INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

135, Avenue de Rangueil
31077 Toulouse Cedex 4

Rapport de Bureau d'Etudes

POO et Graphes

Date de rendu : 02 Juin 2020

HOK Jean-Rémy
INSA* Promotion 55, 3ème Année MIC*

*INSA : Institut National des Sciences Appliquées
*MIC : Modélisation, Informatique et Communication

INSA Toulouse
135, Avenue de Rangueil
31077 Toulouse Cedex 4

Rapport de Bureau d'Etudes

POO et Graphes

Date de rendu : 02 Juin 2020

HOK Jean-Rémy
INSA* Promotion 55, 3ème Année MIC*

*INSA : Institut National des Sciences Appliquées
*MIC : Modélisation, Informatique et Communication

Table des matières

Introduction

Dans le cadre de l'enseignement « Graphes », le bureau d'études a eu pour but principal la découverte des algorithmes de recherche du plus court chemin (en distance ou en temps).

Ce bureau d'études peut être divisé en 4 étapes distinctes : tout d'abord la modélisation des chemins, avec les classes correspondantes ; puis l'implémentation des algorithmes de recherche du plus court chemin ; suivi d'une amélioration vers un algorithme plus performant ; et enfin, des tests permettant de valider les solutions trouvées par les différents algorithmes, et évaluer leur performance.

Cette implémentation a pu être possible grâce à l'outil principal que nous avons utilisé à savoir l'environnement de développement Eclipse, permettant de compiler nos programmes écrits dans le langage de programmation java. Le code produit au cours des séances est accessible sur un dépôt git via ce lien: https://github.com/PiKouri/BE-Graphes.

A noter qu'une interface graphique permettant de visualiser la progression de l'exploration de chaque algorithme de recherche et des cartes et leurs chemins préalablement créés nous ont été mis à disposition.

I - Modélisation des chemins

Tout d'abord, avant de se lancer dans l'implémentation des algorithmes de recherche, nous avons dû modéliser les chemins reliant les différents points du graphe. Nous avons donc travaillé sur le package ***graph***, contenant les **classes *Node, Graph, Path, Arc et RoadInformation*** qui sont présentées ci-dessous.

Diagramme UML des classes du package Graph

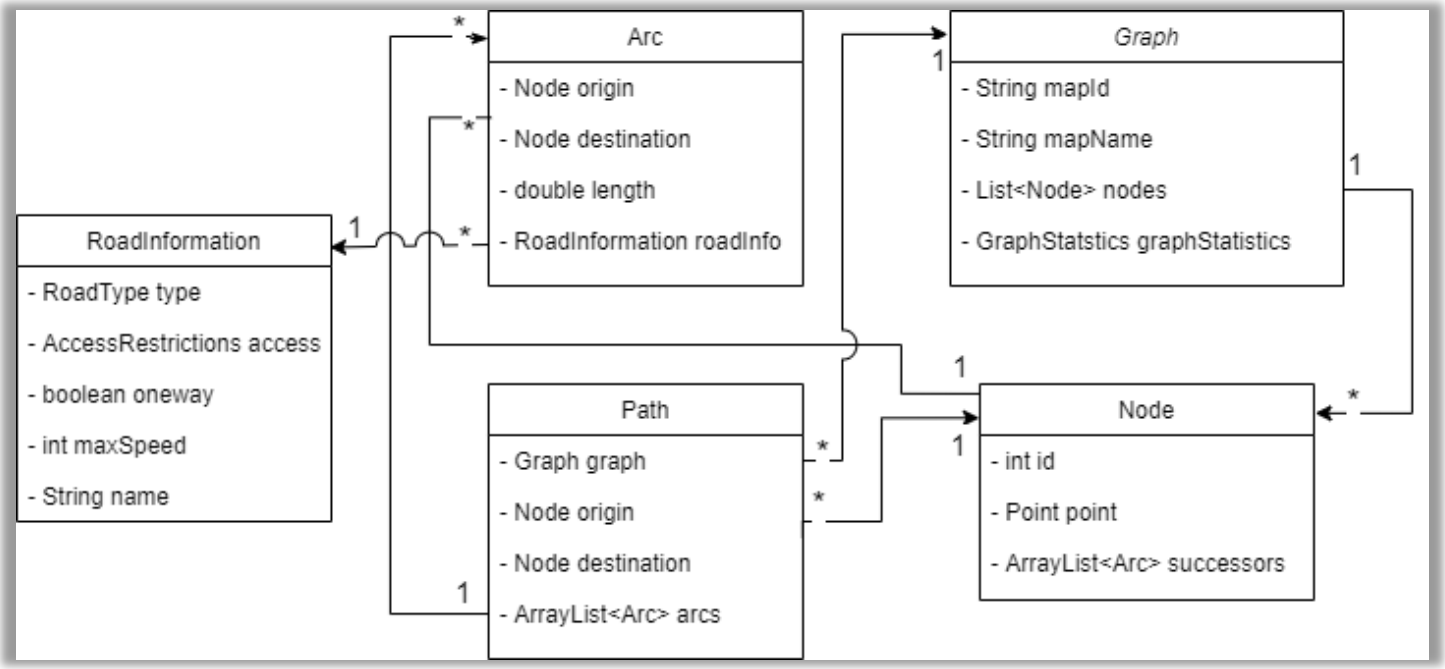

Figure 1: Diagramme UML du package graph

Explication du package graph

Le package graph dans l'ensemble

Le diagramme UML ci-dessus permet d'avoir une vue d'ensemble sur les divers éléments d'un graphe. Premièrement, le ***Graph*** : c'est la carte dans laquelle nous cherchons le plus court chemin d'un nœud à l'autre. Ensuite, le ***Node*** (nœud), c'est un point de la carte, caractérisé par son ***id*** (identifiant) et qui contient l'information de ses arcs ***successors*** (successeurs). Les ***Arcs*** quant à eux, relient deux nœuds, ils contiennent donc une information de distance (***length***). A noter que chaque arc a également une ***RoadInformation*** (information de route) qui permet de déterminer les caractéristiques de la portion de route modélisée par cet arc en question : notamment, la vitesse maximale autorisée, le type de route (rue piétonne, piste cyclable ou route réservée aux voitures) et d'autres restrictions (comme les voies uniques). Enfin, nous avons le ***Path*** qui représente un chemin/une route d'un point ***origin*** à un point ***destination*** et qui contient une liste d'***Arcs*** qui relient deux à deux des points intermédiaires permettant de former ce chemin. Nous allons par la suite détailler cette classe.

La classe Path en détail

Comme expliqué précédemment, nous avons implémenté les fonctions nécessaires à l'utilisation d'objets de la classe ***Path***: en effet, étant donné que nous cherchons le chemin le plus court, il semble indispensable de pouvoir déterminer la **longueur d'un chemin** et également le **temps de parcours** de celui-ci. Pour le temps de parcours nous avons deux fonctions différentes: la première permet de calculer le temps nécessaire si l'on parcourt le chemin à une certaine vitesse constante, tandis que la deuxième permet de calculer le temps minimum de trajet, en utilisant les vitesses maximales de chaque portion composant le chemin. Ensuite, étant donné que le chemin est implémenté par une liste d'arcs, il est important de vérifier la **validité** du chemin : un chemin est valide s'il est vide, s'il contient un seul nœud, ou si le premier arc part du nœud origine du chemin et que pour deux arcs consécutifs, la destination du premier est l'origine du second.

Ainsi nous pouvons récupérer pour chaque chemin:

- Sa longueur avec ***getLength()***

```
public float getLength() {
    float length = 0;
    if (this.isEmpty()) return 0;
    else {
        for (Arc a: this.getArcs()) {
            length += a.getLength();
        }
    }
    return length;
}
```

Figure 2 : classe Path, fonction getLength()

- Son temps de parcours avec ***getTravelTime()*** qui prend la vitesse en argument

```
public double getTravelTime(double speed) {
    return (this.getLength()*3.6/speed);
}
```

Figure 3 : classe Path, fonction getTravelTime()

- Son temps de parcours minimum avec ***getMinimumTravelTime()***

```
public double getMinimumTravelTime() {
    double time = 0;
    if (this.size()==0) return 0;
    else {
        for (Arc a: this.getArcs()) {
            time += a.getMinimumTravelTime();
        }
    }
    return time;
}
```

Figure 4 : classe Path, fonction getMinimumTravelTime()

- Sa validité (ou invalidité) avec ***isValid()***

```java
public boolean isValid() {
    boolean cond = false;
    List<Arc> listArcs = this.getArcs();
    if (this.isEmpty() || (listArcs.isEmpty() && this.size()==1)) cond = true;
    else if (this.getOrigin() == listArcs.get(0).getOrigin()) {
        if (this.size()==2) cond = true;
        else {
            Node prevDest = listArcs.get(0).getDestination();
            List<Arc> sublistArcs = listArcs.subList(1, listArcs.size());
            // Parcours de la liste des arcs à partir du second pour comparer au précédent
            for (Arc a: sublistArcs) {
                cond = (prevDest==a.getOrigin());
                prevDest = a.getDestination();
                if (cond == false) break;
            }
        }
    }
    return cond;
}
```

