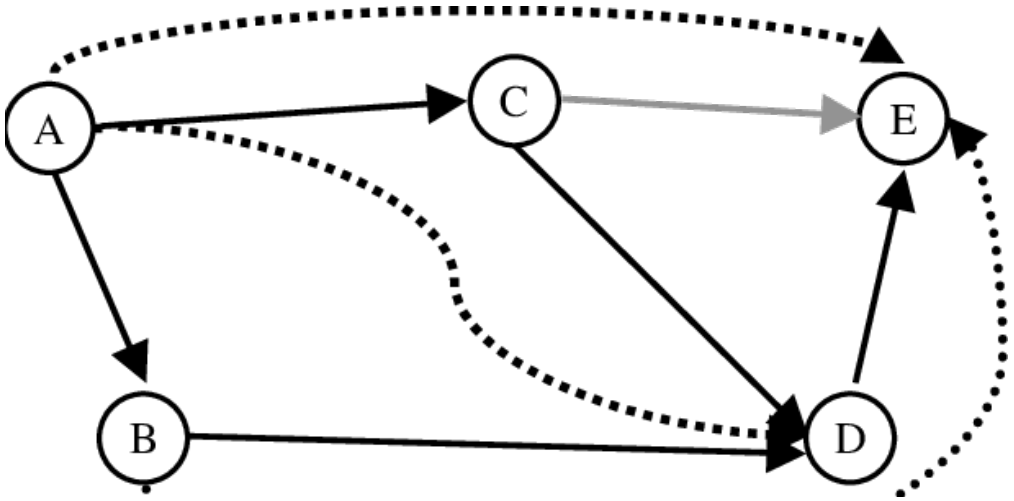
INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Rapport Be Graphes



Arrondelle Hugo – Auban Marc
3 MIC C

Insa Toulouse
2018 / 2019

Table des matières

I. Introduction

Le Bureau d'étude Graphes de 3 MIC IR a pour but la réalisation de plusieurs algorithmes de recherches de plus court chemin. Ces algorithmes seront codés dans un langage orienté objet qui sera le Java. Ce Bureau d'étude va notamment nous permettre de pouvoir mettre à profit nos connaissances acquises en théorie de Graphes.

Les algorithmes que nous avons à réaliser sont, un algorithme de Dijkstra ainsi qu'un algorithme A*. Pour réaliser l'implémentation des algorithmes suivants nous avons dû faire dans un premier temps un travail d'analyse sur le code déjà présent afin de partir sur des bases solides, puis passer à la partie programmation des fonctions à réaliser dans le Bureau d'études pour finir par une phase de test indispensable à la vérification d'un bon fonctionnement de l'algorithme. Les tests ont été réalisés grâce aux cartes déjà présente dans le BE, ce qui nous a permis de voir leur fonctionnement dans des conditions réelles.

Dans la suite de ce rapport, nous vous parlerons dans un premier temps de l'implémentation que nous avons mis en place pour les algorithmes de Dijkstra et de A*. Par la suite, nous vous présenterons les différents tests de validité et de performances que nous avons dû réaliser pour prouver l'efficacité de nos algorithmes sur différentes cartes et sous certaines conditions bien précises.

II. Conception des algorithmes

1. Conception : diagrammes de classes

Nous avons réalisé le diagramme de classe afin de comprendre le code qui nous avait été donné.

