Corentin ENIREVEC
Siméon GAUMART 3MIC D

Rapport de BE GRAPHE

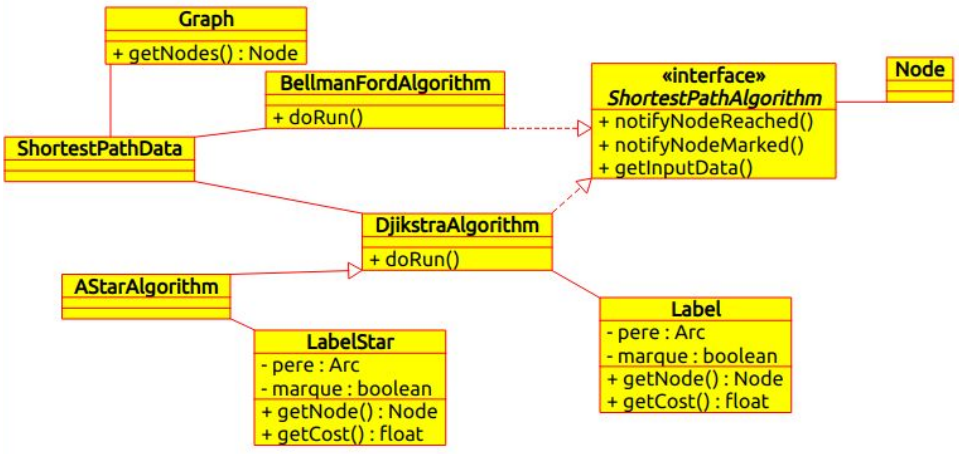
Introduction

Notre projet de BE Graphes porte sur la conception d'algorithmes de parcours du plus court chemin. Cela nous a permis de retourner sur le travail effectué et les algorithmes introduits par le module de Graphe mais aussi de consolider nos compétences en programmation orienté objet. Pour ce faire, nous ne partions pas d'une feuille blanche. Une base nous a été fournie, un code partagé par GitHub, contenant l'interface permettant le choix de la carte, de l'algorithme, et position de départ et de d'arrivée ainsi qu'une interface plus légère permettant l'affichage d'un chemin sur une carte prédéfinie. Cela nous permit de nous occuper de l'essentiel, coder des algorithmes de plus court chemin. Cette algorithme ont donc pu être tester par l'intermédiaire de cette interface en indiquant la carte, l'algorithme et les positions de départ et d'arrivé.

Dans ce rapport, nous reviendrons sur la conception de ces algorithmes ainsi que sur leurs résultats, obtenus en testant tout d'abord s'il on obtient ce que l'on souhaite mais aussi en testant leur performances .

Conception et Implémentation

La première étape avant l'implémentation des deux algorithmes, il était nécessaire de bien connaître quels éléments seront utiles et participeront à leur élaboration. Voici ci-dessous le diagramme UML partiel des objets du BE.



Après avoir implémenté quelques méthodes nécessaires dans les classes Path, BinaryHeap et après avoir testé leur bon fonctionnement avec Junit, nous avons cherché à implémenter

l'algorithme de Dijkstra (avec une version basique et une version "A star") en passant par l'implémentation d'une classe Label.

Label:

Afin d'implémenter les algorithmes de Dijkstra, nous avons donc implémenté une classe Label composée d'un noeud et d'un coût de l'origine vers ce noeud. Cette classe permet aussi de récupérer facilement des informations sur les Labels (comme récupérer son noeud, son père, son coût, savoir s'il est marqué, le comparer avec d'autres Labels) de mettre à jour les Labels (en définissant leur père, leur coût) et de les marquer (pour pas les revisiter dans l'algorithme de Dijkstra).

De plus, cette classe Label implémente la classe Comparable :

```
public class Label implements Comparable<Label> {
```

Ce qui permet d'initialiser la méthode CompareTo dans cette classe Label, et qui pourra être utilisée dans d'autres classes (on l'utilisera par exemple dans la méthode percolateDown de la classe BinaryHeap quand on utilisera un tas de Label dans l'algorithme de Dijkstra):

```
public int compareTo(Label other) {
    return Double.compare(this.getTotalCost(), other.getTotalCost());
}
```

Dijkstra:

Nous avons donc ensuite utilisé cette classe pour les algorithme de Dijkstra qui consistaient à manipuler des Labels (dans un tas binaire) et à les mettre à jour afin d'à chaque fois déterminer le Label avec le plus petit coût et de mettre à jour le coût de ses voisins jusqu'à arriver à la destination et finalement déterminer le plus court chemin. Nous avons initialisé les Labels et les tableaux de Labels dans des méthodes à part pour faciliter ensuite la conception de l'algorithme A*:

```
protected Label[] createLabelTab() {
    Label[] Lbl=new Label[graph.getNodes().size()];
    Lbl[data.getOrigin().getId()]=new Label(data.getOrigin());
    return Lbl;
}

protected Label createLabel(Node node) {
    return new Label(node);
}
```

On remarque ici qu'une des méthodes sert à créer un tableau de Label de taille le nombre de noeuds dans le graphe, et en initialisant le Label d'origine dans ce tableau. Et l'autre permet de créer un Label.

LabelStar:

Pour A*, nous avons dû ajouter au coût la distance à vol d'oiseau entre le noeud du Label et la destination. Pour cela, nous avons dû créer une classe LabelStar héritant de Label et où nous avons modifié la méthode getTotalCost() afin qu'elle soit égale à la somme du coût et de la distance vue précédemment:

```
private double cost_to_dest;

public LabelStar(Node node, Node destination) {
    super(node);
    this.cost_to_dest=Point.distance(node.getPoint(), destination.getPoint());
}

public double getTotalCost() {
    return (this.cout)+(this.cost_to_dest);
}
```

Comme nous avions comparé deux labels avec la méthode compareTo utilisant getTotalCost, la comparaison de deux labels a été mise à jour par l'héritage en changeant la méthode getTotalCost comme au dessus.

DijkstraStar:

Comme dit précédemment, la conception de l'algorithme A* se faisait en ajoutant la distance à vol d'oiseau en plus du coût lors des comparaisons. Pour cela nous avons dû utiliser la classe LabelStar (vue au-dessus) au lieu de la classe Label dans l'algorithme A*.

Pour réaliser cela, nous avons fait hériter l'algorithme A* de l'algorithme de Dijkstra en nous avons seulement dû modifier les méthodes createLabelTab et createLabel:

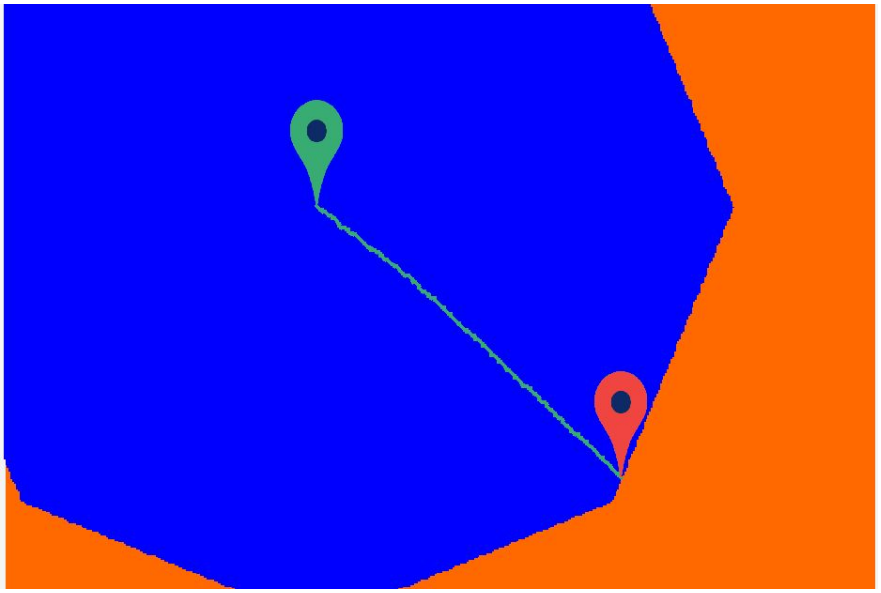
```
protected Label[] createLabelTab() {
    LabelStar[] Lbl=new LabelStar[graph.getNodes().size()];
    Lbl[data.getOrigin().getId()]=new LabelStar(data.getOrigin(),dest);
    return Lbl;
}

protected Label createLabel(Node node) {
    return new LabelStar(node,dest);
}
```

Comme le montre la capture, nous avons modifié ces deux méthodes afin de créer des LabelStar au lieu de labels. Nous pouvons cependant toujours utiliser le tas et autres

fonctions qui utilisent des labels car la classe LabelStar (comme vu au-dessus) hérite de la classe Label et est donc convertible.

