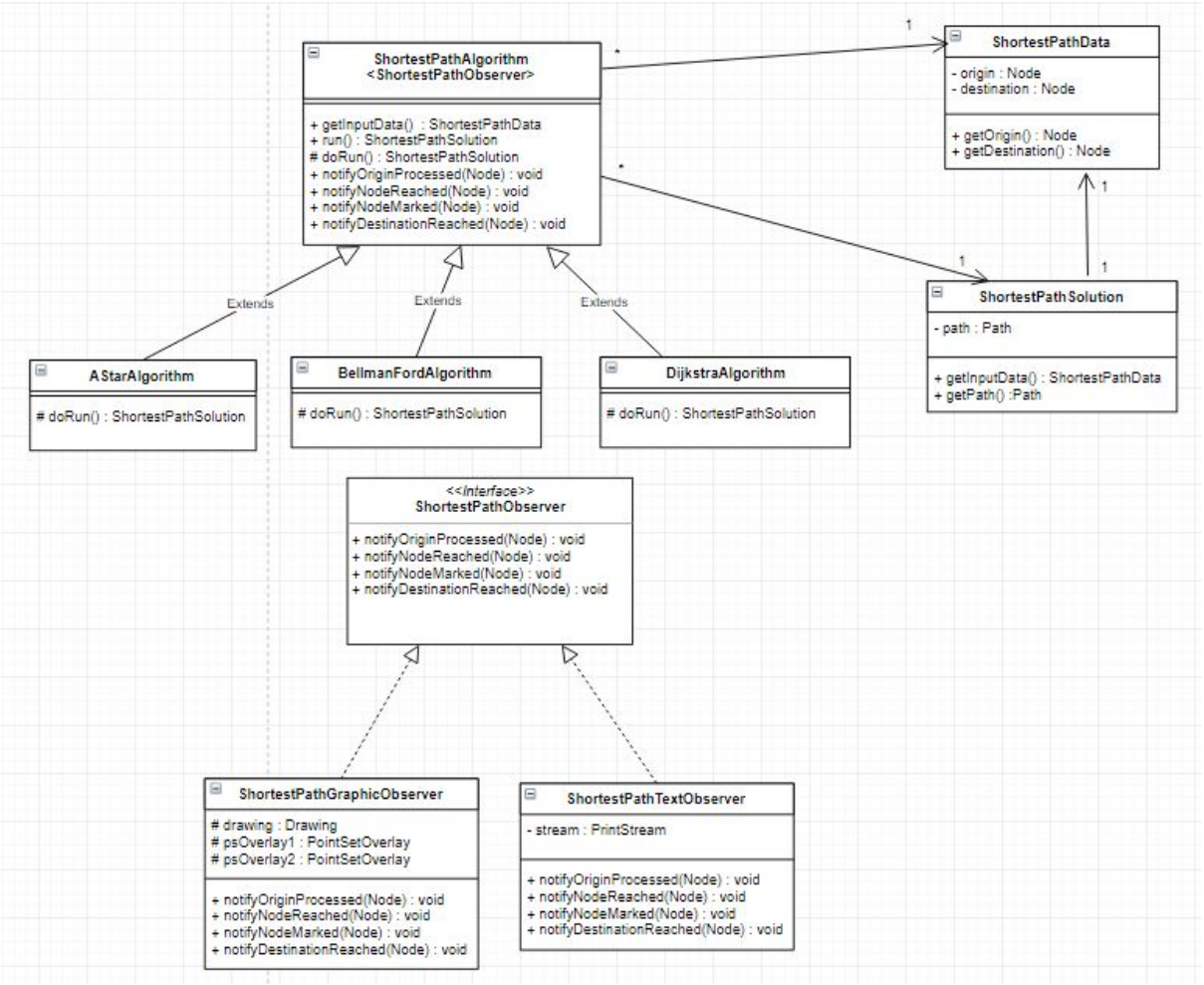
II. Livrables :