Diagramme de classes du package graph

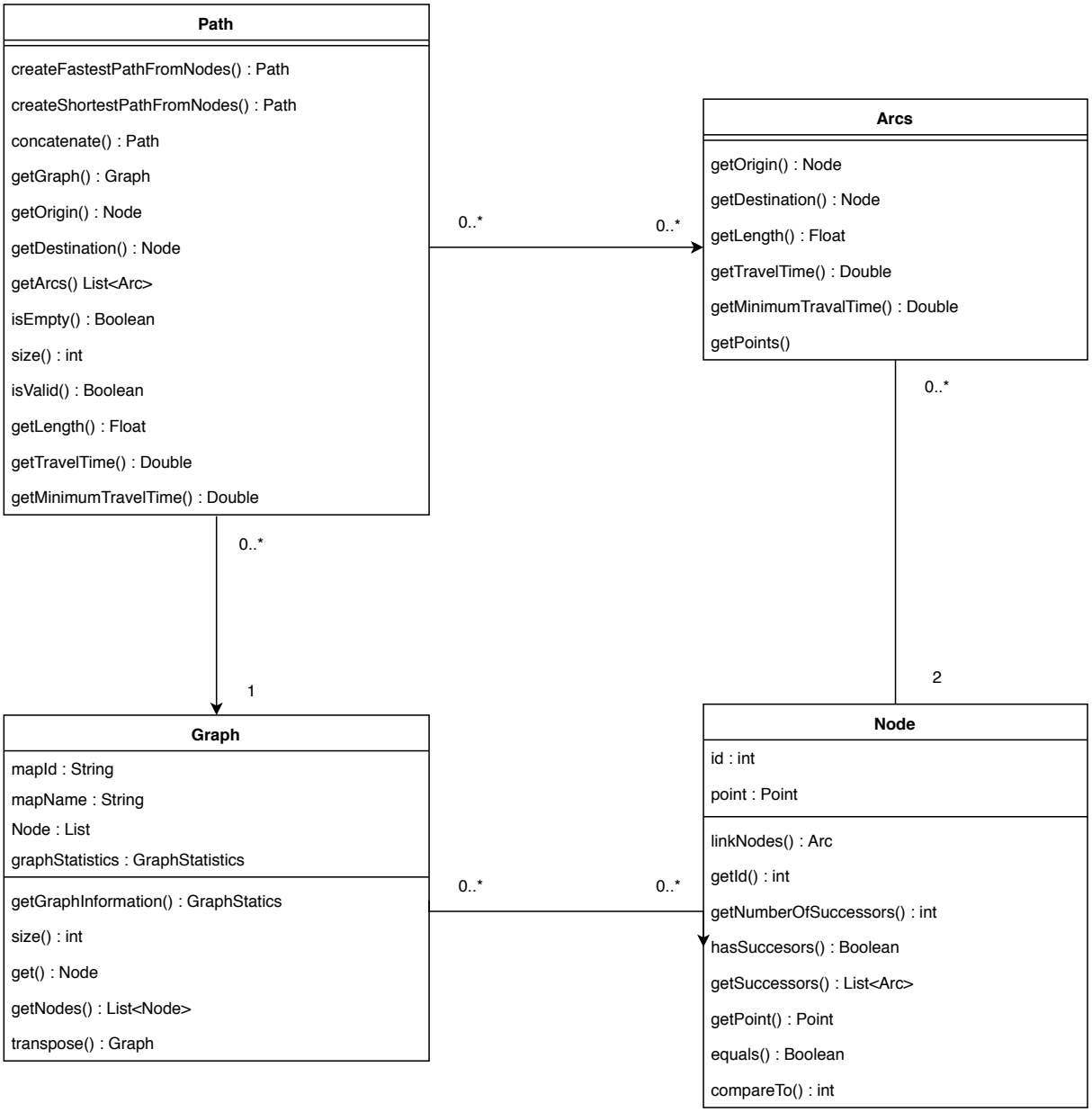