Figure 5 : classe Path, fonction isValid()

Enfin, nous pouvons créer le chemin le plus court et le chemin le plus rapide de manière similaire avec les fonctions respectives ***createShortestPathFromNodes()*** et ***createFastestPathFromNodes()***. Ces fonctions prennent un graphe et une liste de nœuds en arguments, ainsi elle parcourt la liste de nœuds (deux nœuds consécutifs doivent être reliés par un arc), et pour chaque nœud, mémorise l'arc vers le prochain nœud de la liste possédant la distance (ou le temps) minimale.

```java
public static Path createShortestPathFromNodes(Graph graph, List<Node> nodes)
        throws IllegalArgumentException {
    List<Arc> arcs = new ArrayList<Arc>();
    if (nodes.size() > 1) {
        List<Node> sublistNodes = nodes.subList(1,nodes.size());
        Node noeud1 = nodes.get(0);
        for (Node noeud2 : sublistNodes) {
            float lengthMin = Float.MAX_VALUE;
            Arc arcMin = null;
            for (Arc arcActuel : noeud1.getSuccessors()) {
                if (arcActuel.getDestination()==noeud2 && arcActuel.getLength()<lengthMin) {
                    arcMin = arcActuel;
                    lengthMin = arcActuel.getLength();
                }
            }
            if (arcMin != null) arcs.add(arcMin);
            else throw new IllegalArgumentException("Problem with list nodes");
            noeud1 = noeud2;
        }
    } else if (nodes.size() == 0) return new Path(graph);
    else return new Path(graph, nodes.get(0));
    return new Path(graph, arcs);
}
```

Figure 6 : classe Path, fonction createShortestPathFromNodes()

La fonction ***createFastestPathFromNodes()*** est similaire à la différence que l'on n'utilise non pas ***getLength()*** mais ***getMinimumTravelTime().*** Il est important de noter que l'exception ***IllegalArgumentException*** est levée si deux nœuds consécutifs ne sont pas connectés dans le graphe.

```
public static Path createFastestPathFromNodes(Graph graph, List<Node> nodes)
        throws IllegalArgumentException {
    List<Arc> arcs = new ArrayList<Arc>();
    if (nodes.size() > 1) {
        List<Node> sublistNodes = nodes.subList(1,nodes.size());
        Node noeud1 = nodes.get(0);
        for (Node noeud2 : sublistNodes) {
            double timeMin = Float.MAX_VALUE;
            Arc arcMin = null;
            for (Arc arcActuel : noeud1.getSuccessors()) {
                if (arcActuel.getDestination()==noeud2 && arcActuel.getMinimumTravelTime()<timeMin) {
                    arcMin = arcActuel;
                    timeMin = arcActuel.getMinimumTravelTime();
                }
            }
            if (arcMin != null) arcs.add(arcMin);
            else throw new IllegalArgumentException("Problem with list nodes");
            noeud1 = noeud2;
        }
    } else if (nodes.size() == 0) return new Path(graph);
    else return new Path(graph, nodes.get(0));
    return new Path(graph, arcs);
}
```

Figure 7 : classe Path, fonction createFastestPathFromNodes()

Synthèse des tests JUnit

Une fois toutes les méthodes nécessaires implémentées, l'étape finale de la classe ***Path*** a été la phase de test : nous nous sommes assurés de la validité des méthodes de la classe ***Path*** en l'exposant à un jeu de tests fourni. Ce jeu de tests JUnit couvre la majorité des cas possibles, comme un chemin vide, un chemin composé d'un seul nœud, un chemin invalide, un chemin bouclé, ou tout simplement un chemin « standard » court ou long.

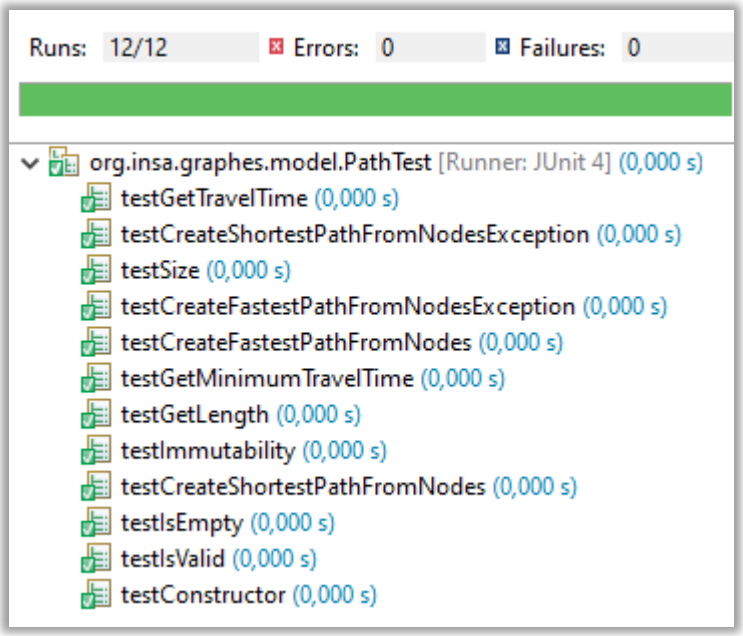

Figure 8 : classe Path, JUnit Tests

II – Algorithmes du plus court chemin

Après avoir implémenté la classe ***Path***, nous sommes passés à l'implémentation des algorithmes du plus court chemin : à savoir dans cette partie, l'algorithme de Bellman-Ford et l'algorithme de Dijkstra.

Explication du fonctionnement des algorithmes

Algorithme de Bellman-Ford

L'algorithme de recherche du plus court chemin de Bellman-Ford (ou aussi appelé Moore-Bellman-Ford) a notamment pour particularité de prendre en compte la présence d'arcs de poids négatif et de détecter la présence de circuits absorbants (poids total < 0).

Initialisation $Cost\_0(x) = 0$; $Cost\_0(x) = \infty$, pour tout $x \neq s$

```
For tous les sommets i  de 1 à n
  Cost(i) ← +∞
  Father(i) ← 0 // sommet inexistant
end for

Cost(s) ← 0
k ← 0 // pour compter les itérations
```

Figure 9 : Bellman-Ford, pseudo-code Initialisation
Source : Support de cours de Graphes de 3MIC INSA Toulouse de Marie-Jo Huguet

Cet algorithme associe à chaque sommet (nœud) une étiquette mémorisant le coût pour atteindre ce sommet et également son père, c'est-à-dire le sommet qui le précède.

A l'initialisation tous les sommets ont un coût infini et un père inexistant, sauf le sommet d'origine qui a un coût nul.

Itérations $Cost\_k(x) = MIN\ (Cost\_k\text{-}1(x), Cost\_k\text{-}1(y) + W(y, x)\ ),$ pour tout $y \in \delta^-(x)$

```
repeat
  continuer ← false
  k ← k+1
  For tous les sommets x (/= de s)
      for tous les prédecesseurs y de x
       Cost(x) ← Min(Cost(x), Cost(y)+W(y,x))
       if Cost(x) a été mis à jour then
         Father(x) ← y
         continuer ← true
       end if
      end for
  end for
until (k=n) or (not continuer)
```

Figure 10 : Bellman-Ford, pseudo-code Itération
Source : Support de cours de Graphes de 3MIC INSA Toulouse de Marie-Jo Huguet

A chaque itération, nous parcourons tous les prédécesseurs de tous les sommets et pour chaque sommet, on met à jour le coût et le père lorsqu'un arc de poids plus faible que celui étiqueté est trouvé.

L'algorithme s'arrête lorsqu'il n'y a plus de mis à jour de poids (chemin le plus court trouvé) ou en cas de circuit absorbant.

Algorithme de Dijkstra

L'algorithme de recherche du plus court chemin de Dijkstra quant à lui, ne peut être utilisé sur un graphe avec des arcs de poids négatif.

Initialisation

```
For tous les sommets i  de 1 à n
  Mark(i) ← Faux
  Cost(i) ← +∞
  Father(i) ← 0 // sommet inexistant
end for

Cost(s) ← 0
Insert(s, Tas)
```

Figure 11 : Dijkstra, pseudo-code Initialisation
Source : Support de cours de Graphes de 3MIC INSA Toulouse de Marie-Jo Huguet

Cet algorithme associe également à chaque sommet (nœud) une étiquette mémorisant le coût pour atteindre ce sommet et également son père, c'est-à-dire le sommet qui le précède mais aussi son marquage : un sommet est marqué quand son coût minimum a été trouvé. Le tas binaire (ou file de priorité) est la structure utilisée pour choisir le nœud à explorer.

A l'initialisation tous les sommets ont un coût infini, un père inexistant et sont non-marqués, sauf le sommet d'origine qui a un coût nul. Le seul nœud dans le tas à l'initialisation est le sommet d'origine.