Tests de validité

Premièrement, on teste l'algorithme de Dijkstra sur un carré dense, et on fait une vérification visuelle:

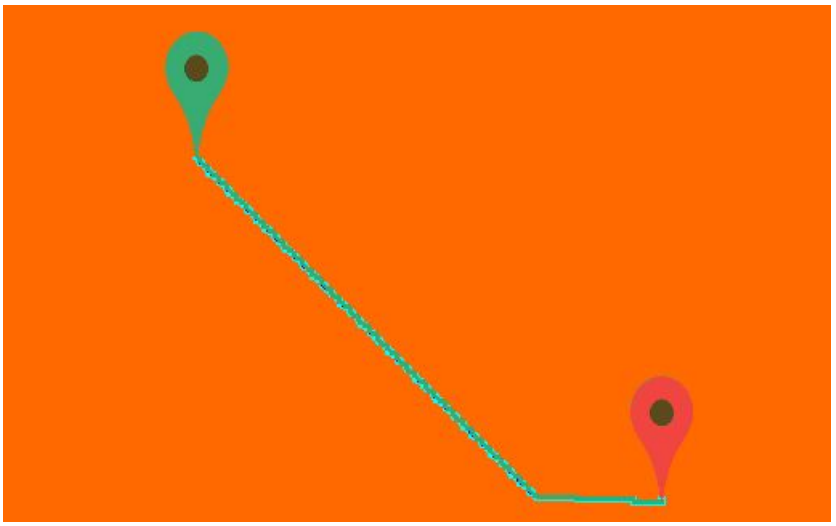


Chemin obtenu lors de la recherche par Dijkstra

On remarque que le chemin trouvé finalement a l'air de correspondre (visuellement) au chemin le plus court. De plus, on peut remarquer que seulement une partie des sommets ont été visités (en bleu), et qu'ils sont autour du sommet de départ.

Pour l'algorithme A*, nous avions fait une première version où nous avions remplacé tout les getCost par des getTotalCost dans l'algorithme de Dijkstra, et cette version fonctionnait mais était encore plus lente que l'algorithme de Dijkstra.

Nous avons donc fait une autre version ou nous avions laissé getCost et où getTotalCost était utilisé seulement avec le compareTo fait lors de l'appel de la fonction deleteMin (et donc percolateDown) dans le tas binaire. Cette version est beaucoup plus rapide mais ne donne pas exactement le plus court chemin et donc ne fonctionne pas correctement, mais nous n'en avons pas encore déterminé la cause.



Chemin obtenu lors de la recherche par A*

Réalisation de tests unitaires

La vérification des résultats passe bien sûr par la conception de tests automatisés des algorithmes de Dijkstra et A* qui est basé sur ce dernier. L'idée est de vérifier la validité des chemins créés lors de la recherche des plus courts chemins sur plusieurs cartes (routières et non routières) et avec des chemins de taille nulle, l'origine est égale à la destination, des chemins réalisables et des chemins irréalisables (la destination est inaccessible) mais aussi de réaliser de réaliser ces tests en distance et en temps.

Nous avons choisi de nous baser sur quatre cartes : carré, carré dense, Toulouse et Guadeloupe. Les deux premières cartes sont non-routières et la dernière en plus d'être routière, et composée d'îles ce qui permet de créer des chemins irréalisables afin de voir le comportement de l'algorithme dans ces cas là.

Pour la réalisation des tests unitaires avec JUnit, il a fallu s'intéresser aux caractéristiques principales de ce paquet. Tout d'abord toutes les classes de tests doivent contenir une fonction initAll() qui sera le point de lancement de ce test, il permettra d'initialiser les éléments communs aux différents tests comme les cartes par exemple. Chaque méthode précédé de la mention « @Test » correspondra à un test, à exécuter après le initAll(). Chacun des tests seront exécutés l'un après l'autre dans l'ordre dans lequel ils ont été placés. La fonction JUnit assertEquals permettra de vérifier étape par étape la validité.