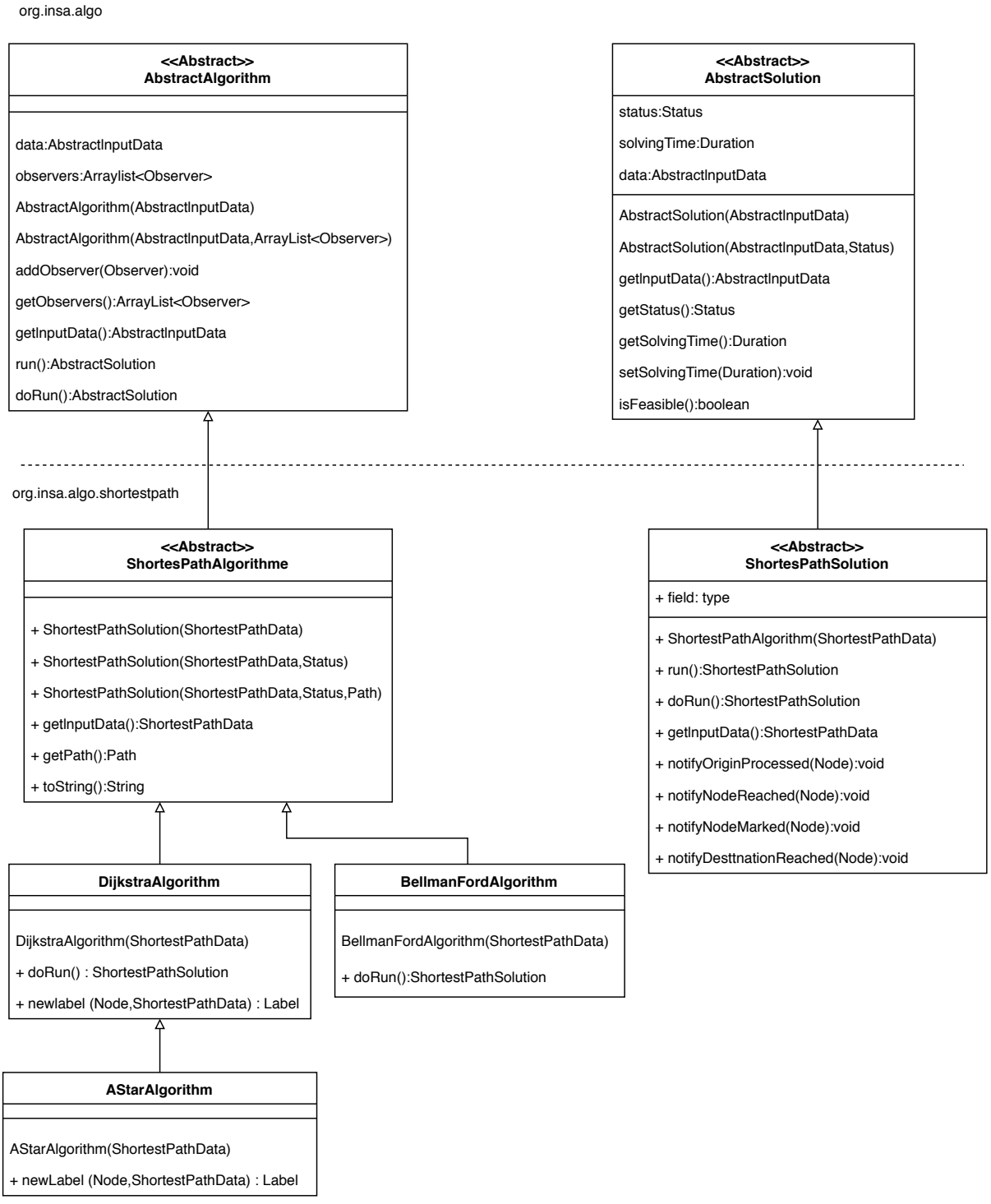