Itérations

```
While il existe des sommets non marqués
  x ← ExtractMin(Tas)
  Mark(x) ← true
  For tous les y successeurs de x
    If not Mark(y) then
      Cost(y) ← Min(Cost(y), Cost(x)+W(x,y))
      If Cost(y) a été mis à jour then
         Placer(y, Tas)
         Father(y) ← x
      end if
    end if
  end for
end while
```

Figure 12 : Dijkstra, pseudo-code Itération
Source : Support de cours de Graphes de 3MIC INSA Toulouse de Marie-Jo Huguet

A chaque itération, nous récupérons le nœud racine ***x*** du tas (avec le coût minimum) que l'on marque et nous parcourons tous ses successeurs non-marqués ***y***, on met à jour le coût et le père de ***y*** lorsque le coût de ***x*** + le poids de l'arc ***(x, y)*** est inférieur au coût de ***y***. Ensuite nous plaçons le nœud ***y*** dans le tas.

L'algorithme s'arrête lorsqu'il n'y a plus de sommet non-marqué (tas vide) ou que le nœud extrait du tas est la destination.

Diagramme UML des classes du package Algo et ShortestPath

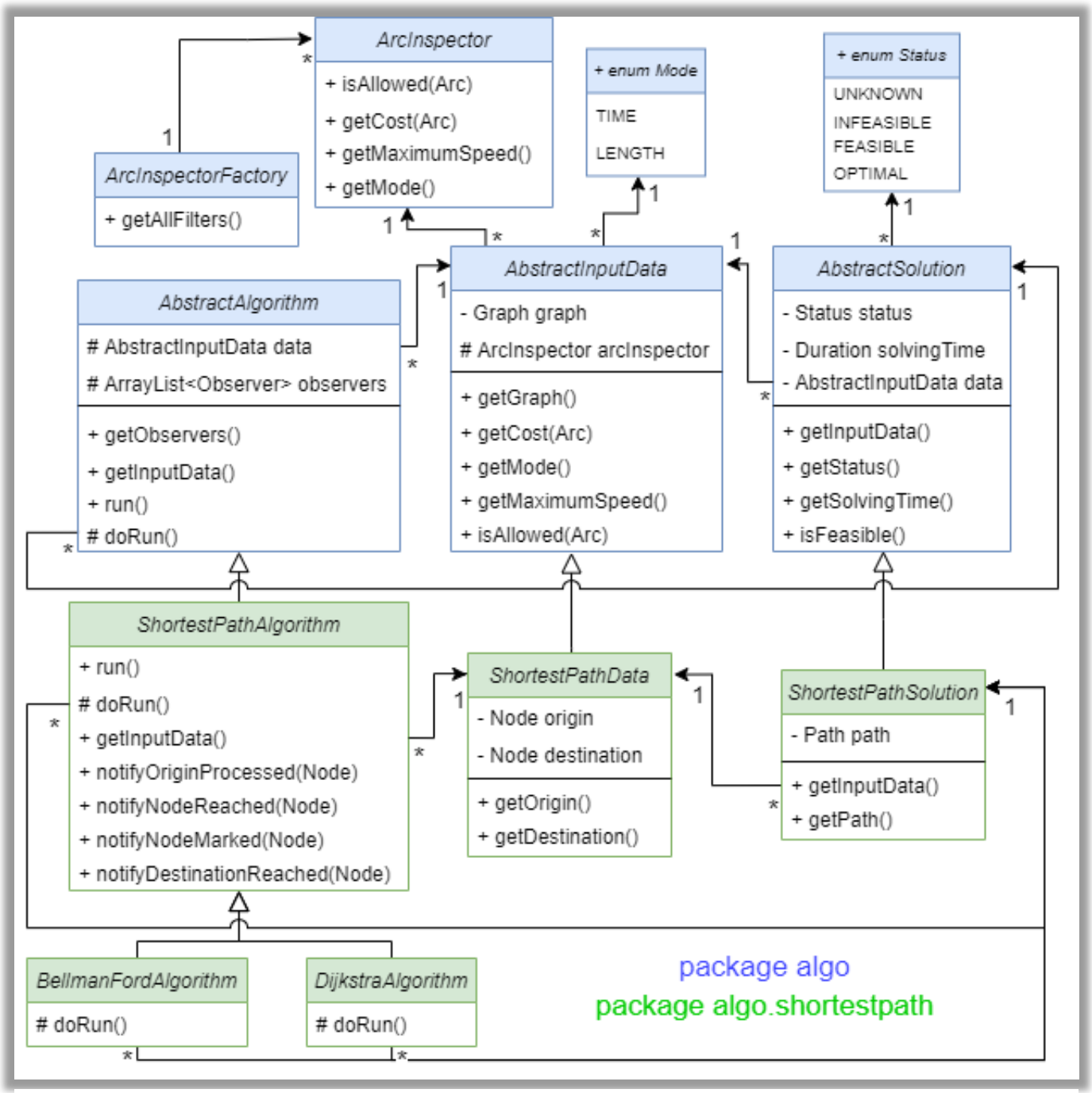

Figure 13 : Diagramme UML des packages algo et algo.shortestpath

L'algorithme de Bellman-Ford étant déjà implémenté, nous nous sommes concentrés sur l'implémentation de celui de Dijkstra, nécessitant l'implémentation d'une classe ***BinaryHeap*** (tas) et d'une classe ***Label*** (étiquette).

Classe BinaryHeap : file de priorité

Explication et implémentation du tas binaire

Le tas binaire ou file de priorité est une structure de données qui peut être implémentée par un arbre binaire dont tous les niveaux à l'exception du dernier doivent être complètement remplis, et dont les sommets respectent une certaine relation d'ordre : dans notre cas, c'est un tas minimum, donc le sommet racine doit être l'élément ayant la plus petite valeur (ici le coût étiqueté), et chaque sommet doit avoir une valeur inférieure ou égale à celle de ses fils.

La classe ***BinaryHeap*** est l'implémentation de cette structure : c'est une classe générique qui peut donc être instanciée pour stocker divers éléments. Dans notre cas, le tas contiendra des ***Labels*** que l'on détaillera ultérieurement. Chaque objet de cette classe est caractérisé par la taille courante du tas et d'une ***ArrayList<E>***, tableau contenant les éléments du tas. Dans la suite, nous utilisons les fonctions ***percolateUp(int index)*** et ***percolateDown(int index)*** qui mettent à jour le tas à partir du sommet donné respectivement en montant et en descendant dans l'arbre.

Cette classe contient les méthodes suivantes:

- ***isEmpty()*** qui permet de déterminer si le tas est vide ou non

```
public boolean isEmpty() {
    return this.currentSize == 0;
}
```

Figure 14 : BinaryHeap, isEmpty()

- ***insert(E x)*** qui permet d'insérer un élément x dans le tas

```
public void insert(E x) {
    int index = this.currentSize++;
    this.arraySet(index, x);
    this.percolateUp(index);
}
```

Figure 15 : BinaryHeap, insert()

- ***arraySet(int index, E value)*** qui permet de modifier un élément du tas

```
private void arraySet(int index, E value) {
    if (index == this.array.size()) {
        this.array.add(value);
    }
    else {
        this.array.set(index, value);
    }
}
```

Figure 16 : BinaryHeap, arraySet()

- ***indexParent(int index)*** qui retourne l'index du parent du sommet donné

```
protected int indexParent(int index) {
    return (index - 1) / 2;
}
```

Figure 17 : BinaryHeap, indexParent()

- ***indexLeft(int index)*** qui retourne l'index du fils gauche du sommet donné

Figure 18 : BinaryHeap, indexLeft()

```
protected int indexLeft(int index) {
    return index * 2 + 1;
}
```

- ***findMin()*** qui renvoie le sommet racine (minimum) du tas s'il n'est pas vide, et lève l'exception ***EmptyPriorityQueueException*** sinon

```
public E findMin() throws EmptyPriorityQueueException {
    if (isEmpty())
        throw new EmptyPriorityQueueException();
    return this.array.get(0);
}
```

Figure 19 : BinaryHeap, findMin()

- ***deleteMin()*** qui supprime et renvoie le sommet racine, et actualise le tas après la suppression (étant donné que nous sommes tout en « haut » de l'arbre, nous utilisons ***percolateDown()***). A noter que nous utilisons ***findMin()*** pour récupérer le sommet racine et vérifier en même temps que le tas n'est pas vide.

```
public E deleteMin() throws EmptyPriorityQueueException {
    E minItem = findMin();
    E lastItem = this.array.get(--this.currentSize);
    if (minItem != lastItem) {
        this.arraySet(0, lastItem);
        this.percolateDown(0);
    }
    return minItem;
}
```

Figure 20 : BinaryHeap, deleteMin()

- ***remove(E x)*** qui supprime un élément donné du tas. Tout d'abord nous cherchons l'élément dans le tas en parcourant le tableau, si nous ne le trouvons pas, l'exception ***ElementNotFoundException*** est levée. Autrement, nous utilisons ***deleteMin()*** si c'est le sommet racine, ou nous déplaçons le dernier élément à la place de l'élément retiré puis nous le comparons au père de l'élément à enlever, afin de déterminer s'il faut actualiser le tas en montant ou en descendant pour replacer le dernier élément : si ce dernier est plus petit que le père, nous utilisons ***percolateUp()*** ; sinon nous utilisons ***percolateDown()***

```
public void remove(E x) throws ElementNotFoundException {
    boolean trouve=false;
    int index=0;
    while (index<this.currentSize & !(trouve)) {
        if (this.array.get(index)==x) trouve=true;
        else index++;
    }
    if (trouve) {
        if (index == 0) {
            this.deleteMin();
        } else {
            E lastItem = this.array.get(--this.currentSize);
            this.arraySet(index, lastItem);
            if (lastItem.compareTo(this.array.get(this.indexParent(index))) <= 0)
                this.percolateUp(index);
            else
                this.percolateDown(index);
        }
    } else throw new ElementNotFoundException(x);
}
```

Figure 21 : BinaryHeap, remove()

A noter que nous avons choisi de ne pas utiliser la méthode ***remove()*** de la classe ***ArrayList<E>***, car si nous l'utilisions il faudrait redimensionner le tableau à chaque changement, tandis qu'ici nous avons besoin d'actualiser seulement la valeur ***currentSize*** (taille courante du tableau).

