

**Rapport BE Graphes**

Git : <https://github.com/TheauTheau/BE_Graphes>

**Théau Giraud**

**29/04/2020**

Partie BE Graphes

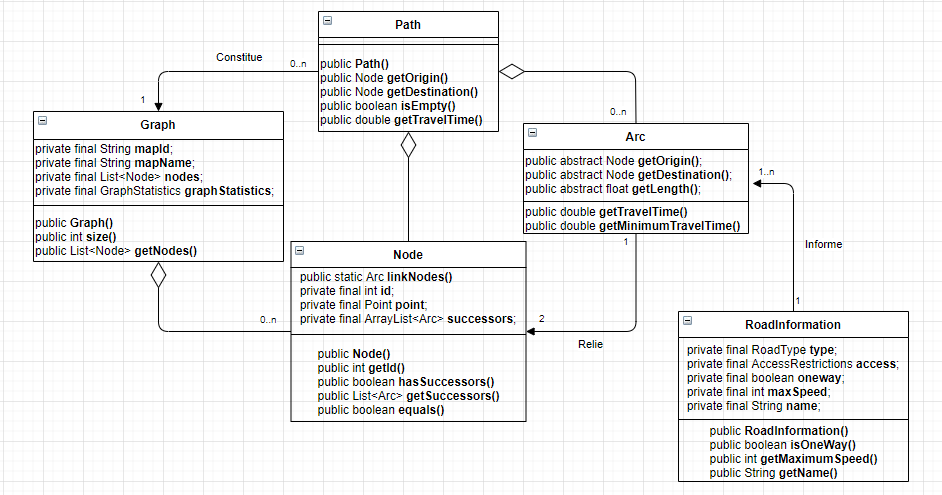
**Introduction**

Le BE Graphes a été notre projet d’études ce semestre. Il a consisté (principalement) d’une implémentation de l’algorithme de Dijkstra vu en cours, puis de l’améliorer en version A\*. Cela m’a permis de bien comprendre les notions du cours, d’acquérir de bonnes connaissances en implémentation de parcours de graphe et de visualiser leurs résultats.

Dans ce rapport, je vais aborder l’implémentation de l’algorithme en java, ainsi que toutes les phases de préparation ultérieures, les tests de validité de mes algorithmes, les difficultés rencontrées et les conclusions faites sur cet apprentissage. L’algorithme en question est Dijkstra tel que l’on l’a vu en cours puis son amélioration, A\*. Je comparerai aussi leurs performances.

1. **Classes Node/Arc/Graph/Path**

On nous a donné un gros projet sous Eclipse presque entièrement implémenté. La réalisation de Dijkstra se fait en parcourant un ***Graph*** qui est un ensemble de ***Nodes* (points)** qui eux-mêmes sont reliés par des ***Arcs****.* On parcourt tous ces points pour construire et rendre un ***Path* (chemin constitué d’Arcs)**. Les classes données dans le projet sont implémentées avec toutes sortes de méthodes utiles pour renvoyer le Node d’origine d’un Arc, la distance que fait cet Arc, pour créer des Path, etc.

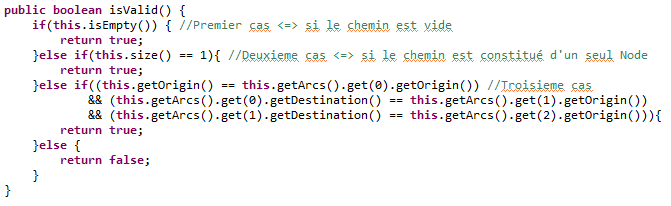
Voici un **diagramme UML** des différentes classes données, avec les méthodes et attributs les plus importants en gras :

On remarque que la représentation informatique des Arcs est de type liste d’adjacence (dans Node on a une méthode qui renvoie *ArrayList<Arc> successors*).

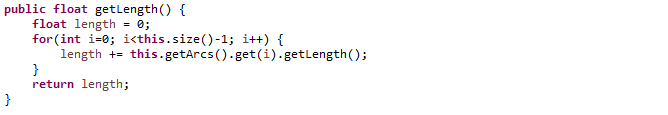
1. **Implémentation de Path()**

La classe **Path** est la classe à implémenter. Il faut coder y coder six méthodes qui nous permettront de mieux comprendre l’agencement de l’algorithme et comment parcourir un Graph. Voici leurs codes :

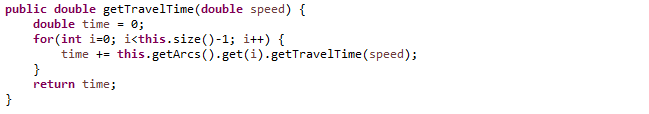
***isValid()*** renvoie *True* si le Path créé est valide. Cela est vrai pour plusieurs cas possibles ; si le Path est vide, s’il contient un seul Node et s’il contient au moins deux Nodes, il faut que le premier Arc ait pour origine l’origine du Path et qu’au moins deux Arcs consécutifs dans le Path créé aient pour destination l’origine de son suivant. C’est ce que j’ai implémenté dans les **différentes conditions de ma boucle** :



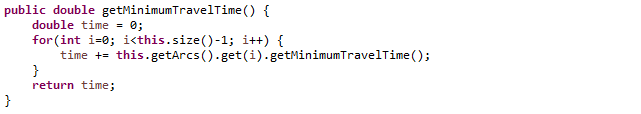
***getLength()*** renvoie la longueur du Path tout entier. C’est une simple somme sur la longueur des Arcs le constituant.



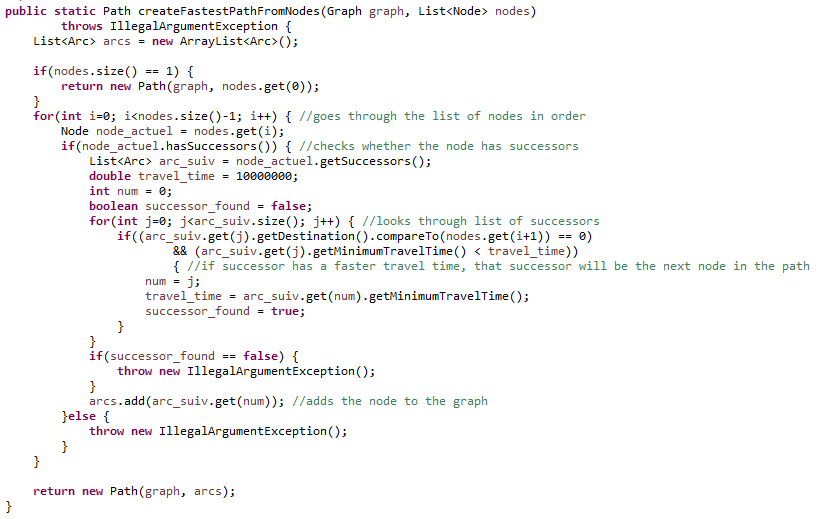
***getTravelTime(double speed)*** renvoie le temps que l’on prendra pour parcourir le Path avec une vitesse (*speed*) constante donnée. C’est ici aussi une somme.



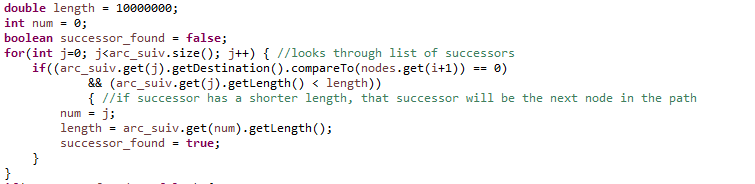
***getMinimumTravelTime()*** renvoie le temps pris pour parcourir le Path si on parcourt le Path avec la vitesse maximale de chaque Arc. J’ai ici utilisé **la méthode *getMinimumTravelTime()* de la classe Arcs** qui renvoie déjà le temps minimum de chaque Arc à vitesse maximale.



