

**Rapport BE Graphes**

**Théau Giraud**

**29/04/2020**

Partie BE Graphes

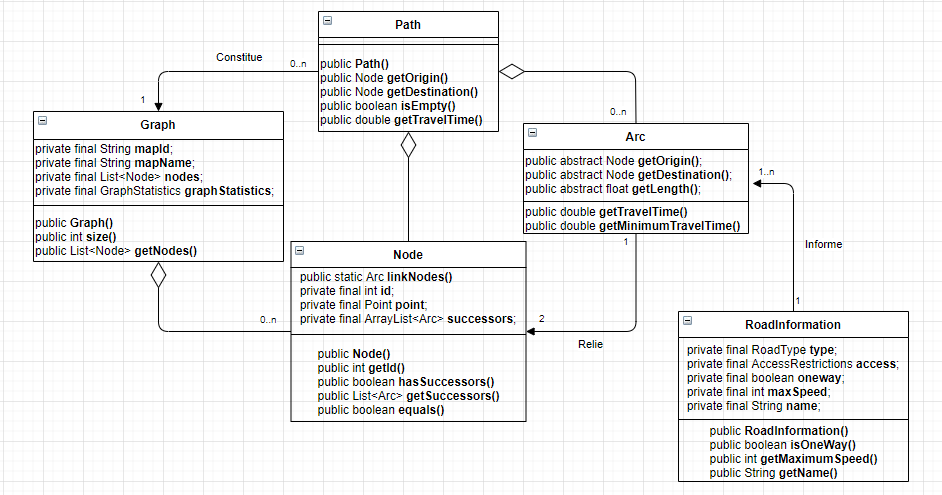
**Introduction**

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.

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 *Graphe* 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.

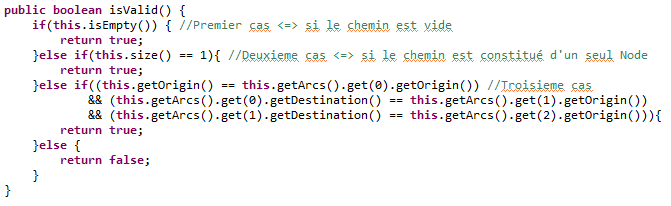
Voici un diagramme UML des différentes classes données, avec les méthodes et attributs les plus importants en gras :

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

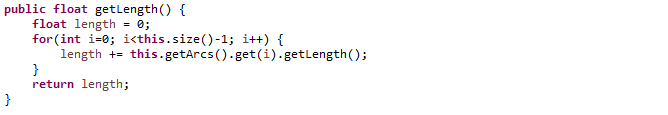
1. **Implémentation de Path()**

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 :

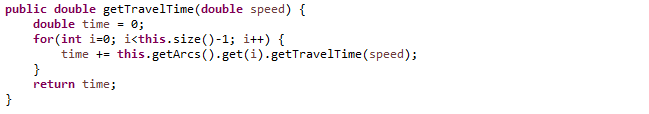
*isValid()* renvoie *True* si le Path crée est valide. Cela est vrai pour plusieurs cas possibles ; si le Path est vide, si il contient un seul Node et si il contient au moins deux Nodes, il faut que le premier Arc ait pour origine l’origine du Path et qu’au moins deux Arcs consécutifs dans le Path créé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



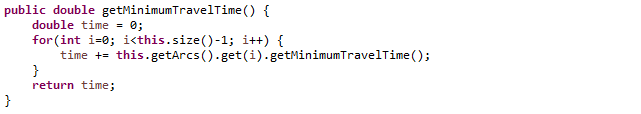
*getLength()* renvoie la longueur du Path tout entier. C’est une simple somme sur la longueur des Arcs le constituant.