Diagramme de classes des packages algo.shortestpath et algo



2. Algorithme de Dijkstra

Pour cette partie, nous avions comme premier support un algorithme déjà présent dans le BE, celui de Bellman-Ford. Il a donc fallu en comprendre le fonctionnement exact ainsi que son implémentation pour pouvoir appliquer nos connaissances ainsi que les méthodes, afin de pouvoir commencer l'implémentation de Dijsktra et A*.

Bellman-Ford utilise **ShortestPathData** afin de récupérer les éléments nécessaires pour l'algorithme (Graphe, Nœud de départ…). L'algorithme nous fournit une solution **ShortestPathSolution** qui est un tableau d'arcs qui constitue le chemin (Path) le plus court possible. Une fois la compréhension du fonctionnement de l'algorithme de Bellman-Ford, nous pouvons passer à l'implémentation de Dijsktra.

L'algorithme de Dijkstra a pour but de trouver le plus court chemin entre deux sommets d'un graphe (orienté ou non orienté).

Le principe de l'algorithme de **Dijktra** est de parcourir toutes les solutions possibles à partir du nœud de départ et d'enregistrer le nœud et le coût de l'Arc le plus faible. Pour les itérations suivantes nous devons prendre en compte ce coût pour trouver le chemin le plus court (Le plus faible coût). Afin d'implémenter la solution de notre algorithme nous devons utiliser un **Tas** (Structure de données). Cela nous permet de récupérer le plus petit élément du tas. Ce Tas est implémenté dans la classe **BinaryHeap**.

```
BinaryHeap<Label> tas = new BinaryHeap<Label>() ;
```

Le tas est stocker grâce à une structure. Cette structure (objet de la classe Label) permet de définir le coût, le père et de marquer les éléments les plus faible.

Pour ce qui est de l'exécution de l'algorithme, la condition d'arrêt est seulement si le tas est vide. Dès la première instruction, nous supprimons le plus petit élément qui se situe dans le tas qui sera ici nommée dans notre code : **current_node**.

```
current_label = tas.deleteMin();
```

Nous sommes sûr de son coût et donc nous pouvons le marquer et enregistrer l'arc avec son prédécesseur dans notre tableau qui construira la solution au fur et à mesure de l'avancement de l'algorithme.

```
double current_length = current_label.getCout();
current_label.setMarque(true);
```

Ensuite, nous étudions tous les successeurs de ce **noeud** qui ne sont pas marqué pour continuer notre petit bout de chemin vers la solution. Au final, notre but dans cette partie du code est de vérifier si nous n'avons pas trouvé un arc qui aurait un plus faible coût que ceux qui sont déjà enregistrer dans le tableau pour la solution finale.

Pour chaque arc que nous ajoutons dans le tableau, nous ajoutons la nouvelle distance de cet arc à notre variable **newdistance**.Par la suite, nous comparons les deux variables **newdistance** et **olddistance**, si la nouvelle distance est plus faible que l'ancienne, cela veut dire que nous avons trouvé un arc avec un plus faible coût et nous mettons à jour nos informations dans le tas ainsi que dans notre tableau.

```
if(Double.compare(newdistance,olddistance) < 0)
{
    if (Double.isFinite(olddistance))
    {
        tas.remove(label_tab[successor.getId()]);
    }
    label_tab[successor.getId()].setCost(newdistance);
    tas.insert(label_tab[successor.getId()]);
    label_tab[successor.getId()].setFather(arc);
}
```

Une fois que notre tas est complètement vide, nous pouvons constituer notre solution qui sera optimale.

```
solution = new ShortestPathSolution(data, Status.OPTIMAL, new Path(graph, arcs));
```

3. Label

Pour éviter tous problèmes dans cette classe, nous utilisons deux constructeurs de classe afin d'avoir aucun problème dans la suite du BE.
Dans le premier nous avons trois arguments : le noeud courant, le coût de l'arc, et le père de ce noeud.

```
public Label(int pCourant , double cost, Arc pFather)
{
    this.sommetcourant = pCourant;
    this.marque = false;
    this.cout = cost;
    this.pere = pFather;
}
```

Dans le deuxième, si le noeud courant n'a pas de père, cela peut provoquer des erreurs. Nous ne prenons donc que deux arguments dans le constructeur qui sont le noeud courant et le coût. Pour ce qui est du père, nous lui mettons directement la valeur null.

```
public Label(int pCourant , double cost)
{
    this.sommetcourant = pCourant;
    this.marque = false;
    this.cout = cost;
    this.pere = null;
}
```

Par la suite, nous créons pleins de setter et de getter qui nous servirons dans la suite de l'implémentation pour notamment l'algorithme de Dijkstra. Nous devons aussi créer une méthode qui sera aussi très utile par la suite : CompareTo(). Cela nous permettra de comparer deux objets de type Label.