***createFastestPathFromNodes(Graph graph, List<Nodes> nodes)*** reçoit une liste de Nodes, la parcourt dans l’ordre et en crée un Path dans le Graph fourni. Si plusieurs chemins différents sont possibles il choisit le plus rapide. En effet, un Node peut avoir plusieurs successeurs et notamment **plusieurs Arcs peuvent exister entre 2 sommets** consécutifs. Dans le code, cela se traduit par un *node\_actuel.getSuccessors()* ;



***createShortestPathFromNodes(Graph graph, List<Nodes> nodes)*** fait la même chose mais renvoie une solution **en distance**. La seule différence est donc sur le deuxième for imbriqué qui fait appel à *getLength()* au lieu de *getMinimumTravelTime()*:

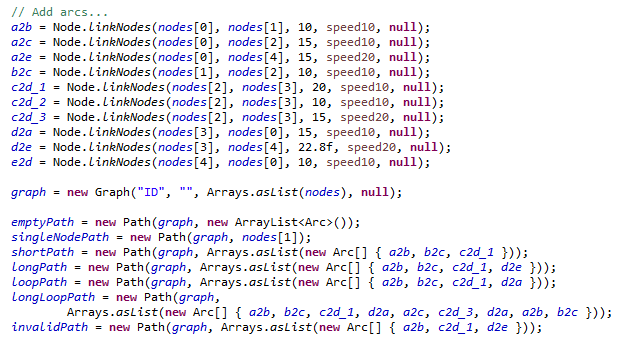


1. **Tests JUnit**

Pour vérifier si les méthodes codées étaient correctes, nous avons eu recours à des **tests *JUnit***. Ce sont des **tests unitaires** correspondants à chaque méthode et qui présentent **différents cas d’erreurs** possibles pour celles-ci et comparent le résultat rendu par la méthode codée avec le résultat attendu.

**Les cas d’erreurs sont ceux-ci** : chemin long et court, chemin vide avec un seul sommet, chemin réalisant une boucle et chemin non valide (par rapport à *isValid()*). Les tests vont réaliser des *assert()* sur ces chemins qui invoquent ensuite les méthodes (*True/False* ou *Equals()* pour comparer avec le résultat attendu.

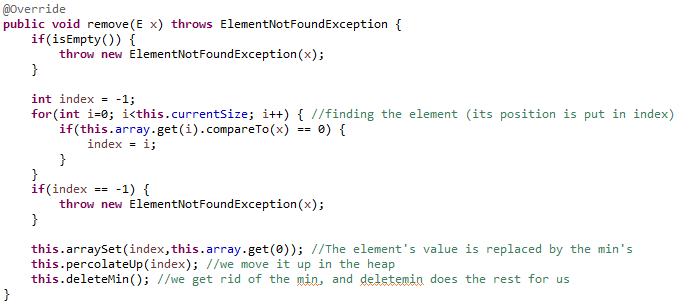
On peut observer ci-dessous l’implémentation des cas de figure dans les *JUnit* :

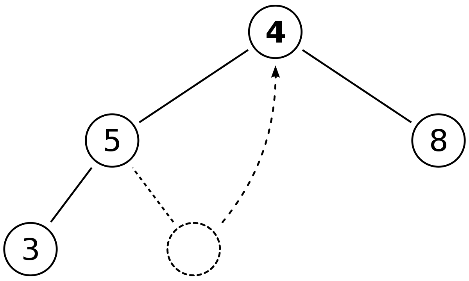


Il faut donc prendre en compte tous ces cas dans le corps des méthodes et s’en aider pour les implémenter. Certaines méthodes ne nécessitent pas d’ajouts en particulier comme *getLength()* ou *getTravelTime()*, alors que *isValid()* nécessite plus de réflexion :il doit gérer séparément dès le début les cas de chemin vide/avec un seul Node dans sa boucle if, et on se base sur ces tests pour créer la méthode.

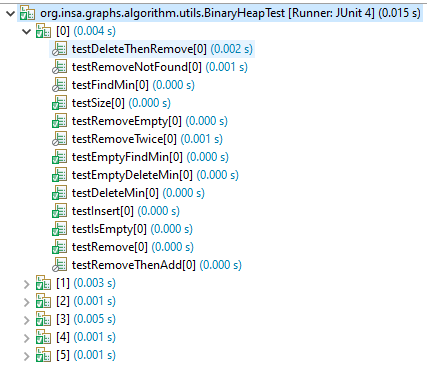
1. **Binary Heap et *remove()***

Pour implémenter proprement l’algorithme de Dijkstra, il nous faut utiliser un tas binaire (aka Binary Heap). Cette structure est partiellement développée dans le projet qui nous a été fourni, il reste une méthode importante à coder qui est la méthode *remove()* que voici :



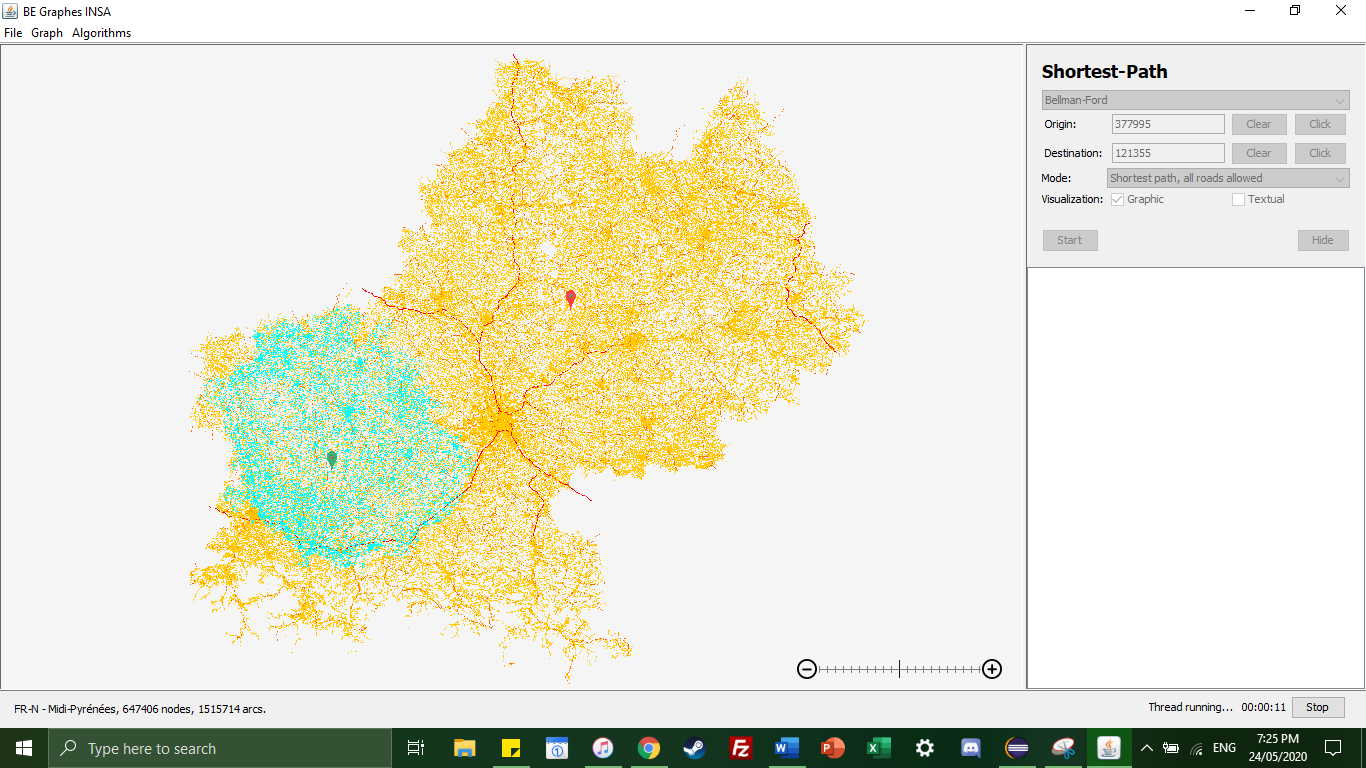
 C’est une méthode qui enlève un certain élément du tas. Si on ne trouve pas cet élément, il nous faut renvoyer une exception *ElementNotFound*. **Le principe du remove dans un tas se déroule ainsi** : On copie le minimum à la place de l’élément qu’on veut détruire, on le ramène ensuite en haut et on enlève le minimum - sachant qu’à ce stade il y en aura deux, donc il nous restera un minimum et le tas restera valide.

Pour l’implémenter, j’ai d’abord parcouru le tas pour trouver l’élément (et levé l’exception si nécessaire), puis j’ai utilisé les **fonctions fournies dans la classe BinaryHeap**. *arraySet* pour mettre la valeur de l’élément à celle du minimum, puis *percolateUp* pour réarranger l’ordre (on a donc deux min égaux en haut du tas), et j’ai enfin utilisé *deleteMin()* pour enlever un de ces deux minimums.