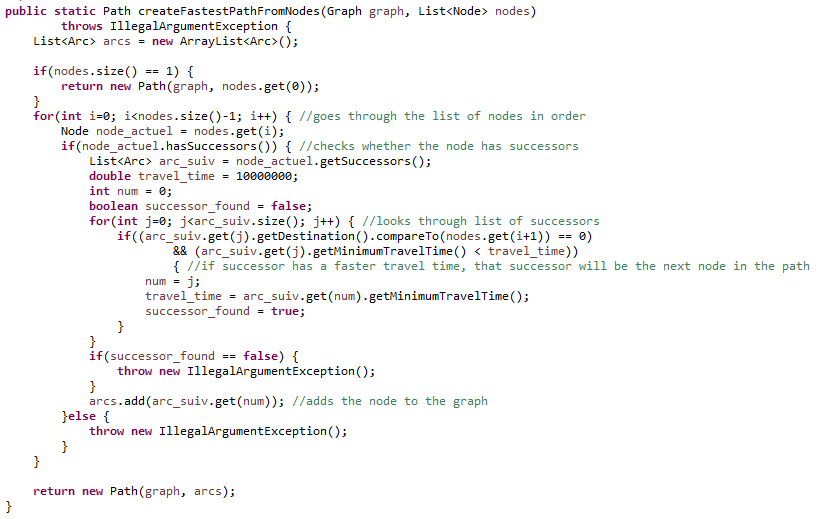
*getTravelTime(double speed)* renvoie le temps que l’on prendra pour parcourir le Path avec une vitesse (*speed*) constante donnée. C’est ici aussi une somme.



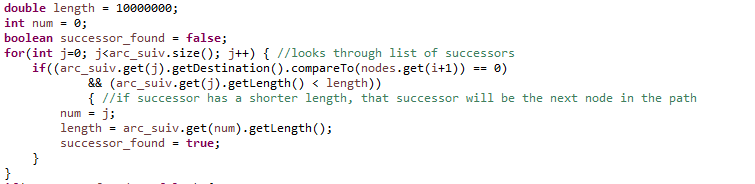
*getMinimumTravelTime()* renvoie le temps pris pour parcourir le Path si on parcourt le Path avec la vitesse maximale de chaque Arc. J’ai ici utilisé la méthode *getMinimumTravelTime()* de la classe Arcs qui renvoie déjà le temps minimum de chaque Arc à vitesse maximale.



*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. 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()*:

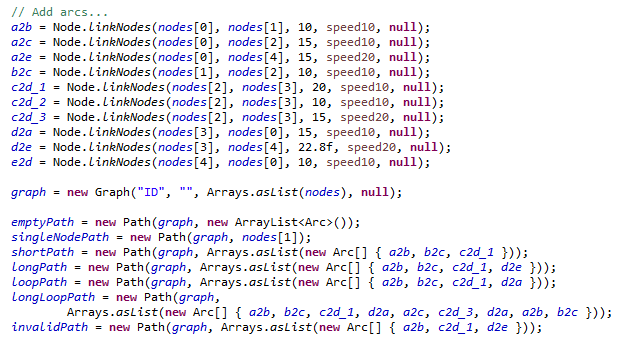


1. **Tests JUnit**

Pour vérifier si les méthodes codées étaient correctes, nous avons eu recours à des tests *JUnit*. Ce sont des tets 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 prendre en compte tous ces cas dans le corps des méthodes et s’en aider pour les implémenter. Certaines 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 tets pour créer la méthode.