1. Documents de conception :

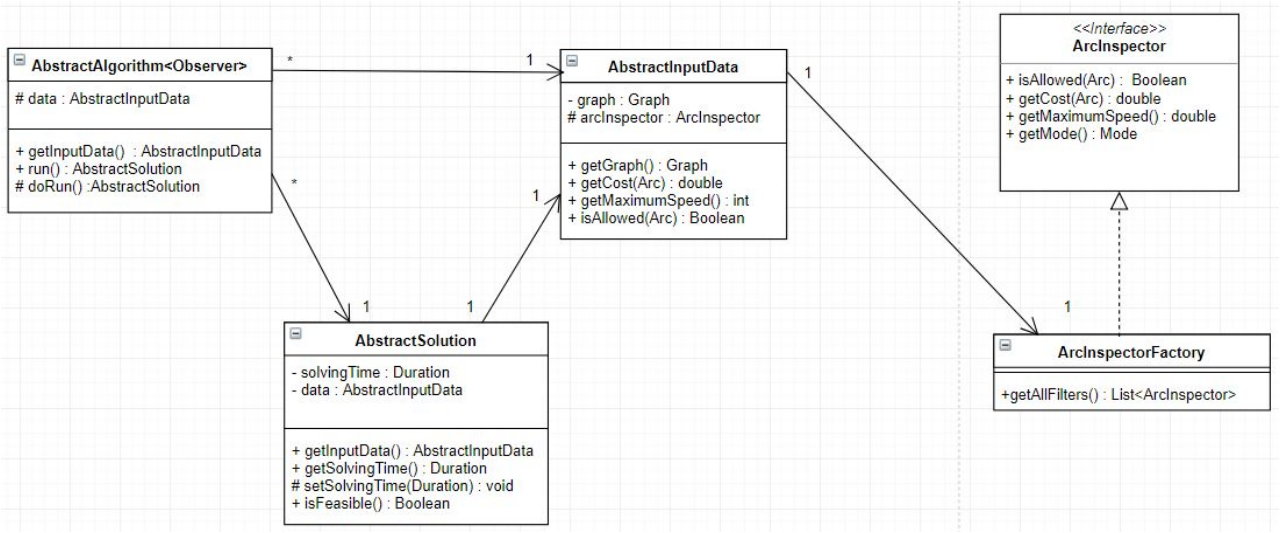
- Diagramme de classe du paquetage *graph* :



- Diagramme de classe du paquetage *algo.shortestpath* :



- Diagramme de classe du paquetage *algo* :



- Conception du label pour l'algorithme de Dijkstra :
  - ➢ Attributs : int **id,** double **cost,** Arc **father,** boolean **mark.**
  - ➢ Constructeur : Label(int id, double cost, Arc father, boolean mark).
  - ➢ Méthodes : getId, getFather, setFather, isMarked, setMark, setCost, getCost, getCostEstimate, compareTo, toString.

- Conception du label pour A* :
  - ➢ LabelStar hérite Label
  - ➢ Attributs en plus : double **costEstimate.**
  - ➢ Constructeur :
    - + LabelStar(int id, double cost, Arc father, boolean mark)
    - + LabelStar(int id, double cost, Arc father, boolean mark, double costEstimate)
  - ➢ Méthodes redéfinies : getCostEstimate, getCost, toString.

2. Campagne de Test de validité :

Tous ces tests sont effectués en nous inspirant sur les tests JUnit de *PathTest, PriorityQueueTest,* où la vérification des résultats se font par la fonction *assert*.

- Test avec oracle sans carte :

Ce test consiste à tout d'abord créer un graphe connexe avec des noeuds et des arcs. Puis on valider le résultat de l'algorithme de Dijkstra en le comparant avec celui de l'algorithme de Bellman-Ford.