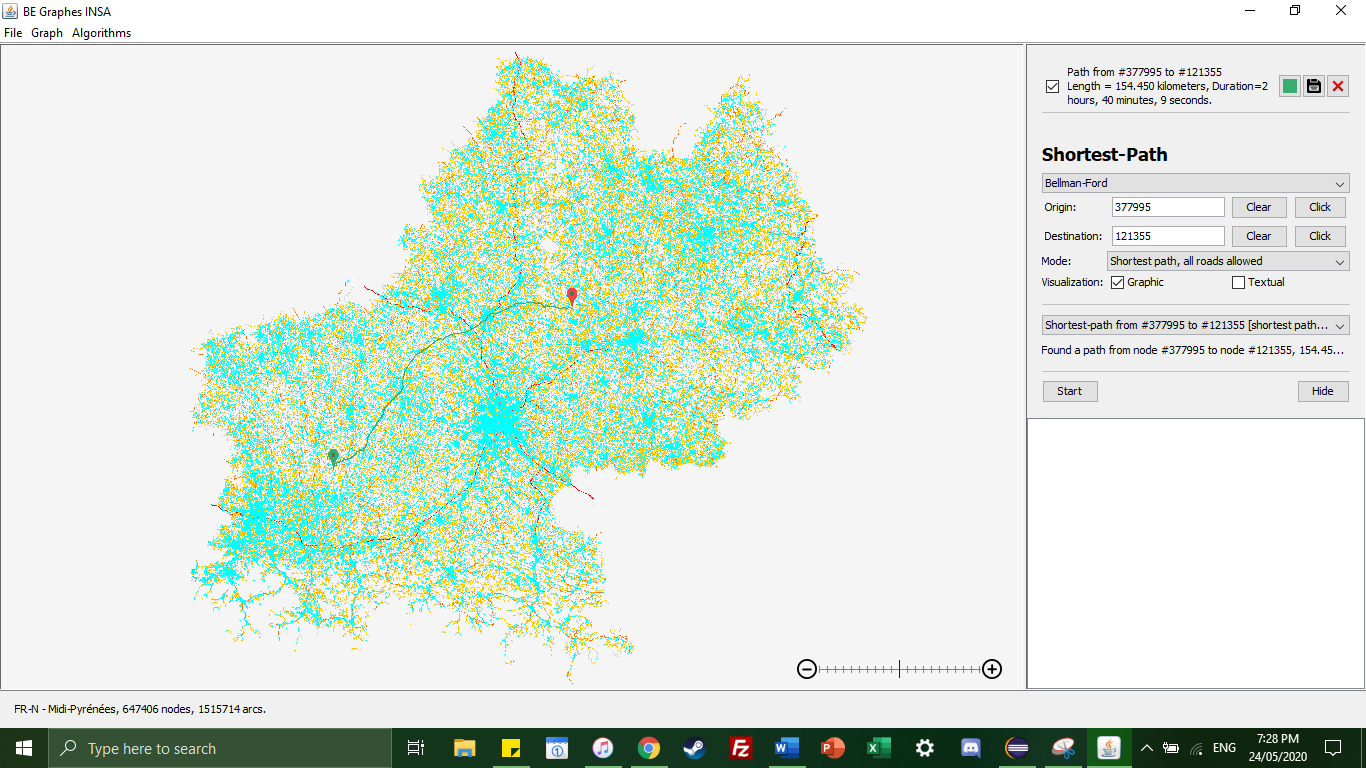
En me servant des tests JUnit associés à la classe BinaryHeap, j’ai pu comprendre les cas d’erreurs et m’adapter. On peut voir que **les test JUnit s’effectuent ainsi** : sur un tas de taille variable (0 à 6), on tente, un *delete()*, un *remove()*, etc. dans des ordres différents pour vérifier que tous les cas passent.

1. **Bellman-Ford**



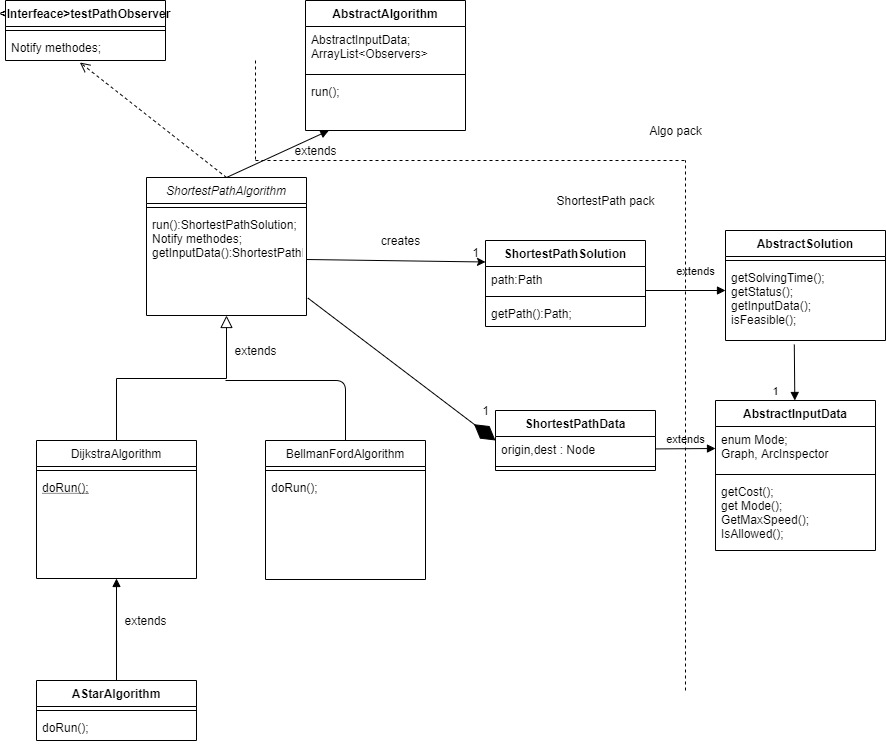
**Bellman-Ford** est un algorithme de recherche de **Plus Court Chemin** (PCC). Il nous est donné pour observer le déroulement attendu de nos propres programmes. Voici un exemple de son déroulement sur la carte des Midi-Pyrénées :

 Bellman-Ford doit visiter **tous les Nodes** d’un Graph pour pouvoir trouver le PCC. On voit ici qu’il a dépassé le point d’arrivée, mais qu’il continue à visiter tous les autres sommets :



Une fois qu’il a visité tous les sommets, il peut enfin donner le **PCC (ligne verte** ci-contre). Bellman-Ford n’est **pas un algorithme très performant** ; il a mis à peu près trois minutes pour donner sa réponse.

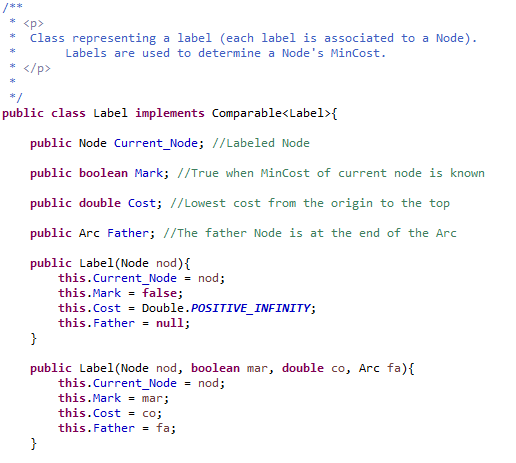
Voici le **diagramme UML** correspondant aux classes fournies pour cet algorithme :



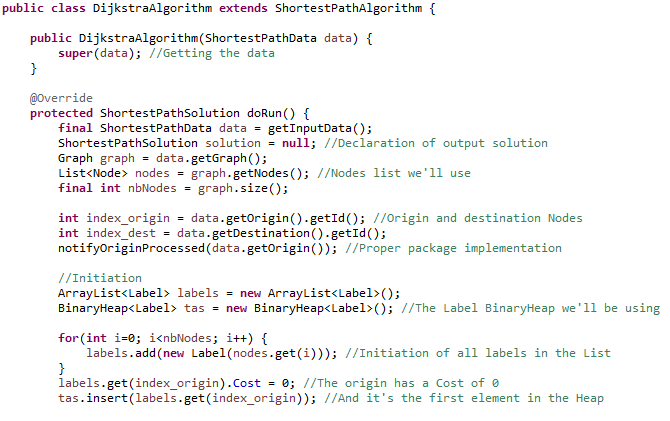
Bellman-Ford (et les algorithmes que nous allons implémenter) est donc un type de *ShortestPathAlgorithm*. Nous avons accès à la *ShortestPathData* donnée en entrée pour parcourir le graphe et implémenter notre solution et la rendre sous le type *ShortestPathSolution*.