1. **Binary Heap/implémentation de remove()**

Truc

1. **Compréhension Bellman-Ford/UML**

Truc

1. **Label/Dijkstra**

Truc

1. **Vérification Dijkstra**

Truc

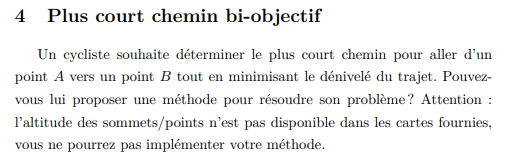
1. **A\***

Truc

**Conclusion**

Truc

Partie Problème Ouvert



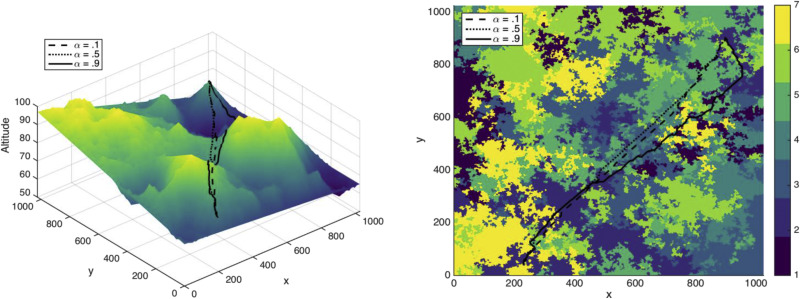
**Introduction**

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. **Complexité du problème**

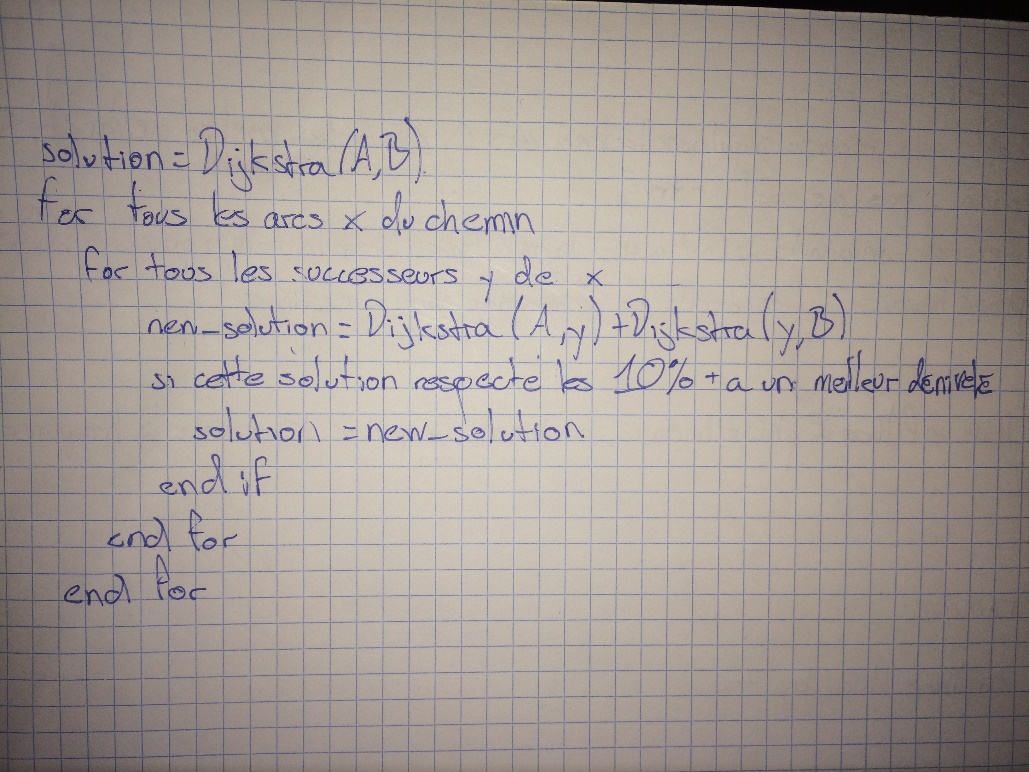
Truc

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**

Cet algorithme ressemble donc à ceci :

[Photo Dijsktra jouli]

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, c’est toujours le même concept.

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 coût par rapport à la destination finale peut aussi être adapté avec l’altitude.

**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** (sachant que j’ai choisi ici de donner moins d’importance au dénivelé qu’à la distance). Sa complexité est donc équivalente à celle de Dijkstra et il donne la solution optimale. Il ne reste plus qu’à l’implémenter!

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 ? 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 que j’ai **la réponse** :

[Photo]