- Test avec oracle sur une carte donnée :

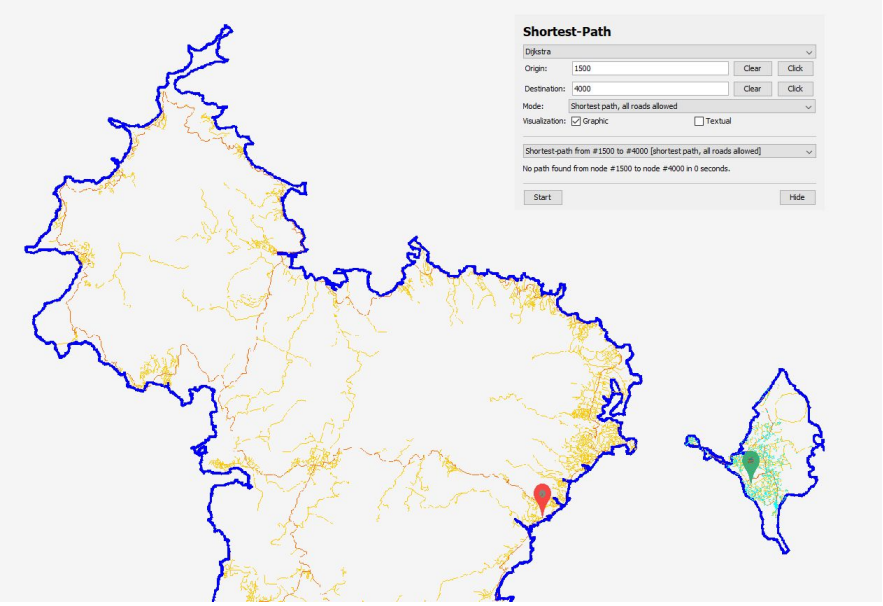
Ce test valide l'algorithme de Dijkstra en comparant son résultat avec celui de l'algorithme de Bellman-Ford, mais cette fois-ci on applique ces algorithmes sur une carte précise.

- Test sur une carte donnée sans oracle :

N'utilisant pas le résultat d'une autre algorithme, on doit utiliser de différents scénarios pour pouvoir valider l'algorithme de Dijkstra.

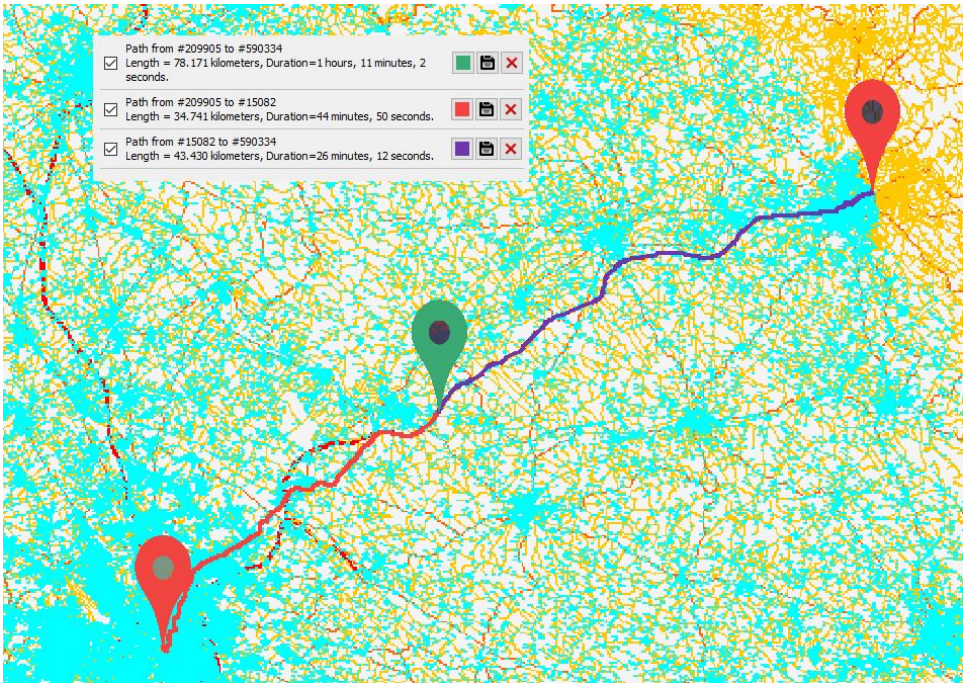
- Test de chemin impossible : on fait le test sur la carte **INSA**, de l'origine 1 et de destination 1000 (un point plutôt bizarre sur la carte donc visuellement impossible à atteindre)

- Test de chemin inexistant (PasDeChemin) : on fait le test sur la carte **Mayotte**. Il s'agit d'une carte avec deux îles, alors il sera possible de trouver des couples de noeuds non connexes car ils se trouvent sur deux différentes îles. Pour le test on a choisi pour l'origine 1500 et pour la destination 4000.