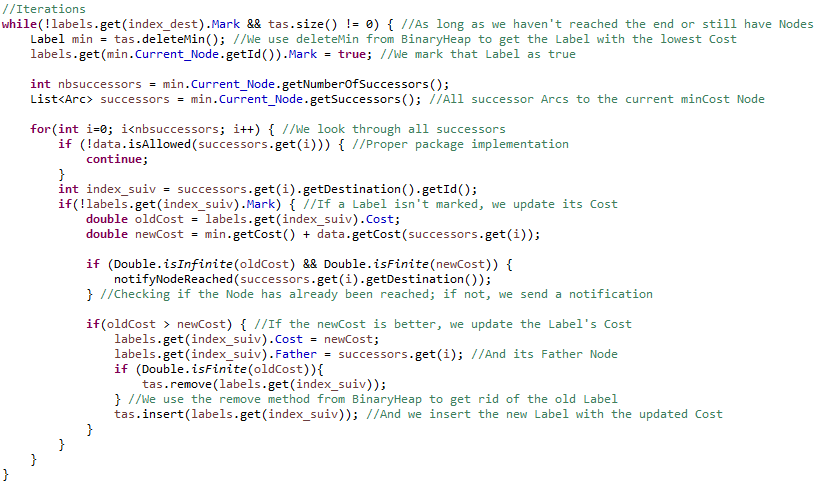
1. **Dijkstra**

Pour coder l’algorithme de Dijkstra, on choisit de représenter les sommets par des ***Labels***. Cette classe Label a les attributs suivants ;



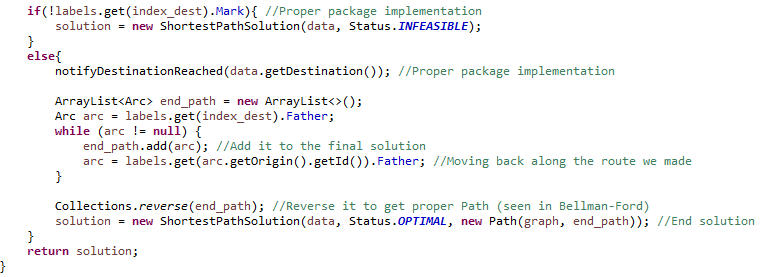
On utilise cette classe au lieu de mettre des Nodes dans la pile car on veut directement avoir accès au **coût d’un Node** (défini par sa distance vis-à-vis du Node précédent). On travaille donc avec un **tas de Labels** dans Dijkstra. Voici **l’initialisation de l’algorithme**:



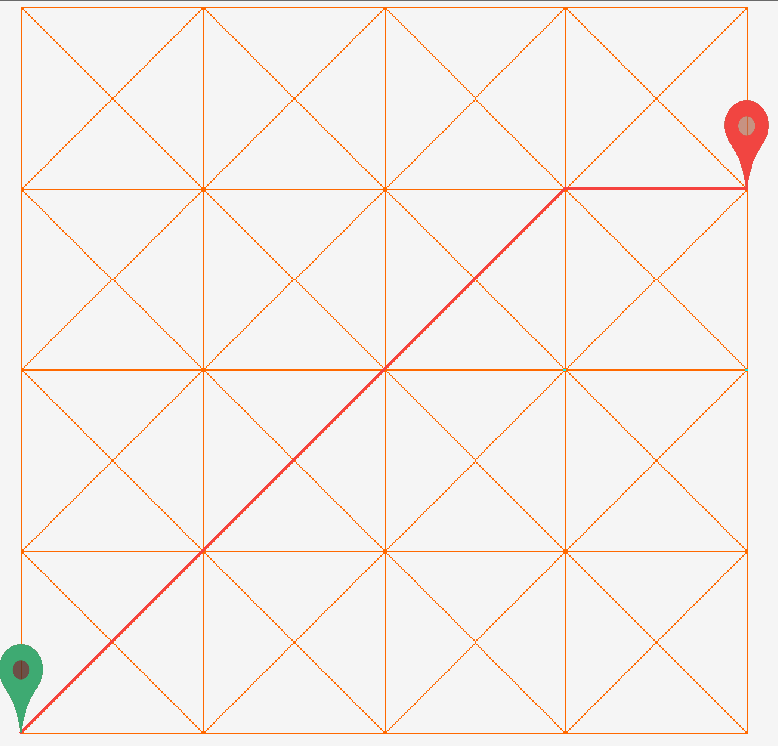
 On **récupère les informati**ons du Graphe grâce à la ShortesPathData donnée. Le tas est **initialisé avec comme premier Label celui associé à l’origine du Graphe**, dont on met le coût à zéro. J’ai aussi utilisé une **liste de Labels** (initialisée avec un coût infini, pas de père, etc. grâce au constructeur dans la classe Label) pour pouvoir parcourir tous les points du Graphe plus facilement. Voici le **corps de la boucle principale** :

Le **corps de la boucle** se déroule ainsi : Tant que l’on n’a pas atteint le Node de destination, on regarde les successeurs du Node de plus petit Cost dans le tas, pour mettre à jour leurs Cost par rapport à cet élément. **On doit utiliser la méthode *remove()*** que l’on avait implémentée dans BinaryHeap pour remplacer le Label ayant un coût supérieur au nouveau coût mis à jour.

Voici la **phase de renvoi du PCC** :

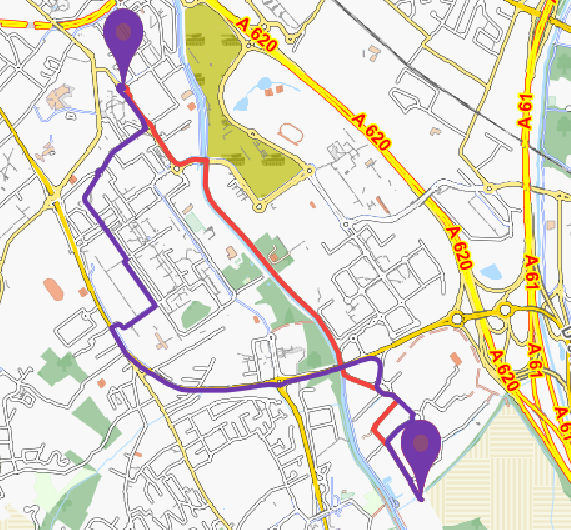


Pour construire le chemin de fin, on **remonte notre *ArrayList* de Labels en partant de la destination**, en prenant le père de chaque élément. Vu que nous avons mis à jour le père de tous les éléments parcourus dans la boucle principale, **le chemin rendu est le PCC**.

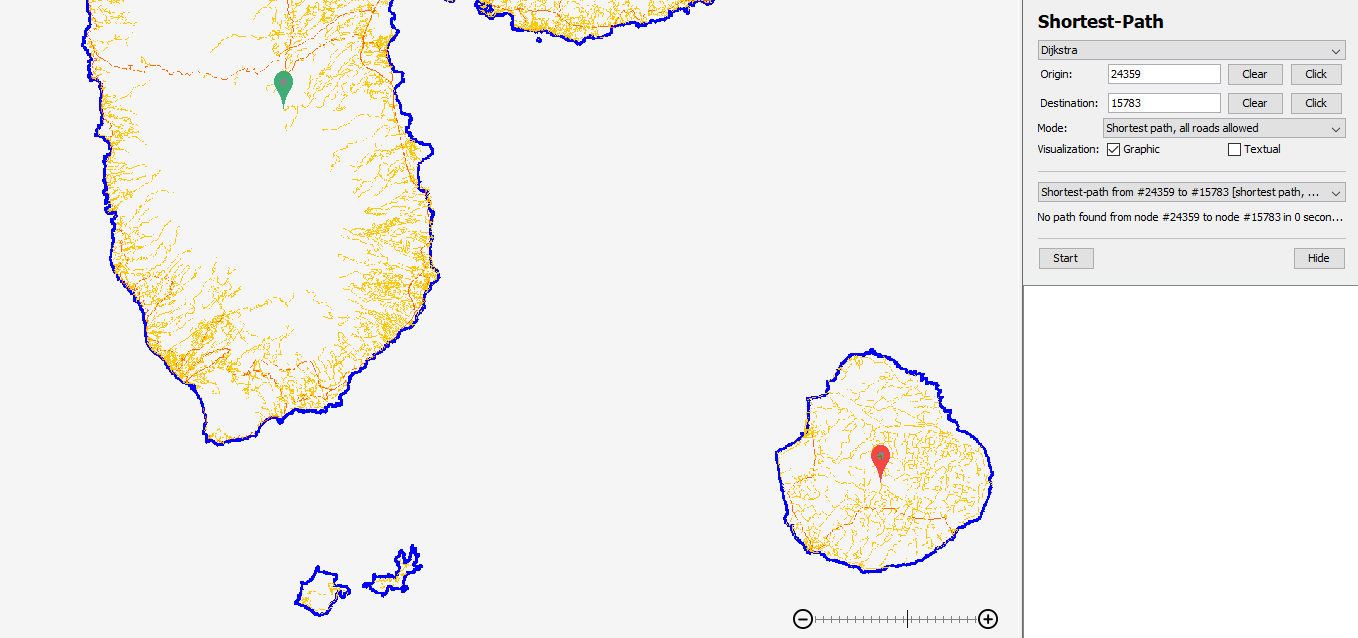


J’ai testé mon algorithme d’abord sur la **carte carrée** fournie, puis sur différentes cartes proposées de plus en plus grandes. En comparant les chemins donnés avec ceux de **Google Maps**, j’ai observé que la différence en distance était négligeable (moins de 5%). Avec ces observations et les résultats visuels de l’algorithme, j’en arrive à la conclusion que **l’algorithme marche comme prévu**.