4. Algorithme de Astar

L'algorithme A* fonctionne sur le principe du rapprochement vers la destination le plus rapidement possible, c'est-à-dire que qu'il va chercher toutes les possibilités qui permettent de se rapprochent le plus de la destination et éliminer les autres. Les autres possibilités ne sont pas supprimées mais placé dans une liste d'attente si la solution en cours d'exploration s'avère incorrecte et donc ne mène pas à notre destination voulue. A* va donc se diriger vers les chemins les plus directs en premier, puis s'il ne trouve pas il retournera en arrière vers des chemins qu'il aura abandonné auparavant ce qui nous permet d'avoir la certitude qu'il trouvera toujours un chemin (S'il existe).

```java
public class AStarAlgorithm extends DijkstraAlgorithm
{

    public AStarAlgorithm(ShortestPathData data)
    {
        super(data);
    }

    @Override
    /* Réécriture de la méthode newLabel */
    /* afin d'utiliser LabelStar au lieu de Label dans l'algo */
    protected Label newLabel(Node node, ShortestPathData data)
    {
        return new LabelStar(node, data);
    }
}
```

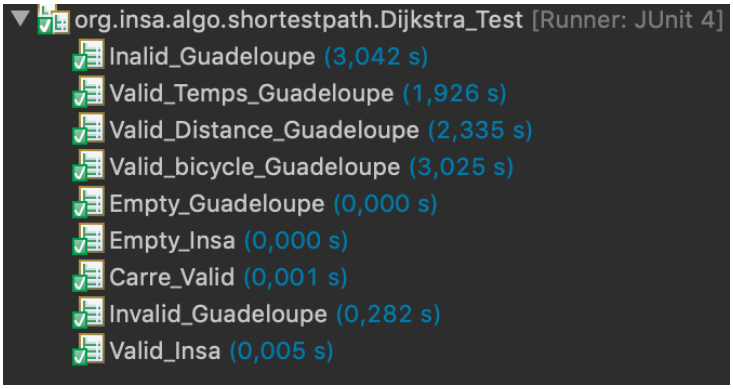
5. LabelStar

Cette nouvelle classe hérite de la classe **Label**. Avec cet algorithme nous avons besoin de coûts différents, nous réutilisons celui de la classe Label qui est cout et nous introduisons la variable inf qui représente la distance à vol d'oiseau entre le noeud actuel et la destination. Dans notre cas, il ne faut redéfinir que la méthode **getTotalCost**() qui n'est plus calculée comme auparavant comme expliqué avant.

```java
public class LabelStar extends Label implements Comparable<Label>{
    private float inf;

    public LabelStar(Node noeud, ShortestPathData data) {
        super(noeud);

        if (data.getMode() == AbstractInputData.Mode.LENGTH) {
            this.inf = (float)Point.distance(noeud.getPoint(),data.getDestination().getPoint());
        }
        else {
            int vitesse = Math.max(data.getMaximumSpeed(), data.getGraph().getGraphInformation().getMaximumSpeed());
            this.inf = (float)Point.distance(noeud.getPoint(),data.getDestination().getPoint())/(vitesse*1000.0f/3600.0f);
        }
    }

    @Override
    /* Renvoie le coût de l'origine jusqu'au noeud + coût à vol d'oiseau du noeud jusqu'à la destination */
    public double getTotalCost()
    {
        return this.inf+this.cout;
    }

}
```

III. Test de validité

Nous avons réalisé des tests de validité sur nos algorithmes en choisissant à la main les points d'origine et de destination afin de valider les différents tests possibles.
Nous nous sommes rendu compte que pour certains tests notre algorithme de Dijkstra n'était pas adapté. Nous n'avons pas pris en compte les chemins vide dans la première version de notre algorithme.