Synthèse des tests JUnit

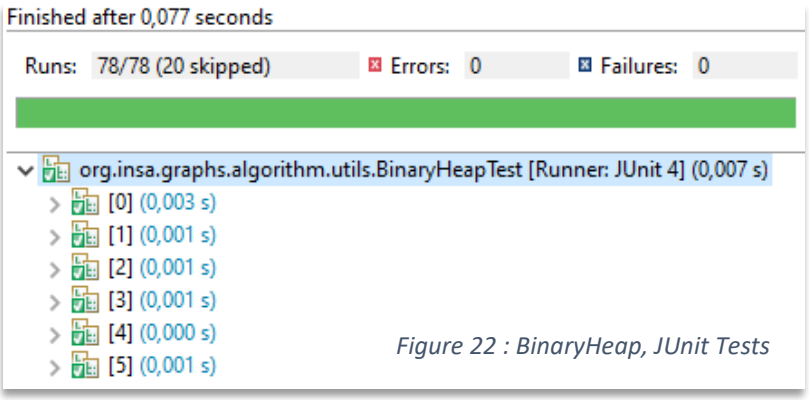

Figure 22 : BinaryHeap, JUnit Tests

De manière similaire à la classe ***Path***, nous avons validé notre classe ***BinaryHeap*** en la faisant passer des tests JUnit. Ce jeu de tests, vérifie le fonctionnement de chaque méthode dans des tas différents.

Classe Label

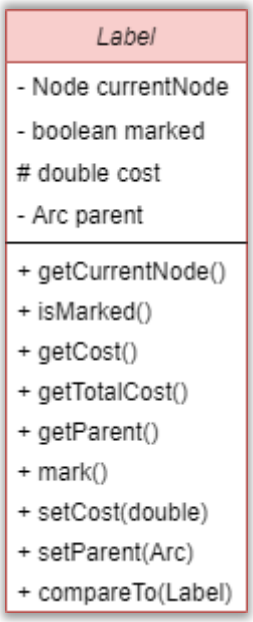

Figure 23 : Label, diagramme UML

Comme évoqué précédemment, nous utilisons un tas binaire de ***Labels***. Cette classe représente les étiquettes des nœuds : elle contient le nœud étiqueté (***currentNode***), le marquage du nœud (***marked***), le coût pour atteindre ce nœud (***cost***), et l'arc ***parent*** qui mémorise l'arc parcouru (donc le nœud parent également) pour rejoindre ce nœud.

Comme vous le montre le diagramme UML ci-contre, nous avons simplement implémenté les setteurs et getteurs nécessaires. A noter que dans cette classe les méthodes ***getCost()*** et ***getTotalCost()*** sont équivalentes, l'implémentation de ***getTotalCost()*** sera utile pour l'implémentation de la classe ***LabelStar***, que nous traiterons plus tard.

Afin de pouvoir utiliser cette classe dans le tas binaire, ***Label*** implémente l'interface ***Comparable<Label>***, ce faisant, nous avons également dû implémenter la méthode ***compareTo()*** qui permet de comparer les coûts de deux nœuds.

Validation de l'algorithme de Dijkstra

Pour implémenter l'algorithme de Dijkstra nous avons simplement suivi le pseudo-code présenté précédemment, et nous avons utilisé les méthodes correspondantes des classes que nous avons implémentées (***BinaryHeap*** et ***Label***). Nous nous sommes ensuite assurés de son bon fonctionnement.

Vérification visuelles

Afin de pouvoir vérifier visuellement le fonctionnement de notre algorithme, nous avons utilisé les méthodes de notifications de la classe ***ShortestPathAlgorithm*** qui permettent de notifier les *Design Pattern* ***Observers*** d'évènements comme l'atteinte, le marquage, ou l'exploration d'un nœud, ou également le premier évènement à savoir l'ajout du nœud d'origine dans le tas, ou encore l'évènement « nœud de destination atteint ». Par la suite, dans les graphes, les nœuds marqués seront de couleur **bleu**, tandis que les nœuds à explorer seront **turquoise** et les autres seront **orange**.

Carte carré simple

Notre premier test visuel a été fait sur la **carte carré simple**, ce qui nous a permis de nous rendre compte de la présence (ou non) de problèmes dans l'algorithme et non réellement de s'assurer de son bon fonctionnement. On voit ici que le chemin le plus court trouvé entre les deux angles est bien la ligne droite les reliant.

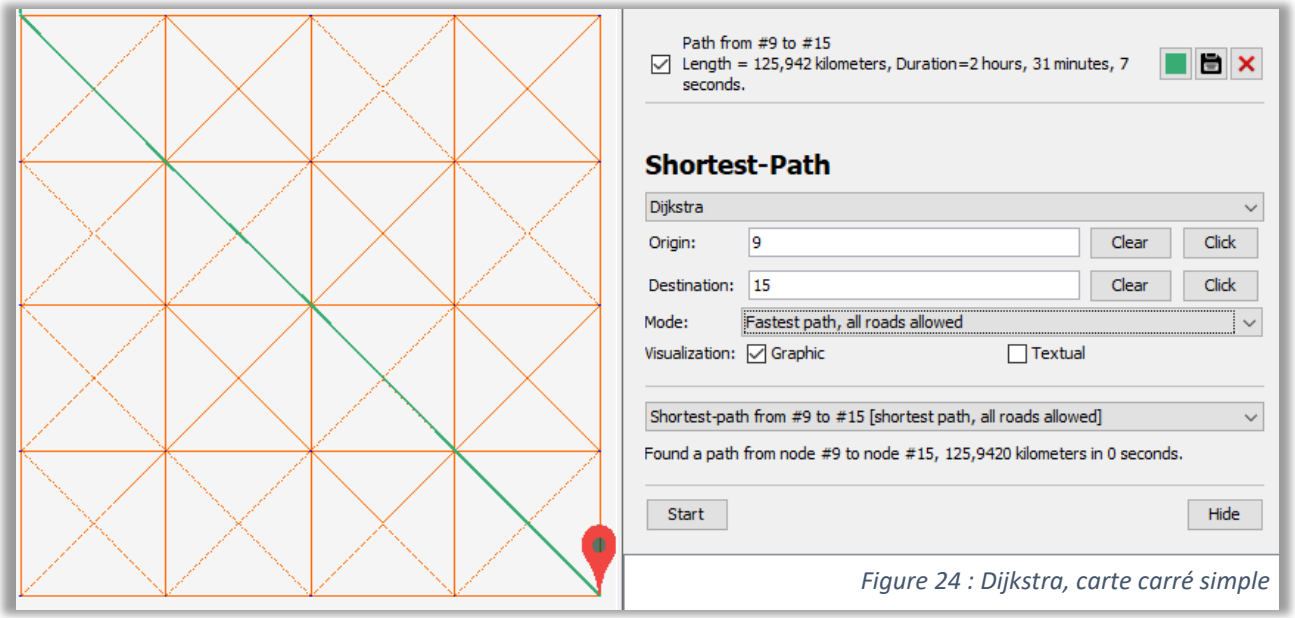

Figure 24 : Dijkstra, carte carré simple

Carte carré dense

Nous avons ensuite testé l'algorithme sur la carte carré dense : ici on peut vérifier le fonctionnement de l'algorithme de Dijkstra. La recherche se fait en largeur en prenant en compte le coût des noeuds, c'est ce qui donne cette forme autour du point de départ. On remarque également que lorsque le point de destination est atteint, l'algorithme s'arrête.