On s’intéresse notamment aux **options dans le choix de chemin de Dijkstra**. Ici, on voit que le premier chemin rendu (le chemin rouge) passe par une voie suivant le canal longeant le campus de l’INSA. Le chemin violet lui suit seulement les routes qu’une voiture peut emprunter, car on a sélectionné l’option *only roads for open cars* comme choix de ***AbstractInputData***.

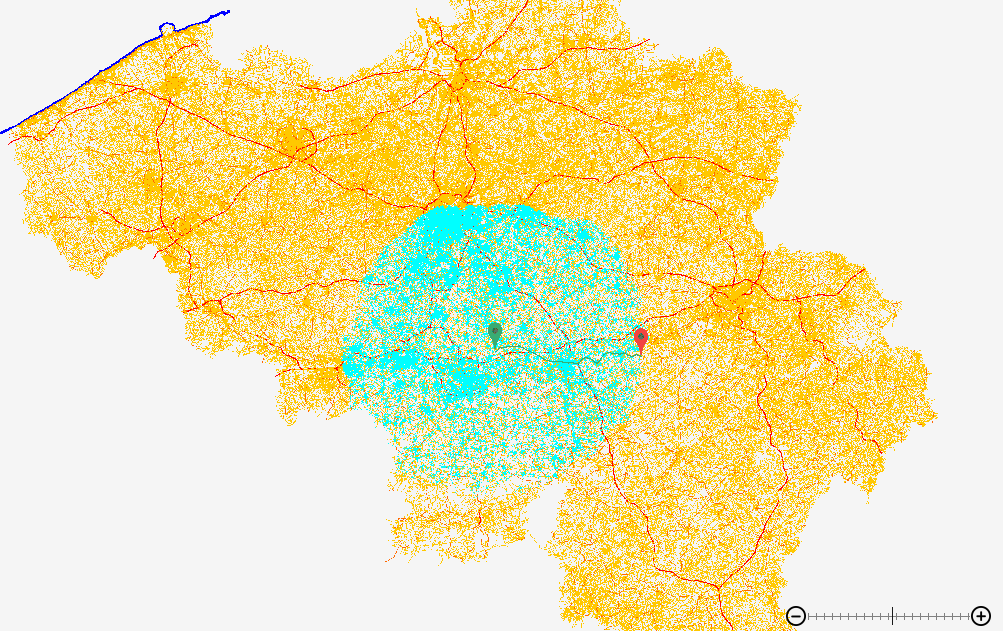
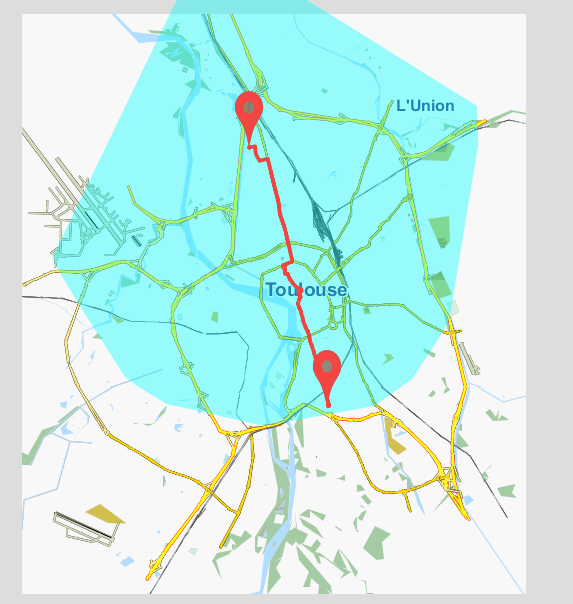


1. **Vérifications de Dijkstra**

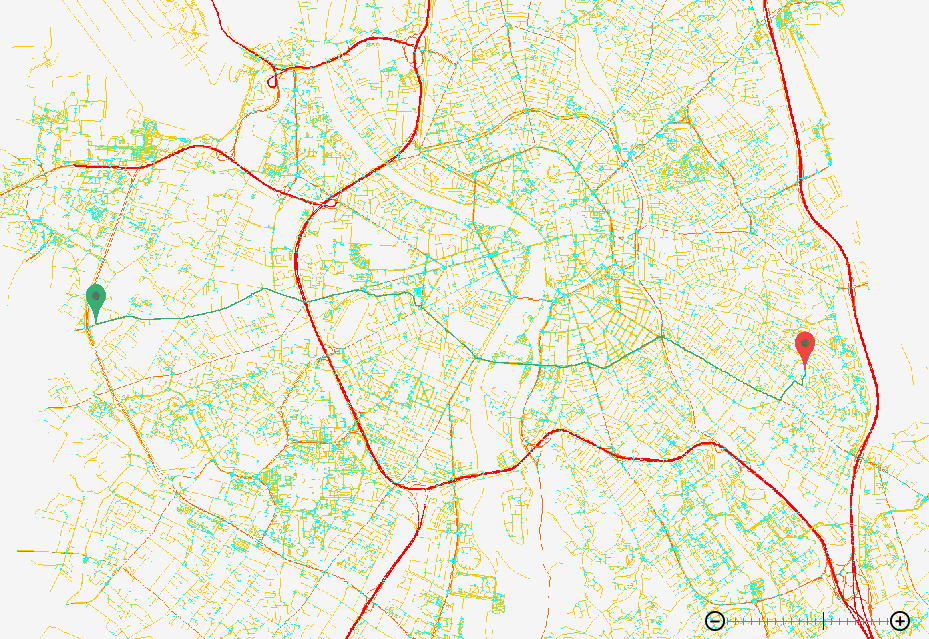
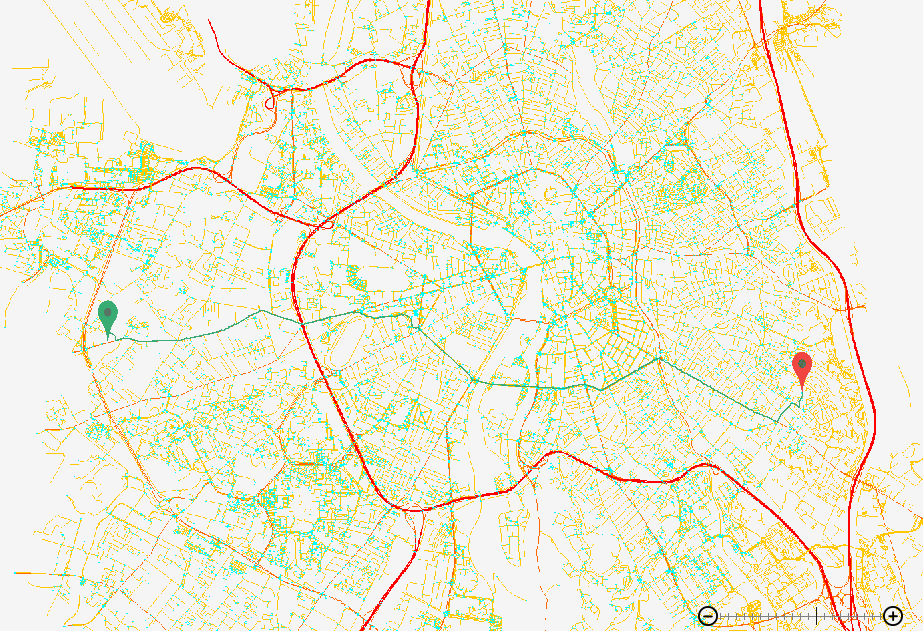
 J’ai aussi voulu vérifier que pour des cas bizarres mon algorithme ne plantait pas. Voici un exemple ou on met comme sommets de départ et d’arrivée **deux sommets non connectés** sur la carte de la Guadeloupe:

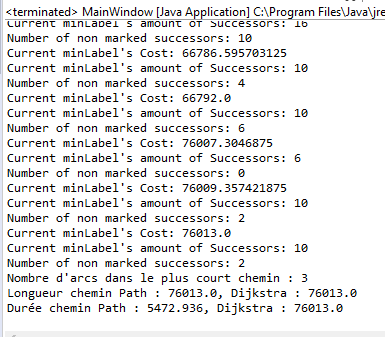
L’algorithme me renvoie bien ***no path found***, tout se déroule bien.