1. Chemin vide

Carte	Algorithme	Mode	Origine	Destination	Coût
Insa	Dijkstra	Distance	0	0	0
Insa	AStar	Distance	0	0	0
Guadeloupe	Dijkstra	Distance	0	0	0
Guadeloupe	AStar	Distance	0	0	0

Nous avons bien un coût de 0.

2. Chemin valide

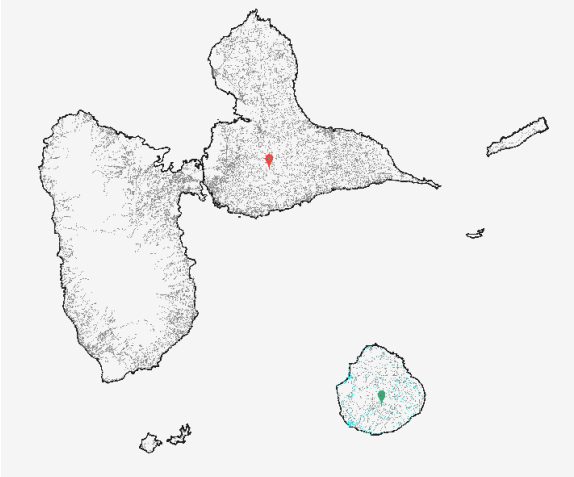
Carte	Algorithme	Mode	Origine	Destination	Coût
Insa	Dijkstra	Distance	255	525	645.86835 m
Insa	AStar	Distance	255	525	645.86835 m
Guadeloupe	Dijkstra	Distance	33022	15053	82787.56 m
Guadeloupe	AStar	Distance	33022	15053	82787.56 m
Carre	Dijkstra	Distance	23	16	85233.234 m
Carre	AStar	Distance	23	16	85233.234 m

Nous avons les mêmes résultats que ceux attendus.

3. Chemin invalide

Carte	Algorithme	Mode	Origine	Destination	Coût
Guadeloupe	Dijkstra	Distance	16060	7864	Impossible
Guadeloupe	AStar	Distance	16060	7864	Impossible

Résultat impossible



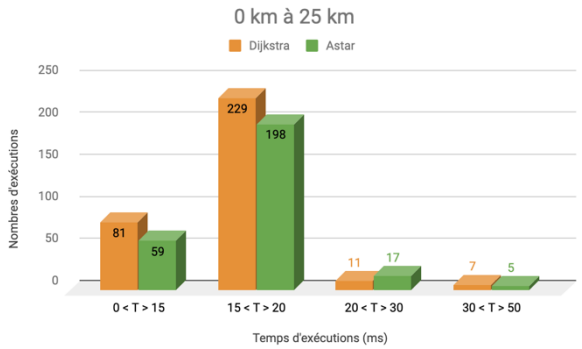
IV. Test de performance

Nous utilisons deux fonctions pour effectuer nos tests de performances. Dans la première, **InitAll**, nous créons un fichier où nous allons générer des centaines de points aléatoires qui nous servirons par la suite pour les calculs de performances.

```
Random rand = new Random();
for(int i=0;i<nb_Echantillions*2;i++)
{
    bufferedwriter.write((rand.nextInt(graphe.getNodes().size())+" "+rand.nextInt(graphe.getNodes().size())));
    bufferedwriter.newLine();
}
```

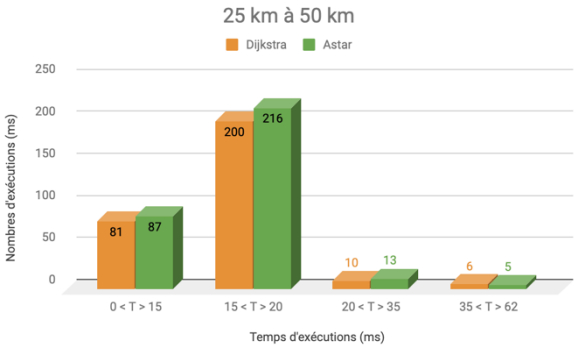
Dans la deuxième, **Extract**, nous allons lire les valeurs du premier fichier. Par la suite, pour chaque ligne, en fonction de l'algorithme choisi, nous calculons le temps d'exécution du test et nous vérifions que la solution trouvée est faisable ou non et nous enregistrons toutes ces données dans un nouveau fichier.