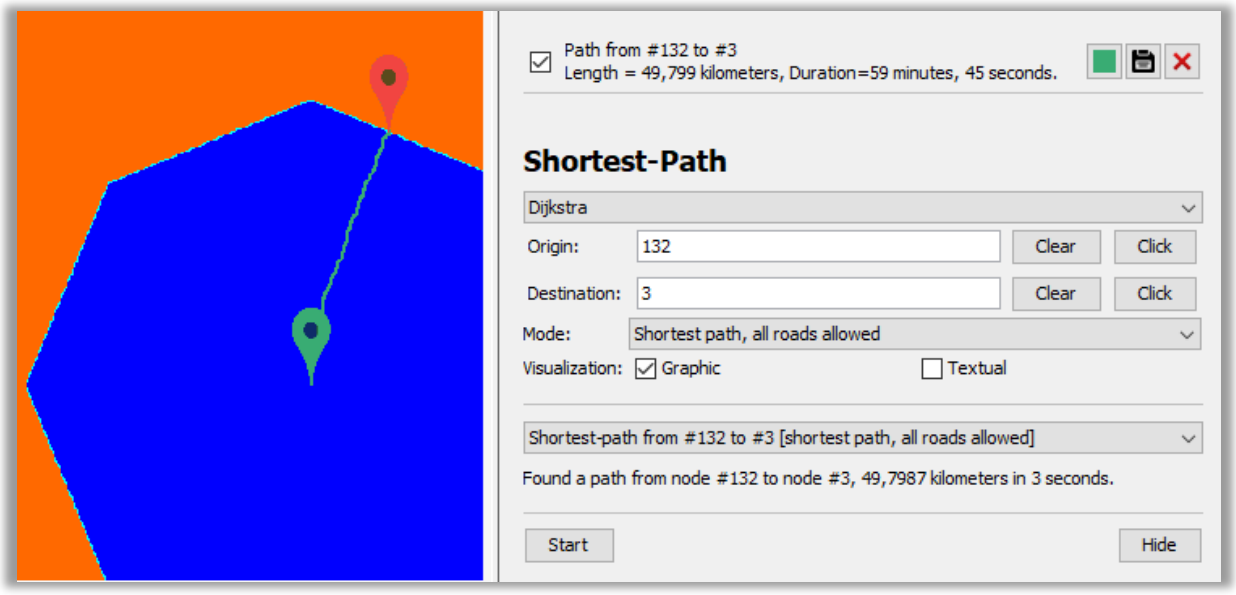

Figure 25 : Dijkstra, carte carré dense

Carte routière 1 : Haute-Garonne

Après avoir vérifié le fonctionnement de l'algorithme, nous avons voulu vérifier la prise en compte des contraintes routières : ici nous pouvons voir que le chemin trouvé par l'algorithme, lorsque nous autorisons toutes les routes (tracé rouge), longe le canal (voie piétonne) tandis que lorsque nous n'autorisons que les routes ouvertes aux voitures, le chemin est un peu plus long (tracé violet). A noter qu'il est plus rapide en temps, ce qui est expliqué par l'utilisation de la voiture.

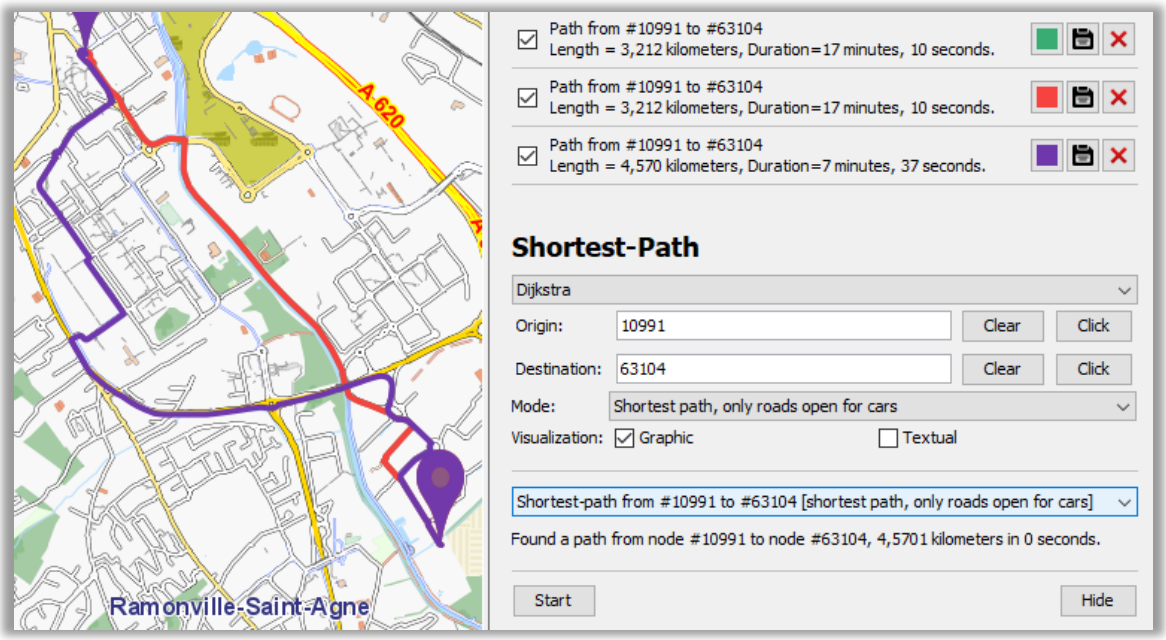

Figure 26 : Dijkstra, Haute-Garonne

Le tracé vert est le chemin le plus court fourni dans les ***Paths*** disponibles, il concorde avec le chemin trouvé par notre algorithme ce qui valide son fonctionnement.

Chemin entre deux nœuds non-connexes

Comme on peut le voir dans l'image suivante, nous avons testé l'algorithme de recherche entre deux points non-connexes de la carte Haïti/République Dominicaine: le résultat affiché « No path found » correspond bien aux attentes.

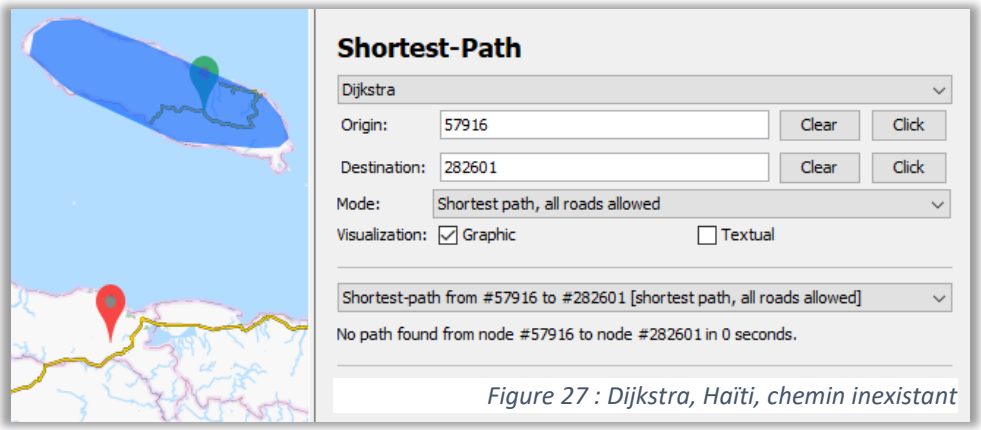

Figure 27 : Dijkstra, Haïti, chemin inexistant

Affichage de texte d'information

Au-delà des vérifications visuelles sur les cartes, nous avons ajouté des affichages d'informations (temporaires : les informations sont affichées seulement lorsque la variable booléenne *INFOS* est vraie) comme par exemple : le coût des labels marqués qui est croissant au cours des itérations ; le rapport du nombre de successeurs explorés / le nombre de successeurs total pour chaque nœud exploré ; et également la validité du tas à chaque itération.

En ce qui concerne la validité du tas, nous avons dû implémenter une méthode supplémentaire dans la classe ***BinaryHeap*** : la fonction ***isValid()*** retourne la validité du tas (un tas vide est valide). Celle-ci appelle une fonction récursive qui parcourt le tas et renvoie la « validité locale » en comparant le nœud visité et ses deux fils avant de visiter les deux fils jusqu'à atteindre les feuilles de l'arbre.

```
public boolean isValid() {
    return isEmpty() || recursiveIsElementValid(0);
    // Si le tas n'est pas vide, appel de la fonction récursive à partir de la racine
    // Arrêt lorsque l'on atteint les feuilles
}

private boolean recursiveIsElementValid(int index) {
    boolean result = true;
    // Si l'element a un fils de droite, vérifie qu'il est plus petit que lui
    // et puis vérifie que le fils en question est valide
    if (indexLeft(index)+1<this.currentSize)
        result &= ((this.array.get(index).compareTo(this.array.get(indexLeft(index)+1))<=0)
                && (this.recursiveIsElementValid(indexLeft(index)+1)));
    // De même avec le fils de gauche
    if (indexLeft(index)<this.currentSize)
        result &= ((this.array.get(index).compareTo(this.array.get(indexLeft(index)))<=0)
                && (this.recursiveIsElementValid(indexLeft(index))));
    return result;
}
```

Figure 28 : BinaryHeap, isValid()

Tests unitaires JUnit

Après avoir effectué des vérifications visuelles du fonctionnement de notre algorithme, nous avons mis en place un jeu de tests unitaires JUnit en se basant sur les tests fournis pour la classe ***Path***. Nous testons sur un graphe vide, dans un graphe crée par nos soins, contenant 6 nœuds dont 1 inaccessible, et également sur la carte chargée Haute-Garonne. Les tests vérifient la validité de la solution retournée, son statut (Optimale normalement) et pour le test sur la carte Haute-Garonne, nous vérifions pour un certain nombre de paires de points tirées aléatoirement (le nombre de paires est défini par la variable *nbIterations*), que les solutions données par l'algorithme de Dijkstra correspondent aux solutions données par celui de Bellman-Ford. A noter que le temps de test est assez important, cela est dû au fait que l'algorithme de Bellman-Ford est assez demandant en temps.