J’ai aussi implémenté des méthodes pour mieux **visualiser le parcours** de l’algo dans les Nodes de la carte en utilisant le *Design Pattern Observer* (toutes les lignes commentées avec *Proper package implementation* dans Dijkstra). Elles permettent d’observer la recherche en **coloriant tous les Nodes atteints en bleu**, comme ceci :



En testant aussi le **même chemin avec Dijkstra et le PCC en distance**, on obtient le même résultat :



En ajoutant des *System.out.println()* j’ai aussi pu afficher des **informations** sur mon tas et le **déroulement de l’algorithme**; on observe aussi que **la longueur du chemin rendu par Dijkstra est la même que celle donnée par la méthode** *getPath().getMinimumTravelTime()*, de la classe Path.

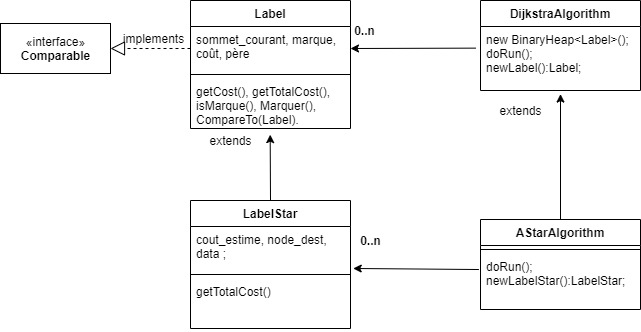
Donc, **mon algorithme de Dijkstra fonctionne comme attendu en distance et en temps**, il n’y a **pas de problème si on rencontre une erreur de saisie** et **la solution retournée est valide**.

1. **A\***

Nous souhaitons maintenant améliorer l’algorithme de Dijkstra codé pour le rendre plus rapide. La variante la plus connue est **l’algorithme A\*** qui va parcourir le Graphe donné **en étant guidé par la destination** au lieu de regarder chaque fils de chaque Node que l’on parcoure. Pour ce faire, nous allons changer les coûts de nos Labels. Le **coût d’un LabelStar** (nouvelle classe créée) prendra aussi en compte le **coût estimé par rapport à la destination**, c’est-à-dire la **distance entre le Node associé et le Node de destination**.

La classe LabelStarhérite donc de Label pour pouvoir accéder aux mêmes méthodes. **La seule différence dans le calcul du coût** (ici on redéfinit une méthode *getTotalCost()*). On utilise la méthode ***distanceTo()*** entre deux points pour pouvoir prendre la distance entre le Node associé au LabelStar et le Node de destination. Voici **l’implémentation de LabelStar** :

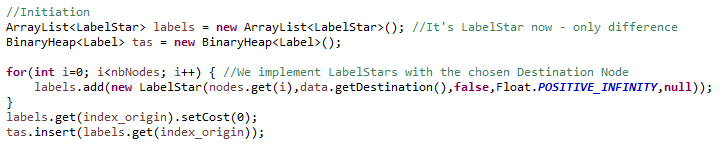


 Et voici le **diagramme UML associé à A\* et LabelStar** :

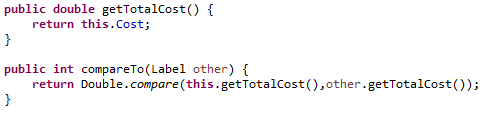
CurrentNode, Mark, Cost, Father

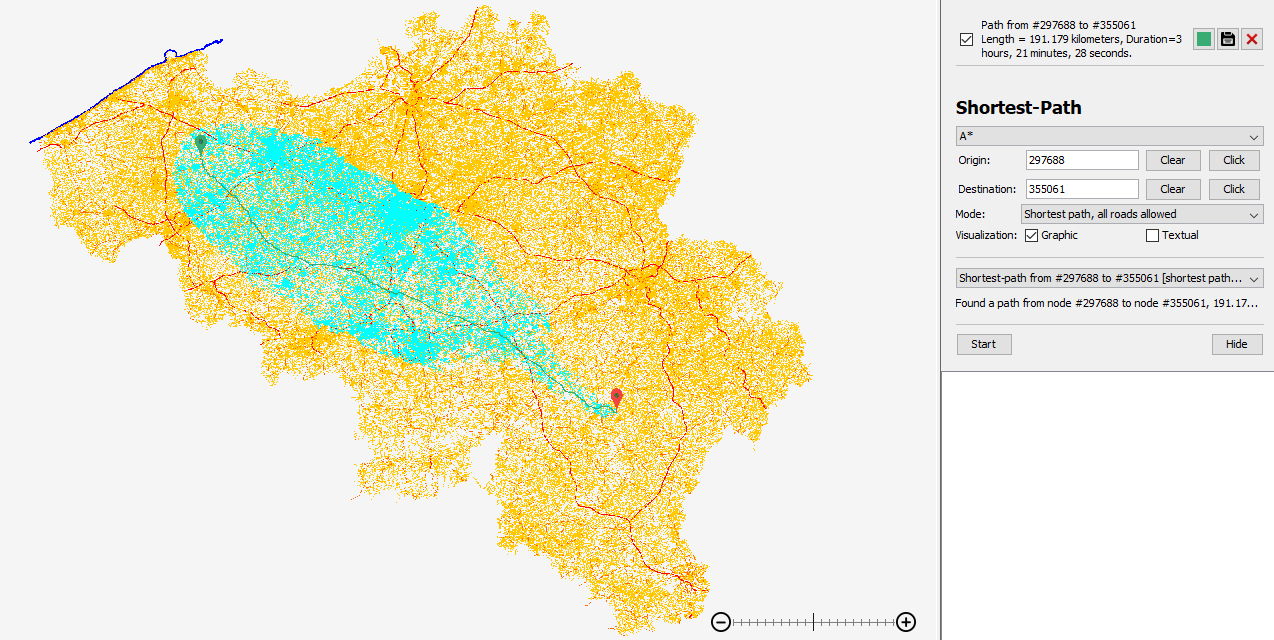
DistanceToDestination

L’algorithme **A\* est donc très similaire à Dijkstra**. Seulement trois lignes changent, pour prendre en compte l’ajout des LabelStar :

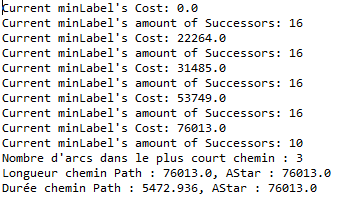


L’algorithme fonctionne exactement comme Dijkstra**, la différence est dans la méthode *compareTo()*** utilisée dans la classe *BinaryHeap* que nous avons implémentée, car je l’ai redéfinie dans Label pour qu’elle appelle ***getTotalCost()***. Dans Label évidemment cette méthode est la même que *getCost()*. En déclarant **le tas binaire de Labels dans A\* mais en l’implémentant avec des LabelStars**, quand on les compare pour sortir le plus petit élément (dans ce cas-ci, celui avec le coût le plus faible), on fera appel à la méthode ***compareTo()* de Label**, qui elle-même fera appel à *getTotalCost()* de LabelStar car l’élément à comparer est un LabelStar. Voici l’utilisation de *getTotalCost()* dans Label :



Voici un exemple du **déroulement de l’algorithme** sur la carte de Belgique :