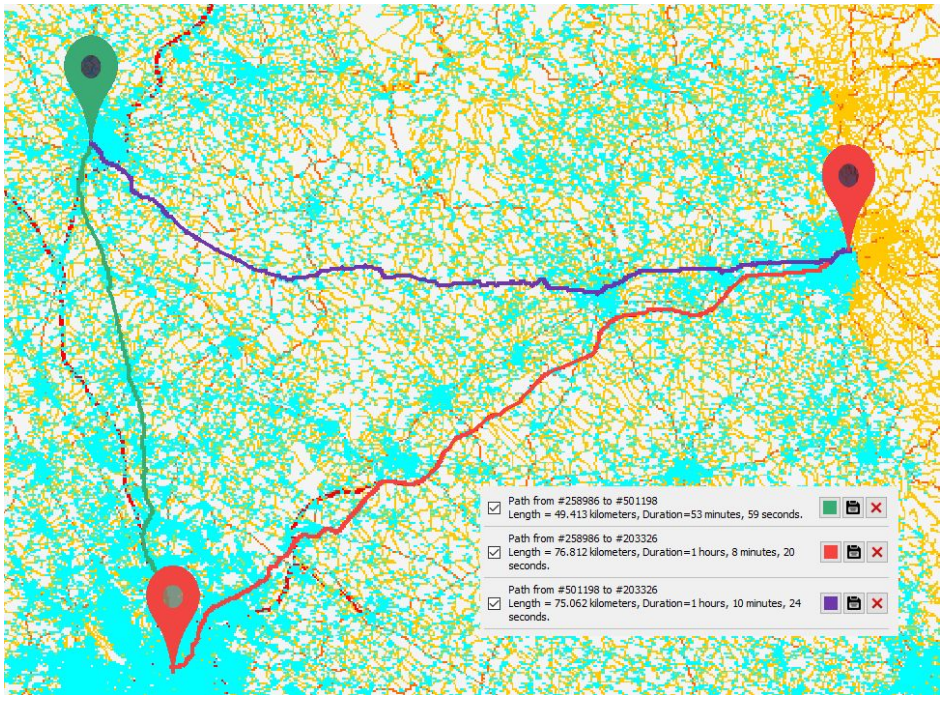


- Test de chemin nul : Il suffit de choisir le même point de départ que le point d'arrivée. On a fait le test sur la carte **INSA**, de point de départ et d'arrivée 250.

- Test de sous chemin (CheminsComposants) : Considérons les trois points A, B, C où B se trouve sur le plus court chemin (PCC) entre A et C. Alors il faut vérifier que la somme en distance et en temps entre le PCC entre A et B et le PCC entre B et C soit égale au PCC entre A et C. On a fait le test sur la carte **Midi-Pyrénées**, avec les coordonnées de A, B, C respectivement 209905, 15082, 590334.



- Test de l'inégalité triangulaire : Soit AC la longueur du PCC entre A et C, et idem pour AB et BC. Donc il faut vérifier, par inégalité triangulaire, que : (AB + BC > AC && AB - BC < AC). On a fait le test sur la carte **Midi-Pyrénées,** avec les coordonnées de A, B, C respectivement 258986, 501198, 203326.



3. Campagne de Test de Performance :

Dans cette partie, on cherche à vérifier la complexité des algorithmes de plus court chemin produites, sachant que la complexité de l'algorithme de Dijkstra est de O(m + nlog(n)) en utilisant le binary heap, avec m le nombre d'arcs et n le nombre de noeuds.

Pour pouvoir le faire, on va calculer le temps que le CPU a utilisé pour effectuer ces algorithmes.