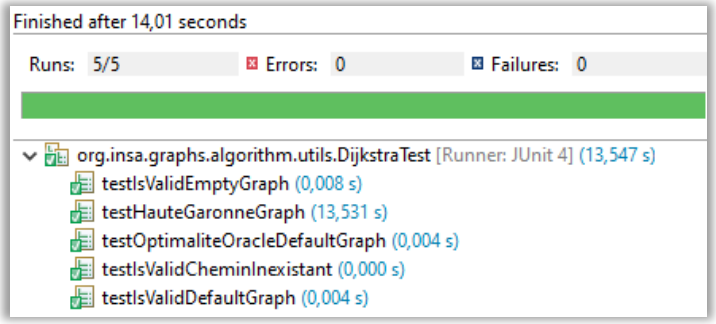

Figure 29 : Dijkstra, JUnit Tests

Chaque test contient en réalité deux « sous-tests », en effet, nous voulons tester l'algorithme en mode distance (Shortest Path) et en temps (Fastest Path), se faisant nous devons utiliser des ***ArcInspectors***. Nous les récupérons grâce à la classe ***ArcInspectorFactory***.

III – Amélioration de l'algorithme de Dijkstra

Classe LabelStar

Après avoir implémenté l'algorithme de Dijkstra et vérifier son bon fonctionnement, nous avons cherché à améliorer ce dernier en y rajoutant la notion d'heuristique (dans notre cas, le temps ou la distance estimée entre le nœud courant et la destination). Pour se faire, nous avons donc dû « améliorer » la classe ***Label*** : nous avons rajouté une classe fille ***LabelStar*** dans laquelle nous redéfinissons la méthode ***getTotalCost()*** pour qu'elle prenne en compte l'heuristique.

Le constructeur de la classe ***LabelStar*** prend en arguments supplémentaires : le nœud de destination, le mode de recherche (en distance ou temps) et également la vitesse maximale (utile seulement pour la recherche en temps). Elle appelle également la méthode ***getEstimationCost()*** détaillée ci-dessous.

```
public LabelStar(Node currentNode, boolean marked, double cost,
        Arc parent, Node destinationNode, Mode mode, int maxSpeed) {
    super(currentNode, marked, cost, parent);
    this.destinationNode = destinationNode;
    this.mode = mode;
    this.maxSpeed = maxSpeed;
    this.estimatedCost = getEstimationCost();
}
```

Figure 30 : LabelStar, constructeur

La méthode ***getEstimationCost()*** fait appel à deux sous-méthodes ***getEstimatedDistance()*** et ***getEstimatedTime()*** selon le mode de recherche de l'algorithme. Pour la recherche en distance, nous utilisons la méthode statique ***distance(Point p1, Point p2)*** de la classe ***Point***. A noter que nous vérifions que les nœuds analysés ont un ***Point*** qui les caractérise. Pour la recherche en temps, nous vérifions qu'il y a une vitesse maximale finie, auquel cas nous renvoyons le rapport de la distance estimée / la vitesse maximale autorisée.

```
private double getEstimationCost() {
    return (mode == Mode.LENGTH) ? this.getEstimatedDistance() : this.getEstimatedTime();
}

private double getEstimatedDistance() {
    if (this.getCurrentNode().getPoint() != null && this.destinationNode.getPoint() != null)
    return Point.distance(this.getCurrentNode().getPoint(), this.destinationNode.getPoint());
    // If the Node has no linked point : use Dijkstra Algorithm (Labels)
    else return 0;
}

private double getEstimatedTime() {
    return (this.maxSpeed!=-1) ? this.getEstimatedDistance()/this.maxSpeed : this.getEstimatedDistance();
}
```

Figure 31 : LabelStar, getEstimationCost()

Comme évoqué précédemment, la nouvelle méthode ***getTotalCost()*** de la classe ***LabelStar*** prend en compte le coût estimé.

```
@Override
public double getTotalCost() { return this.cost + this.estimatedCost; }
```

Figure 32 : LabelStar, getTotalCost()

A noter que nous avons également dû mettre à jour la méthode ***compareTo()*** de la classe ***Label***, pour qu'elle compare les coûts totaux (ou simplement les coûts en cas d'égalité). Ainsi, le tas binaire utilisé dans l'algorithme A* mettra le nœud qui a le coût total minimum à la racine.

```
public int compareTo(Label other) {
    int cmp = Double.compare(this.getTotalCost(), other.getTotalCost());
    return (cmp == 0) ? Double.compare(this.getCost(), other.getCost()) : cmp;
}
```

Figure 33 : Label, compareTo()

Algorithme A*

Après avoir créé la classe ***LabelStar***, il restait simplement à créer une classe fille à ***DijkstraAlgorithm*** dans laquelle les étiquettes utilisées soient des ***LabelStar***. Cela se fait à l'initialisation de la liste d'étiquettes dans laquelle nous utilisons une nouvelle méthode ***createInitLabel(Node n)*** qui renvoie un ***Label*** dans la classe ***DijkstraAlgorithm*** et un ***LabelStar*** dans la sous-classe ***AStarAlgorithm***.

```
Label[] listLabels = new Label[nodes.size()];
for (Node node : nodes) {
    listLabels[node.getId()] = createInitLabel(node);
}
```

Création de la liste d'étiquettes.

Figure 34 : Dijkstra, Label List

Création des ***Label*** dans la classe ***DijkstraAlgorithm***

```
protected Label createInitLabel (Node node) {
    double inf = Double.POSITIVE_INFINITY;
    return new Label(node, false, inf, null);
}
```

Figure 35 : Dijkstra, createInitLabel

Création des ***LabelStar*** dans la sous-classe ***AStarAlgorithm***

```
@Override
protected LabelStar createInitLabel (Node node) {
    double inf = Double.POSITIVE_INFINITY;
    ShortestPathData data = getInputData();
    int maximumSpeed =
            (data.getGraph().getGraphInformation() == null) ?
                    -1 : data.getGraph().getGraphInformation().getMaximumSpeed();
    return new LabelStar(node, false, inf, null, data.getDestination(), data.getMode(), maximumSpeed);
}
```

Figure 36 : AStar, createInitLabel()

Nouveau diagramme UML

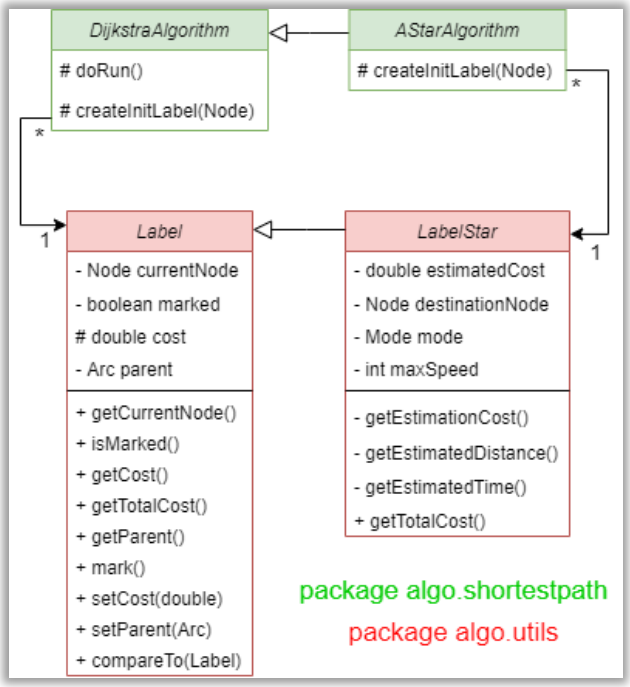

Figure 37 : Diagramme UML actualisé: DijkstraAlgorithm et AStarAlgorithm

Validation de l'algorithme A*

Vérifications visuelles

De manière analogue à la méthode utilisée précédemment pour vérifier visuellement le fonctionnement de l'algorithme de Dijkstra, nous testons l'algorithme A* sur diverses cartes. Ci-dessous, nous avons un test sur la carte de France, entre Toulouse et Vielmur-sur-Agout. Comme on peut le voir, les nœuds marqués ont une forme orientée vers la destination dans la recherche en distance étant donné que nous prenons maintenant en compte la distance estimée pour atteindre la destination. Se faisant, nous gagnons énormément de temps de calcul étant donné la diminution du nombre de nœuds visités.