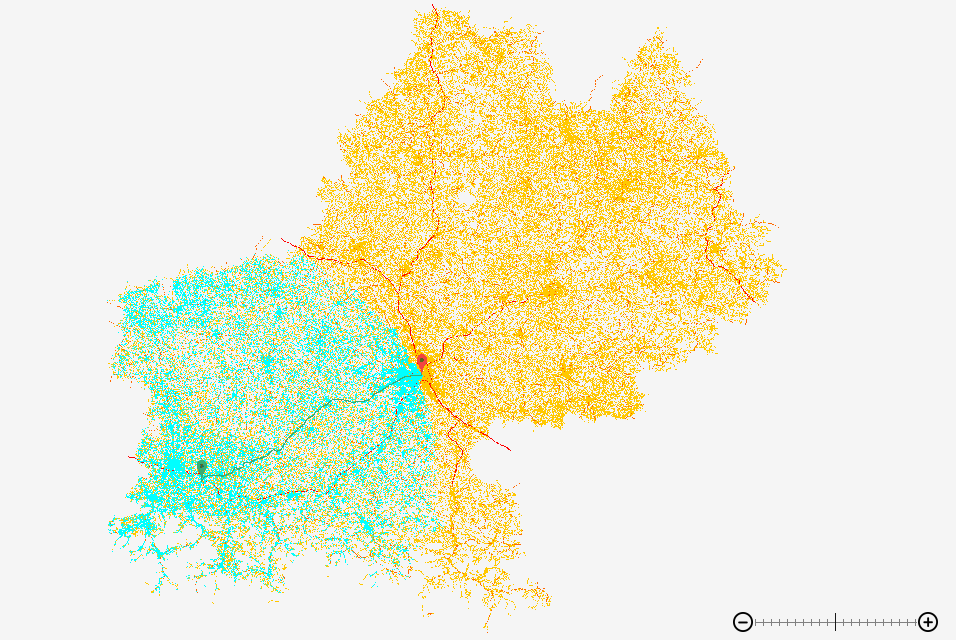
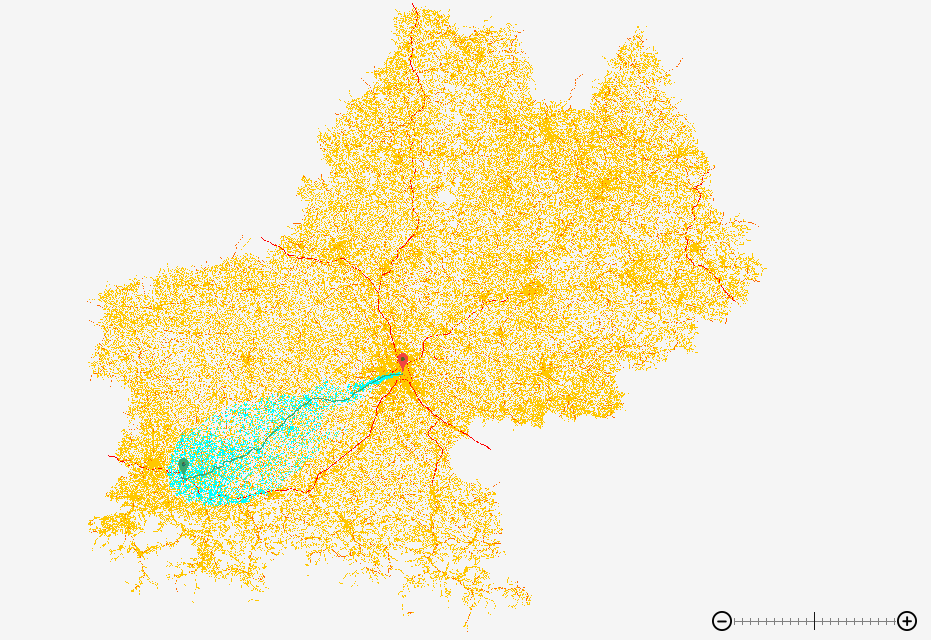
Il semble bien se comporter vis-à-vis des mes attentes, on voit qu’il **s’oriente vers la destination pendant sa recherche de sommet**s.

 J’ai aussi fait des tests en affichant les labels marqués au fur et à mesure, **leurs coûts sont croissants, ainsi que le nombre de successeurs explorés à chaque itération**.

On voit aussi sur le dernier affichage que **le chemin solution obtenu par A\*** est valide et que **la longueur du chemin est identique au plus court chemin de A\***. Comme Dijkstra, mon A\* **fonctionne comme attendu en distance et en temps**, **il n’y a pas de problème si il y a une erreur de saisie**, et **les solutions retournées sont valides**.

1. **Comparaisons Dijkstra/A\***

On peut déjà observer la différence entre les deux algorithmes **visuellement**:

A vue d’œil, **A\* semble être plus efficace que Dijkstra**. Il parcourt moins de Nodes, et son trajet orienté donne envie de dire que c’est le meilleur des deux. Je vais donc **comparer les deux algorithmes** en les faisant tourner sur les mêmes cartes, puis observer la moyenne d’iitérations et d’exécution de chacun. Je ferai aussi la distinction entre recherche PCC et chemin plus rapide.

Premier test : 5 trajets entre **5 et 15km**.

C’est flagrant, on voit que **sur des distances courtes A\* est bien plus performant que Dijkstra**, que ce soit en termes de temps d’exécution ou d’itérations effectuées.

2ème test : **25 à 50km**.

Il semblerait que A\* soit toujours meilleur que Dijkstra dans ce cas-ci, mais on voit **qu’en multipliant la distance parcourue par 5, le temps d’exécution de A\* a beaucoup augmenté** proportionnellement à celui de Dijkstra.

3ème test : **100 à 150km**.

En nombre d’itérations, nous avons toujours le même résultat, mais on peut voir que **sur des distances plus longues, A\* a un temps d’exécution plus important** !

4ème test : Grandes distances, c.à.d. **supérieures à 200km**.

C’est maintenant flagrant, A\* ralentit d’un facteur bien plus important que Dijkstra quand la distance augmente. En conclusion, on peut dire **que A\* sera toujours meilleur que Dijkstra en termes de moyenne d’itérations** (même si le ratio semble augmenter petit à petit) mais que **sur des distances de plus en plus grandes, A\* devient très rapidement lent**, voire inutilisable. Tout cela en termes de PCC bien sûr.

Je vais maintenant faire les mêmes tests en mode chemin plus rapide.

1er test : **5 à 15km**.

On dirait bien que **Dijkstra est plus adapté à un parcours en vitesse de petits chemins** (en termes de temps d’exécution). Pour ce qui est des itérations, on observe les mêmes résultats qu’en distance.

2ème test : **25 à 50km**.

On voit qu’en augmentant un peu la distance parcourue, l’écart entre les itérations effectuées par les deux algorithmes se referme un peu. **A\* augmente en itérations plus vite que Dijkstra**. Pour ce qui est du temps d’exécution, l’écart s’est plutôt creusé. **Comme dans le cas précédent, A\* s’avère de plus en plus lent quand les distances augmentent**.

3ème test : **100 à 150km**.

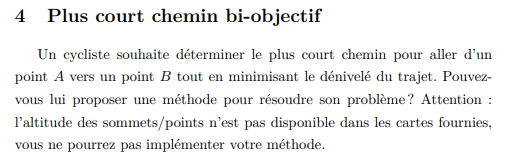
C’est clair et net, sur de longues distances, Dijkstra est bien plus rapide, mais demandes plus d’itérations avant d’arriver à destination que son rival A\*.

Nous pouvons donc en déduire que A\* est plus intéressant en distance qu’en temps, pour calculer le meilleur chemin. On peut expliquer cela par le fait que la borne inférieure fournie par A\* est obtenue par le coût à vol d’oiseau, qui correspond plus au chemin le plus court que le plus rapide (à vol d’oiseau on aura tendance à aller droit au but, en essayant de suivre une ligne droite, alors que le chemin le plus rapide peut prendre de nombreux ‘’virages’’).

**Conclusion**

Truc

Partie Problème Ouvert



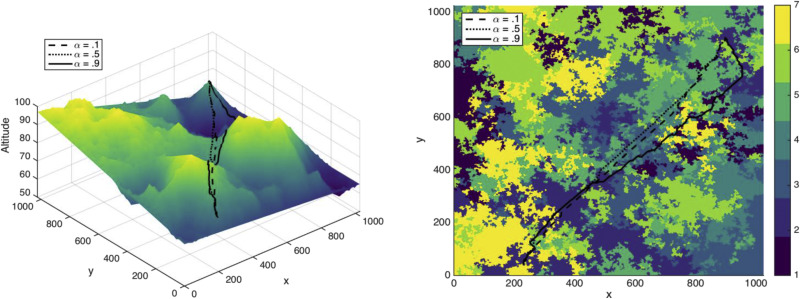
**Introduction**

Etant un grand fan de VTT, ce sujet m’a fort intéressé. Je tiens d’abord dire que limiter le dénivelé d’un trajet à vélo va à l’encontre du sport – le vélo c’est la côte et la côte c’est le vélo. J’aborderai donc ce sujet avec le but d’aider les jeunes débutants qui souhaitent économiser leurs mollets.

Il semblerait au premier abord que ce problème soit un sujet de **graphes à trois dimensions** – longitude, latitude et altitude. Je fais donc l’hypothèse dans ma conception que **l’altitude d’un point est référencée dans ses paramètres**, et qu’en conséquence un arc a une valeur d’élévation positive si l’arc passe d’une altitude plus basse à une altitude plus haute et vice-versa.

Ensuite, pour définir le chemin optimal il faudra que le chemin rendu approche le plus court chemin rendu par Dijkstra (ou A\*, l’idée est de prendre le chemin de distance optimale comme référence). Je prends comme valeur arbitraire une **tolérance de 10%** - que l’on peut adapter selon l’importance du dénivelé vs la distance - sur la distance totale de la solution de mon algorithme par rapport à la distance optimale.