L'idée initiale c'est que l'on va faire un test JUnit qui va générer un fichier contenant le temps CPU pour effectuer 100 fois chaque algorithme sur une carte donnée, entre un point de départ et un point d'arrivé choisis aléatoirement. Avant toute explication faite sur le test effectué, nous nous permettons de vous présenter tout d'abord le tableau de résultat :

Nom de la carte	Taille de données (noeuds + arcs)	Temps CPU de Dijkstra (ms)		Temps CPU de A* (ms)	
		Temps effectué	Temps en réalité	Temps effectué	Temps en réalité
carre	313	23	17	16	6
insa	4236	104	93	212	53

mayotte	25311	236	215	838	122
bordeaux	50014	496	457	1588	154
toulouse	131050	904	862	4070	302
benin	468505	3700	3623	14296	2564
vietnam	136678	10309	10082	43927	8669
midi-pyrenees	2163120	22274	20542	68146	7343
carre-dense	6105608	27963	25873	40651	7681
belgium	3539291	33608	32684	105849	13680
greece	4157960	32067	28518	137279	25031

Où le *"temps effectué"* est le temps calculé depuis l'appel de l'algorithme jusqu'à la fin, calculé dans le test JUnit :

```
…
//boucle de 100 fois d'effectuer l'algorithme
...
long begin = System.currentTimeMillis();
algo.doRun();
long duration = System.currentTimeMillis() - begin;
time1 = time1 + duration;
...
…
```

Et le *"temps en réalité"* est le temps mesuré depuis la partie principale de l'algorithme, calculé dans la classe de cette algorithme. Ainsi ce temps ne comprends pas les étapes d'initialisations, et il n'y a que de vraies étapes de calcul de l'algorithme. Donc on a ajouté les variables publiques ***Duration*** dans la classe de l'algorithme qui sont utilisées à cet effet :

```
<Algorithme.java>
//Etape d'initialisation
…
long begin = System.currentTimeMillis();
//Etape de calcul
…
//fin calcul
this.Duration = System.currentTimeMillis() - begin;

<TestPerformance.java>
…
//boucle de 100 fois d'effectuer l'algorithme
```

```
//calcul de time1
time2 = time2 + algo.Duration;
…
```

Mais pourquoi ces distinctions ? En effet, au début nous n'avons fait que des tests pour avoir ces *"temps effectué"*. Puis on a observé des anomalies dans ces résultats : comme A* est une algorithme qui prend en compte la distance absolue entre le noeud à traiter et la destination, elle doit avoir moins de noeuds à traiter que l'algorithme de Dijkstra et donc avoir le temps CPU plus vite. Or les résultats trouvés ont montré le contraire.

On a ensuite détecté ce problème qui vient de l'étape de l'initialisation dans la classe de l'algorithme A* : on a fait un boucle pour calculer la distance absolue entre la destination et TOUS les noeuds de la carte. C'est pourquoi à chaque lancement de A*, elle va prendre beaucoup de temps à s'initialiser, avant de commencer les étapes de calcul.

En conséquence, d'une part, on a calculé ce *"temps en réalité"* pour pouvoir observer la vraie qualité de A* au niveau de complexité en temps, et les résultats obtenus paraissent logiques, vu que le vrai temps CPU de A* est beaucoup plus vite que celui de l'algorithme de Dijkstra.