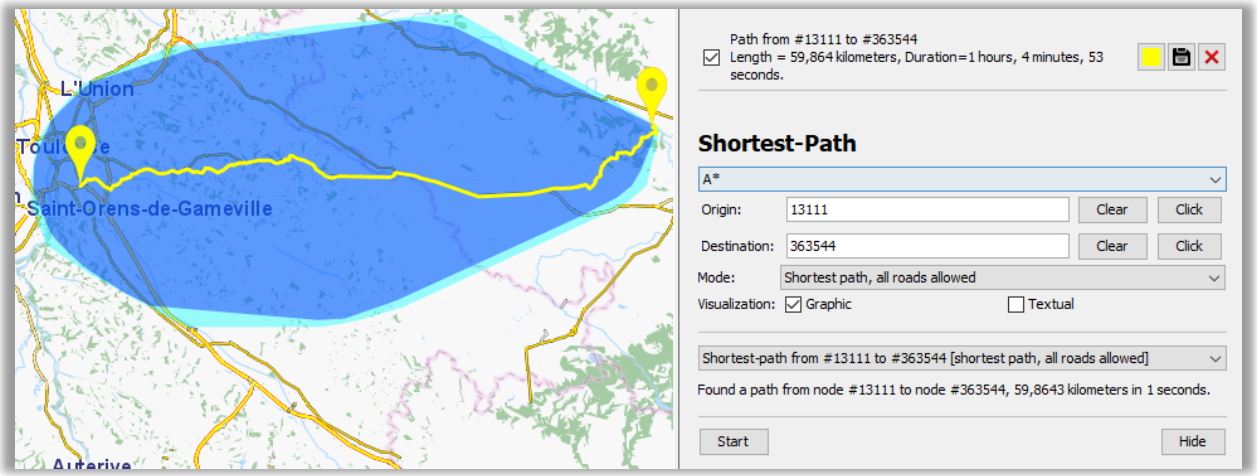

Figure 38 : AStar, test Midi-Pyrénées ShortestPath

Comme on peut le voir ici, la recherche en temps n'est pas très performante étant donné que notre heuristique n'est pas très efficace : nous prenons une vitesse maximale pour toute la carte, ce qui ne donne pas une estimation assez proche de la réalité.

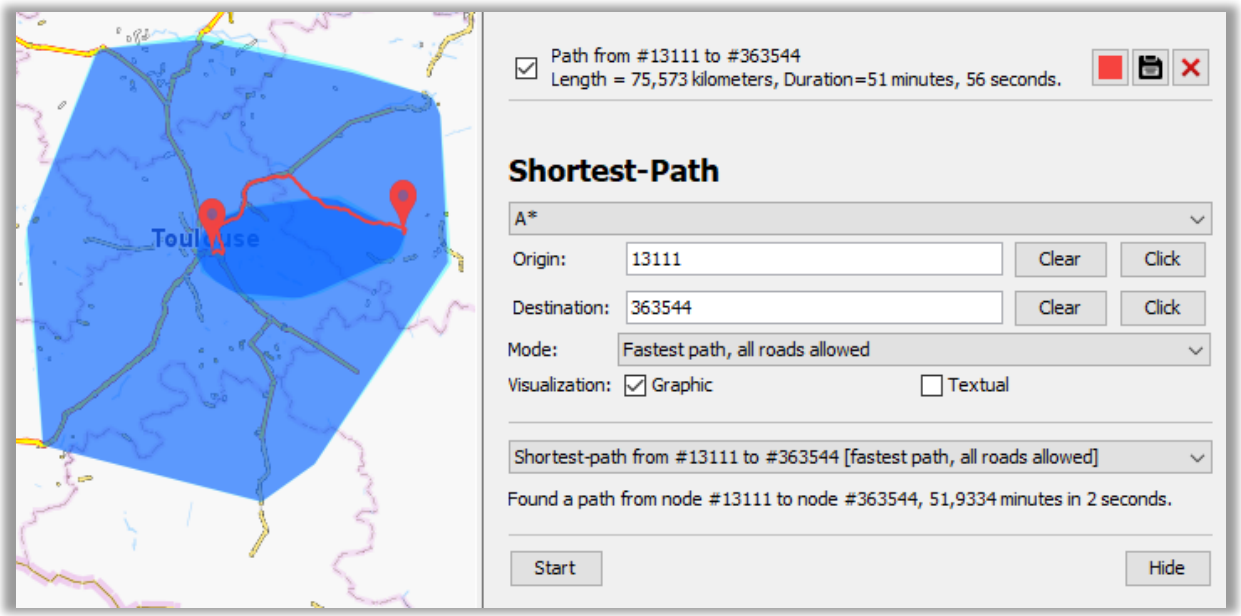

Figure 39 : AStar, test Midi-Pyrénées FastestPath

Tests unitaires JUnit

En ce qui concerne les tests unitaires après les vérifications visuelles, nous avons réutilisé les jeux de tests crées pour l'algorithme de Dijkstra. Pour se faire, nous avons mis les jeux de tests dans une classe abstraite ***AlgorithmTest***, avec deux méthodes abstraites : ***createSolution1()*** (algorithme à tester) et ***createSolution2()*** (algorithme de référence) qui permettent de choisir les algorithmes à utiliser pour créer les deux solutions à comparer. Nous utilisons ensuite deux classes filles : ***DijkstraTest*** et ***AStarTest*** qui implémentent ces deux méthodes.

```
protected ShortestPathSolution createSolution1() {
    return (new AStarAlgorithm(data)).run();
}

protected ShortestPathSolution createSolution2() {
    return (new DijkstraAlgorithm(data)).run();
}
```

Figure 41 : AStarTest, createSolution()

Ci-contre, la classe ***AStarTest*** utilise l'algorithme A* pour la solution 1 et l'algorithme de Dijkstra pour la solution 2 (pour éviter d'avoir des durées de test trop importantes avec l'algorithme de Bellman-Ford).

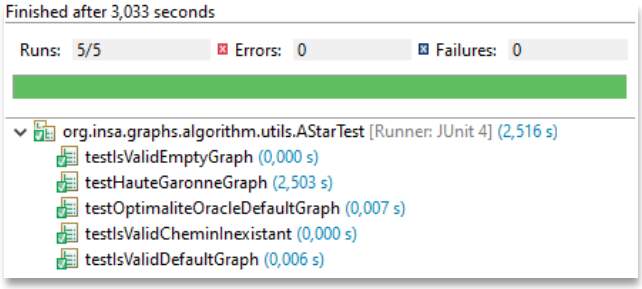

Figure 40 : AStar, tests unitaires JUnit

IV – Tests de performance des 3 algorithmes

Une fois les 3 algorithmes implémentés, il est important de les tester et comparer leurs performances. Comme nous avons pu le voir précédemment, l'algorithme de Bellman-Ford est très lent, étant donné qu'il effectue beaucoup d'itérations pour prendre en compte les arcs de poids négatifs.

Ensuite, dans le cas général, l'algorithme A* explore beaucoup moins de nœuds en s'orientant directement vers la destination, tandis que l'algorithme de Dijkstra parcourt en largeur et agrandit son « diamètre » petit à petit : A* est plus rapide. Cependant, comme on le peut le voir dans le cas suivant, s'il n'y a pas de solution possible : les deux algorithmes parcourent le même nombre de nœuds, ainsi l'algorithme A* nécessite plus de temps à cause des calculs d'estimation.

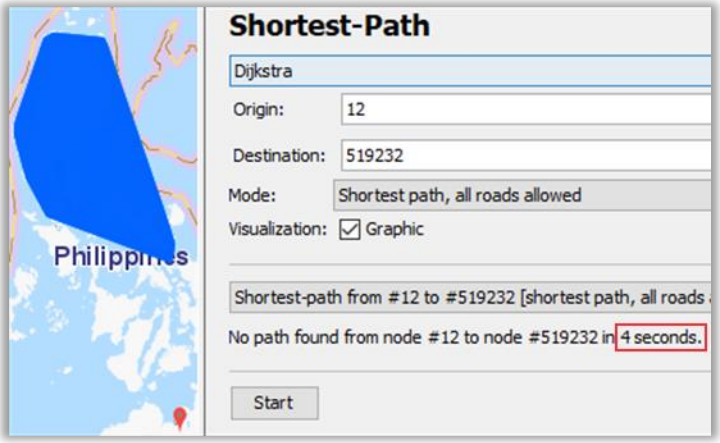

Figure 43 : Tests de performance, Dijkstra sur un chemin inexistant

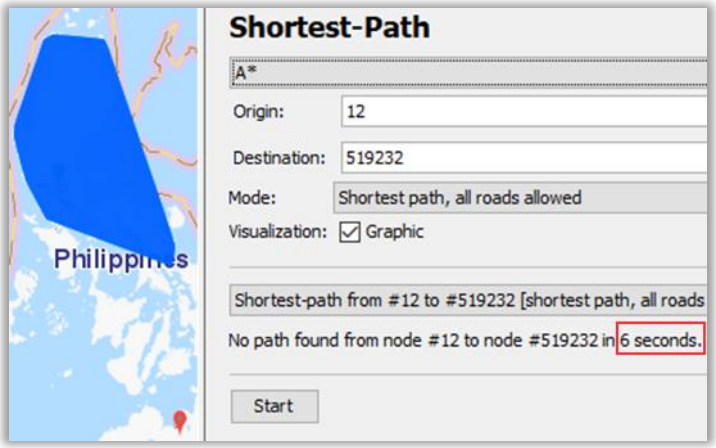

Figure 42 : Tests de performance, A sur un chemin inexistant*

Pour évaluer les performances de nos algorithmes, nous avons pensé à la création de fichiers « résultats » (qui contiendrait le temps de calcul, la valeur de la solution, la taille maximale du tas, le nombre de sommets explorés et le nombre de sommets marqués) associé à un fichier « jeu de tests ».