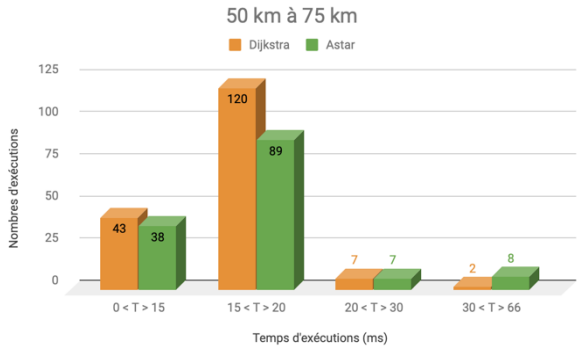
```
while(i<nb_Echantillions) {

    nodesString = br.readLine().split(" ");
    Ori = Integer.parseInt(nodesString[0]);
    Dest = Integer.parseInt(nodesString[1]);

    ShortestPathData data = new ShortestPathData(graph,
            graph.getNodes().get(Ori),
            graph.getNodes().get(Dest),
            ArcInspectorFactory.getAllFilters().get(0));

    ShortestPathAlgorithm Choix_Algo;

    switch(args) {
    case "B" :
        Choix_Algo = new BellmanFordAlgorithm(data);
        break;
    case "D" :
        Choix_Algo = new DijkstraAlgorithm(data);
        break;
    case "A" :
        Choix_Algo = new AStarAlgorithm(data);
        break;
    default :
        System.out.println("Erreur algo pas reconnu: use is A of A* B for BellmanFord or D for Djikstra");
        throw new IOException();
    }
    long startTime = System.currentTimeMillis();
    ShortestPathSolution Solution = Choix_Algo.doRun();
    long endTime = System.currentTimeMillis();
    long temps_exe = endTime - startTime;

    bwR1.write("*************************************************************");
    bwR1.newLine();
    bwR1.write("Origine : " + Ori + "  ----- Destination : " + Dest);
    bwR1.newLine();

    if(Solution.isFeasible())
    {
        //bwR1.write("Distance : " + Solution.getPath().getLength() + " Temps d'exécution : " + temps_exe + " ms");
        bwR1.write(Solution.getPath().getLength() +" "+ temps_exe);
        bwR1.newLine();
        //bwR1.newLine();
    }
    else
    {
        //bwR1.write("Infaisable");
        //bwR1.newLine();
    }
    i++;
}
```

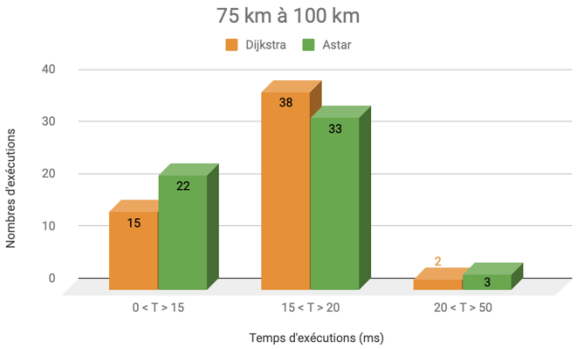


Nous avons testé la performance de nos algorithmes de plus court chemin en mesurant leur temps d'exécution. Nous avons réalisé ces tests à l'aide de 1000 couples aléatoires d'origine / destination pour la carte de Guadeloupe.