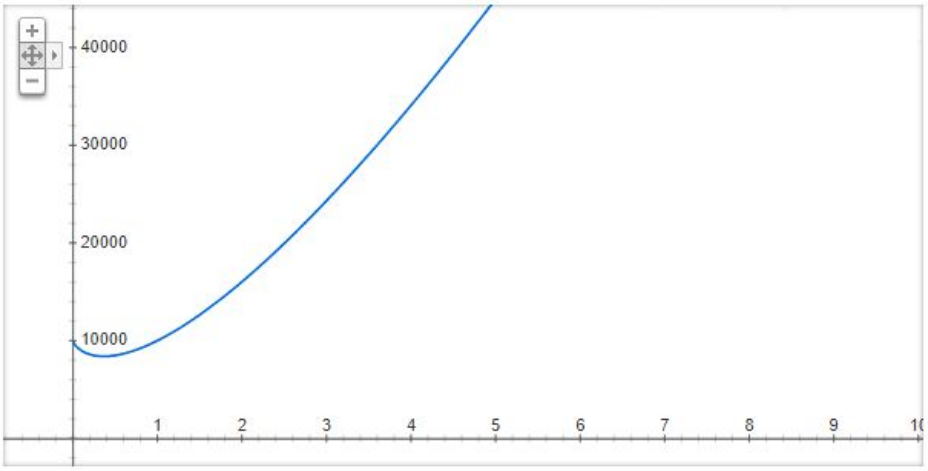
D'autre part, même si on arrive à tester A* visuellement sur les cartes, on doit penser à des modifications nécessaires pour l'étape d'initialisation, probablement sur la conception des Labels.

Des autres debuggings à faire : concernant le binary heap des algorithmes, comme vu en TP, nous avons rencontré des erreurs anormales sur la méthode ***remove()***, et nous n'arrivons pas à corriger malgré l'intervention des enseignants. Donc nos enseignants nous ont donné une idée qu'il ne faut pas enlever un label dans l'étape de mise à jour de distance, car celui avec la distance la plus courte se trouve toujours au début du heap trié (par comparaison en poids de l'arc). Donc il faut juste l'ajouter dans le heap si sa nouvelle distance est meilleure que l'ancienne. Pourtant cette façon va éventuellement causer des problèmes sur la mémoire du heap, à cause de laquelle on n'a pas pu tester les cartes volumineuses comme la France.

Voici le graphe obtenu à travers les résultats :

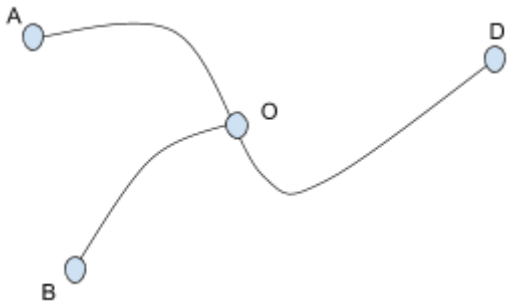
En effet les cartes traitées contiennent à la fois des cartes routières, un carré et un carré dense, où le dernier explique une chute en temps CPU au bout de ce graphe. Sans tenir compte de cette carte-là, on aura un graphe qui est relativement proche à la courbe
o(m + nlog(n))

(cf. Google) :



III. Problème ouvert : Le covoiturage

Problématique : On veut implémenter le problème de covoiturage sur des graphes : une voiture (en A par exemple) et un piéton (en B) veulent aller à la même destination (en D). Alors, la solution est de se rencontrer à un noeud de "rendez-vous" O pour que la voiture puisse accueillir le piéton et aller ensemble à la destination. Pour ne considérer qu'un seul objectif, on cherche à minimiser la *somme des coûts* des parcours, càd le chemin à partir de l'origine de la voiture, du piéton vers le point rendez-vous et du point rendez-vous vers la destination.



Il existe O tel que coût(AO + BO + OD) min ?

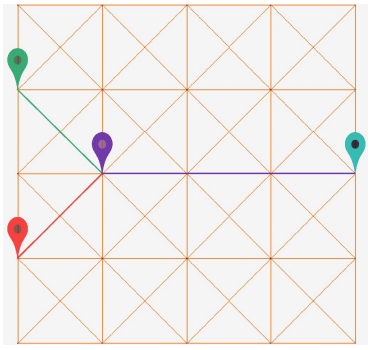
Principe de résolution :