Voici à peu près ce qu’on aimerait obtenir :

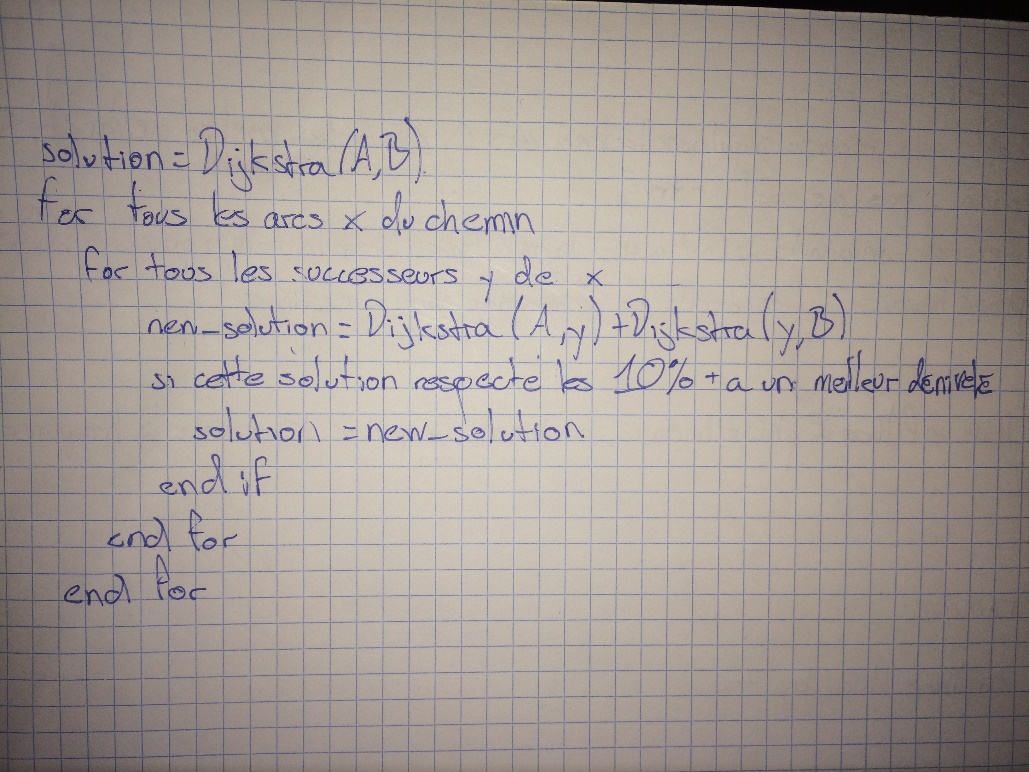


1. **Conception papier**

Le but de mon algorithme sera de **minimiser le dénivelé** de mon trajet le plus possible. Pour minimiser le dénivelé, je prends la différence d’élévation entre les points A et B, puis je souhaite m’en approcher le plus possible en minimisant la somme des valeurs absolues de l’élévation de chaque arc emprunté par mon chemin. On aimerait, si possible, obtenir :

Avec di le dénivelé du ième arc, et dA et dB les altitudes des points de départ et d’arrivée.

Ma **première idée** était de parcourir tous les chemins qui augmentent la distance totale de 10% maximum, comparer le dénivelé proposé à celui du chemin, et garder les arcs qui économisent le plus de dénivelé. Voici ce premier algorithme sur papier :



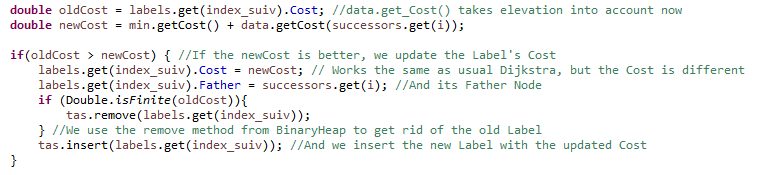
Malheureusement, **cet algorithme ne donne pas la solution optimale** : imaginons qu’une boucle parallèle au PCC nous permette de trouver un chemin avec un meilleur dénivelé : l’algorithme ne le parcourrait pas entièrement. Il passerait par le premier Node successeur au chemin, puis reprendrait la direction du PCC.

En plus, la complexité de Dijkstra est de **O((m+n).log n)** (n étant le nombre de sommets et m les successeurs de chaque sommet). Donc en l’appelant à l’intérieur de deux boucles for, je me retrouve avec un algorithme en **O(m.n.(m+n).log n)**, ce qui doit au moins **tripler le temps d’exécution de mon algorithme.**

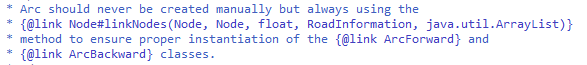
En me penchant donc plus sur le problème, j’ai donc décidé d’opter pour un **Dijkstra en 3D**, c’est-à-dire que le **coût de chaque Arc** dépend aussi maintenant de **l’élévation**. La moyenne entre la distance et l’élévation se ferait avec un certain pourcentage ajustable, selon l’utilisateur : souhaite-t-il/elle plus valoriser la distance parcourue ou la côte à monter, etc.

1. **Algorithme**

L’algorithme est donc **pareil que Dijkstra mais la différence est dans le coût des Arcs**. Donc à cet endroit dans Dijkstra :



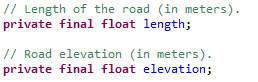
On fait en fait appel à la fonction ***getCost()*** de la classe *ShortestPathData* qui elle-même implémente la classe *AbstractInputData* qui elle-même appelle la méthode *getCost()* de *ArcInspector* qui elle appelle la méthode ***getLength()*** de l’Arc fourni. Or, un Arc est créé en reliant un *ArcForward* et un *ArcBackward*:



Et c’est enfin dans la méthode *getLength()* de *ArcForward* qu’on voit que la **longueur d’un Arc est un paramètre spécifié** par la Carte donnée, car elle est elle-même un ensemble d’Arcs :

Il nous suffit de **changer *ArcForward*** comme ceci :

**Et l’algorithme de Dijkstra déjà codé fera le reste**. C’est le même principe qu’en codant A\*, l’algorithme prendre la classe mise à jour et adaptera son comportement. Dans mes explications, j’ai souvent dit que le dénivelé devrait être moins important que la distance ( j’imaginais le facteur d’élévation à 20% plutôt), mais j’ai choisi de mettre un paramètre ajustable dans *getLength()*, comme ça l’utilisateur choisit. **En prenant un facteur d’élévation égal à 1** cependant, on aura donné autant d’importance aux deux, ce qui risque de faire **des chemins** qui - à vol d’oiseau – auront l’air **bien différents du PCC**.

Voici une bonne **visualisation** du déroulement de l’algorithme:

<http://sruzek.github.io/algorithms/dijkstras.html>

1. **Solution optimale et Complexité**

Mon algorithme Dijkstra 3D **donne bien la solution optimale** car c’est une **adaptation directe de Dijkstra**. Le coût associé à chaque sommet est actualisé à chaque Node marqué, le concept est toujours le même. La solution optimale avec un facteur d’élévation de 1 ou de 0,5 ne sera évidemment pas la même. C’est **donc l’utilisateur qui définira les conditions d’optimalité de cet algorithme**.

L’ajout du dénivelé dans le coût ne change en rien la complexité, les accès au tas sont toujours les mêmes. La **complexité de cet algorithme** est donc aussi **O((m+n).log n)**. Sachant que l’on peut toujours **l’améliorer en A\*, le principe est le même**, notre classe LabelStar aura juste besoin d’une implémentation supplémentaire pour prendre en compte les nouveaux coûts en changeant la classe *distanceTo()* d’un point.

**Conclusion**

Trouver la distance la plus courte en prenant aussi en compte le dénivelé du chemin est en fait un problème de **plus court chemin en 3 dimensions**. Une fois que j’ai fait tilt, la solution était évidente. **Dijkstra est donc un algorithme capable de s’adapter à n dimensions.** Sa complexité est donc équivalente à celle de Dijkstra et il donne la solution optimale. Il ne reste plus qu’à l’implémenter. Mais pour cela il me faudrait des cartes 3D en accordance avec le projet de BE donné.

On peut enfin **se poser plus de questions** par rapport au sujet : Cela vaut-il le coup d’emprunter un grand nombre de petits chemins (et donc faire beaucoup de détours) pour prendre le chemin le plus « plat » possible par rapport au chemin à vol d’oiseau? Faudrait-il aussi implémenter une option qui permet de prendre des chemins plus panoramiques? Car si on développe un tel algorithme pour les amateurs de vélo, il faut se poser la question du point de vue de l’utilisateur ; qu’est-ce qu’un cycliste préfère ?

Et à ça, je pense avoir la réponse :