Nous nous sommes rendu compte que A* est plus rapide à s'exécutés



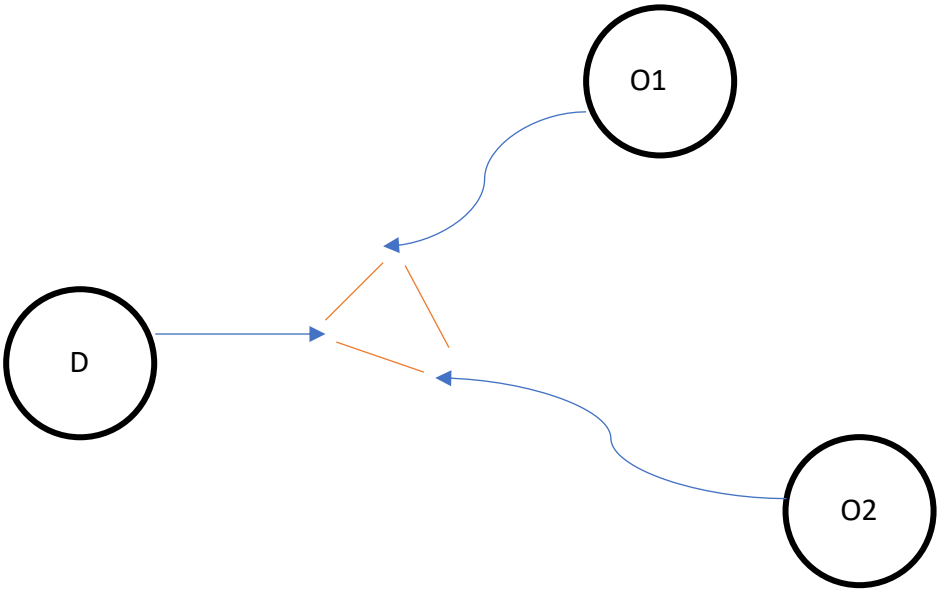




V. Problème Ouvert

Nous avons choisi de traiter le problème ouvert du Covoiturage.
Pour ce qui est du problème, nous avons eu comme idée d'utiliser l'algorithme de Dijkstra. Étant donné que le sujet porte sur la recherche de plus court chemin entre trois points et leur destination, il nous semble être l'algorithme le plus adapté à la situation. Partant de cette idée, nous nous sommes interrogés sur comment résoudre le problème. Après multiples réflexions, nous avons pensé à lancer un Dijkstra simultanément depuis chaque point.

Cela voudrait dire, qu'à chaque itération, nous regarderons les coûts des trois algorithmes et garderons celui qui offre la meilleure performance. Ceux qui n'ont pas eu la chance de pouvoir avoir le meilleur coût, continue quand même. On recommence cette étape jusqu'à que deux Dijkstra se rencontre. Quand cela arrive, nous décidons d'arrêter les deux qui se sont rencontrés et de laisser le dernier continuer son chemin jusqu'au point de rencontre des deux autres.



VI. Conclusion

Dans ce BE, nous aurons appris beaucoup de choses qui nous seront certainement utiles dans les années à venir ainsi que dans notre futur métier. L'utilisation du langage de programmation orienté objet Java qui est l'un des langages les plus utilisés dans le monde, nous a permis d'approfondir nos connaissances ainsi que nos méthodes. En parallèle de cela, l'introduction de l'outil Git permettant de travailler ensembles dans le groupe, fut une très bonne idée ainsi qu'une belle expérience et une belle découverte qui nous permettra d'être déjà formés sur les notions de base de son utilisation. De plus, l'environnement de travail Éclipse est aussi une découverte, ce qui ajoute encore de nouvelles capacités de travail ainsi que de développement à notre bagage. Étant donné que la cohésion dans le groupe été présente, l'efficacité au fur et à mesure des séances fut la même. Malgré les quelques problèmes rencontrés, nous avons sû persévéré et par conséquent réussi à les surmonter afin de fournir des résultats satisfaisants. Pour conclure, nous avons adorés travailler sur ce sujet passionnant qui est la recherche de plus court chemin sur des cartes toutes les unes différentes des autres grâce à différents algorithmes.