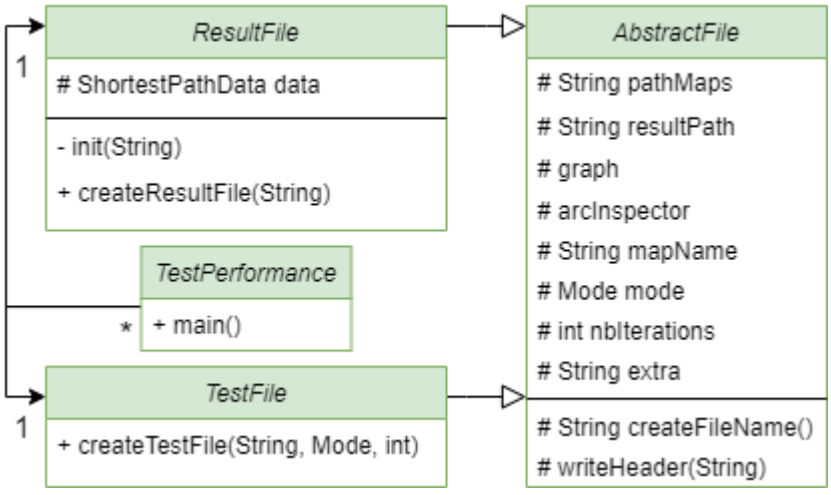

Figure 44 : Tests de performance, Diagramme UML création des fichiers

La classe abstraite ***AbstractFile*** permet d'implémenter les méthodes ***createFileName()*** et ***writeHeader()***, qui respectivement créer le nom du fichier et écrit l'entête du fichier à partir des informations disponibles (nom de la carte, mode de recherche, nombre d'itérations et un extra qui correspondra au nom de l'algorithme dans les résultats). Nous utilisons ensuite les sous-classes ***TestFile*** et ***ResultFile*** dans la classe ***TestPerformance***.

V – Problème ouvert n°6 : Trajets alternatifs

Enoncé du problème

Nous considérons un graphe composé de ***nb_noeuds*** nœuds, nous cherchons à trouver ***k*** plus courts chemins, entre le nœud d'origine ***origine*** et le nœud de destination ***destination***, qui doivent être le plus différent possible (avec une tolérance pouvant être acceptée pour partir de l'origine ou arriver à destination). Nous utiliserons un tableau de chemins de taille ***k*** : ***chemins[k]***. Nous considérons que chaque chemin est caractérisé par une ***taille***, un booléen ***trouve*** (FAUX pour un chemin inexistant), et ses ***nœuds*** qui le composent. Nous utiliserons également une liste de nœuds : ***nœuds_disponibles*** qui est passé en argument à l'algorithme de recherche A* : l'algorithme doit vérifier que les nœuds explorés appartiennent à cette liste.

Solution – Pseudo-code

```
DEBUT_ALGORITHME
    origine PREND_LA_VALEUR graphe.origine
    destination PREND_LA_VALEUR graphe.destination
    nb_noeuds PREND_LA_VALEUR graphe.taille
    noeuds_disponibles PREND_LA_VALEUR graphe.noeuds
    POUR i ALLANT_DE 0 A k
        DEBUT_POUR
        nb_noeuds_toleres PREND_LA_VALEUR 0
        chemin_trouve PREND_LA_VALEUR FAUX
        SI (i != 0) ALORS
            DEBUT_SI
            noeuds_disponibles PREND_LA_VALEUR noeuds_disponibles - chemins[i-1].noeuds
            noeuds_disponibles PREND_LA_VALEUR noeuds_disponibles + origine + destination
            FIN_SI
        TANT_QUE (NON chemin_trouve) FAIRE
            DEBUT_TANT_QUE
            chemins[i] PREND_LA_VALEUR AStar(graphe, noeuds_disponibles)
            chemin_trouve PREND_LA_VALEUR chemins[i].trouve
            SI (NON chemin_trouve) ALORS
                DEBUT_SI
                SI (i == 0) ALORS
                    DEBUT_SI
                    APPELER_FONCTION Exception : pas de chemin
                    FIN_SI
                SI (++nb_noeuds_toleres < chemin[i-1].taille - 1) ALORS
                    DEBUT_SI
                    noeuds_disponibles PREND_LA_VALEUR noeuds_disponibles + chemins[i-1].noeuds[nb_noeuds_toleres]
                    FIN_SI
                SINON
                    DEBUT_SINON
                    RENVOYER chemins
                    FIN_SINON
                FIN_SI
            FIN_TANT_QUE
        FIN_POUR
    RENVOYER chemins
FIN_ALGORITHME
```

Figure 45 : Problème ouvert, pseudo-code solution

Solution - Explications

Nous utilisons ***k*** fois l'algorithme de recherche du plus court chemin A* :

- Initialisation : on récupère les nœuds ***origine*** et ***destination*** du graphe et également le nombre de nœuds. On copie la liste de nœuds du graphe dans la liste ***nœuds_disponibles***.

- 1ère utilisation : nous effectuons simplement une recherche du plus court chemin entre ***origine*** et ***destination***. On met le chemin trouvé dans ***chemins[0]***. Si le chemin n'est pas trouvé, nous levons une Exception « Pas de chemin ».

- Autres itérations : à chaque itération suivante nous récupérons les ***noeuds_disponibles***, auxquels nous enlevons les nœuds présents dans le chemin trouvé à l'itération précédente (à l'exception de l'origine et de la destination) pour ensuite réappliquer l'algorithme A*. Tant qu'aucun chemin n'est trouvé, nous rajoutons un par un les nœuds composant le chemin trouvé à l'itération précédente.

Tolérance

En ce qui concerne la tolérance, si lors de la recherche d'un chemin ***chemins[i]***, nous atteignons le nombre maximum de nœuds tolérés à savoir la taille du chemin précédent ***chemins[i-1]***, nous nous arrêtons et renvoyons les ***i*** chemins trouvés.

Optimalité

Notre programme donne ***k*** solutions « optimales » en considérant seulement les ***nœuds_disponibles*** à chaque itération, étant donné que nous utilisons simplement l'algorithme A* qui fournit des solutions optimales.

Complexité

Dans le pire des cas la complexité peut être approximée à ***k*** * celle de l'algorithme A* : $\boldsymbol{k} * \boldsymbol{O}(\boldsymbol{b}^{\boldsymbol{d}})$, avec ***b*** le facteur de branchement et ***d*** la profondeur de la solution.

Conclusion

Durant ce bureau d'études, nous avons été initiés à la programmation orientée objet et aux concepts des graphes. A l'aide de l'interface fourni, nous avons pu programmer dans un environnement organisé et nous concentrer principalement sur les algorithmes. Ça été l'occasion de découvrir la programmation dans un « gros projet »

Tout d'abord, la conception des diagrammes UML en amont nous a permis d'avoir une vision globale des objets manipulés, et une compréhension exhaustive.

Ensuite, l'assimilation du concept de tas binaire a été primordial dans la conception des algorithmes. Également, nous avons pu nous rendre compte des difficultés pouvant être rencontrées lors du passage du pseudo-code à la réelle implémentation, notamment lors de la programmation de l'algorithme de Dijkstra.

Enfin, la phase de tests étant indispensable, nous avons dû nous intéresser au concept de *Design Pattern Observers*, et lors de l'élaboration des jeux de tests, il a été important de chercher une liste de tests exhaustive couvrant la majorité des cas possibles.

Finalement, le bureau d'études nous a demandé autonomie et efficacité, et il a été nécessaire de faire appel à nombreuses de nos compétences : allant de la compréhension de concepts complexes, à l'implémentation de programmes développés et l'utilisation du dépôt git.

INSA Toulouse
135, avenue de Rangueil
31077 Toulouse Cedex 4 - France
www.insa-toulouse.fr

Université Fédérale Toulouse Midi-Pyrénées

Liberté • Égalité • Fraternité
RÉPUBLIQUE FRANÇAISE

MINISTÈRE
DE L'ÉDUCATION NATIONALE,
DE L'ENSEIGNEMENT SUPÉRIEUR
ET DE LA RECHERCHE