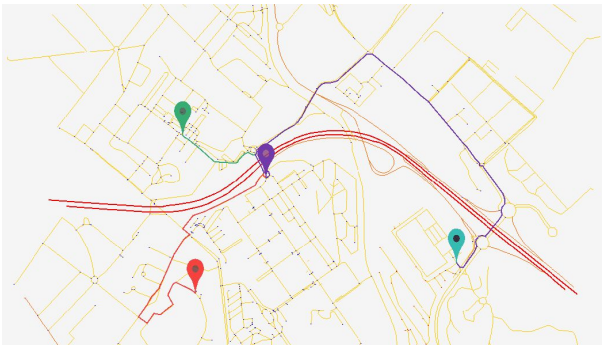
On recherche tous les noeuds communs entre le trajet de A vers D et de B vers D , puis on calcule le coût pour se déplacer entre ces noeuds communs vers la destination D. Pour

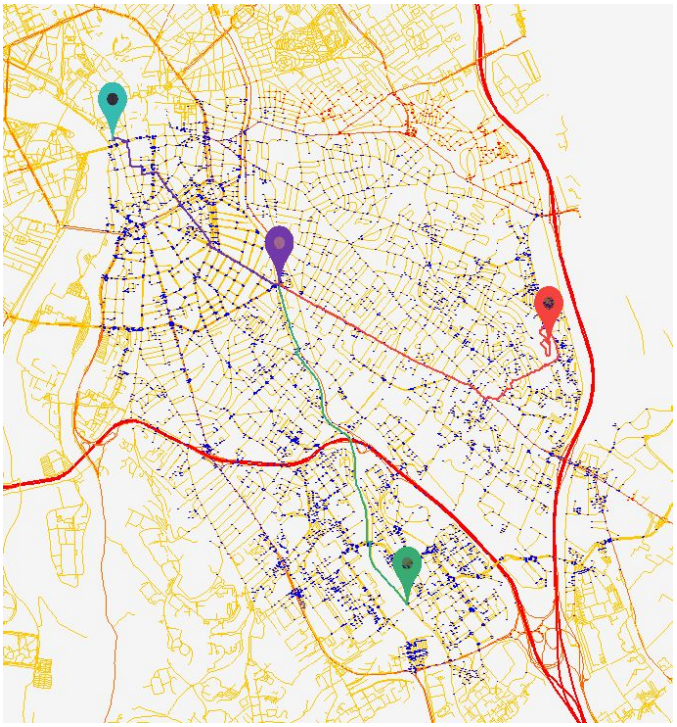
chaque noeud O comme cela, on va minimiser le coût AO + OB + OD pour trouver un bon noeud. Pourtant, il faut limiter les points considérés pour que le temps d'exécution ne soit pas trop long. Alors notre idée est d'appliquer l'algorithme PCC (plus court chemin) comme suivant :

- Chercher le plus court chemin de A vers tous les noeuds, posons N1 l'ensemble des noeuds marqués .
- Chercher le plus court chemin de B vers tous les noeuds, de même façon on a N2.
- Dans le graphe transposé, on cherche aussi le plus court chemin de D vers tous les noeuds et notons N3 l'ensemble des noeuds marqués. Le fait de chercher le plus court chemin depuis D sur un graphe transposé revient à la recherche du plus cours chemin depuis tous les points vers D sur le graphe initial. La différence c'est qu'il ne faut que lancer l'algorithme de PCC une seul fois sur le graphe transposé.
- Puis avec les noeuds dans l'ensemble d'intersection de N1, N2 et N3, on cherche le noeud qui a le coût le plus petit (min AO + BO + OD).

Quelques résultats du test :







IV. Conclusion

A travers des séances de bureau d'études de Graphes, nous avons l'opportunité de faire des travaux pratiques avec les connaissances théoriques acquises à l'issue des cours de Graphe en implémentant les algorithmes de plus court chemin en programmation objet orienté et en langage Java.
Pour la première fois nous avons témoigné l'efficacité des tests automatisés effectués avec le JUnit4. Ceci nous donne des idées sur la façon d'examiner, de valider et éventuellement de debugging pour nos projets Java dans l'avenir.
Malgré un petit soucis sur les résultats des tests de performance, on arrive encore à produire des algorithmes qui fonctionnent bien, comme vous pouvez trouver dans les tests de validité, ainsi que leur application pour le problème de covoiturage.

-Fin-