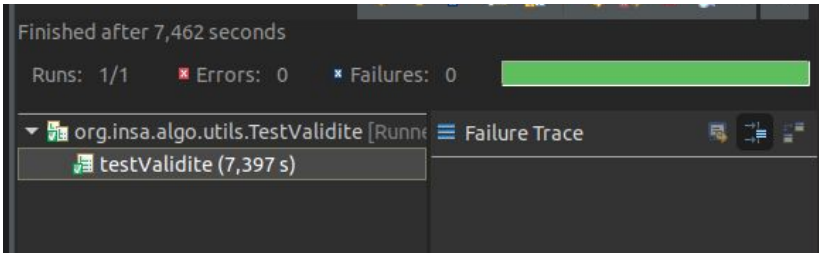
La fonction initAll() récupère toutes les cartes qui seront utilisées par la suite

```
public void initAll() throws Exception {
    System.out.println("Chargement des cartes");

    //Initialisation des graphes
    String prefix = "/home/user/Téléchargements/";

    for (String str: mapList) {
        String mapName = prefix + str;
        // Create a graph reader.
        GraphReader reader = new BinaryGraphReader(
                new DataInputStream(new BufferedInputStream(new FileInputStream(mapName))));

        // TODO: Read the graph.
        graphList.add(reader.read());
    }
}
```

Voici la fonction de test de la validité des chemins, chaque test sera effectués de cette manière , un assertEquals du paquet JUnit permettra de savoir s'il on obtient bien le résultat obtenu par l'algorithme de Bellman-Ford.

```
public boolean testPath(Graph graph,int origin,int destination,int arcInspectorNumber,String algorithm) {
    //Création d'un ShortestPathData
    ArcInspector arcInspector = ArcInspectorFactory.getAllFilters().get(arcInspectorNumber);
    ShortestPathData data = new ShortestPathData(graph, graph.getNodes().get(origin), graph.getNodes().get(destination), arcInspector);
    //Définition de l'algorithme utilisé
    ShortestPathAlgorithm spa;
    switch (algorithm) {
            case "dijkstra":
                spa = new DijkstraAlgorithm(data);
                break;
            case "bellman":
                spa = new BellmanFordAlgorithm(data);
                break;
            case "a*":
                spa = new AStarAlgorithm(data);
                break;
            default:
                System.out.println("Unknown type of algorithm");
                return false;
    }
    //Exploitation de la solution
    ShortestPathSolution sps = spa.run();
    return (sps.isFeasible())? sps.getPath().isValid() : false;
}
```

Résultat



À part un chemin entre la Désirade et Grande Terre (Carte Guadeloupe) tous les chemins obtenus sont valides, on obtient bien ce que l'on souhaite.

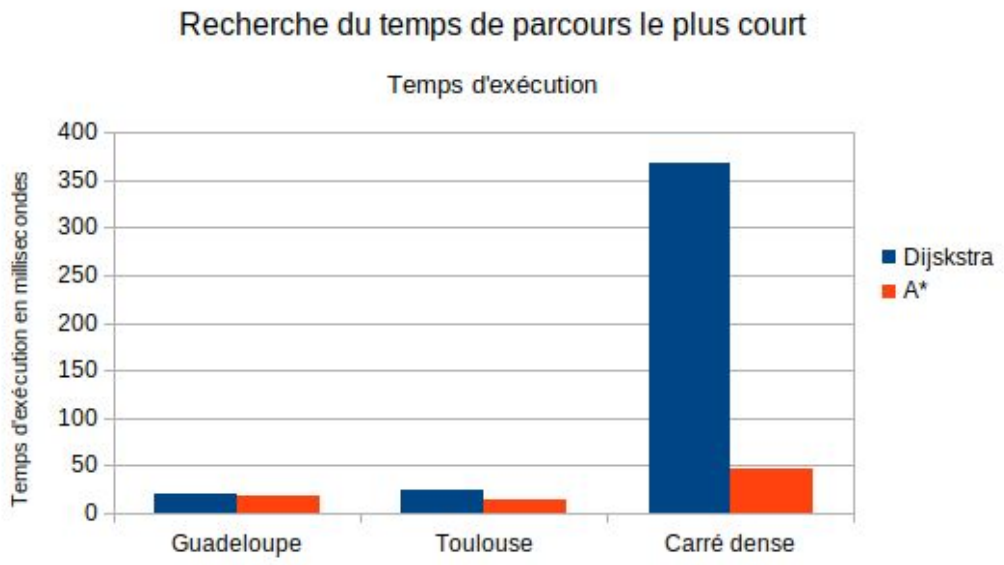
Tests de performance

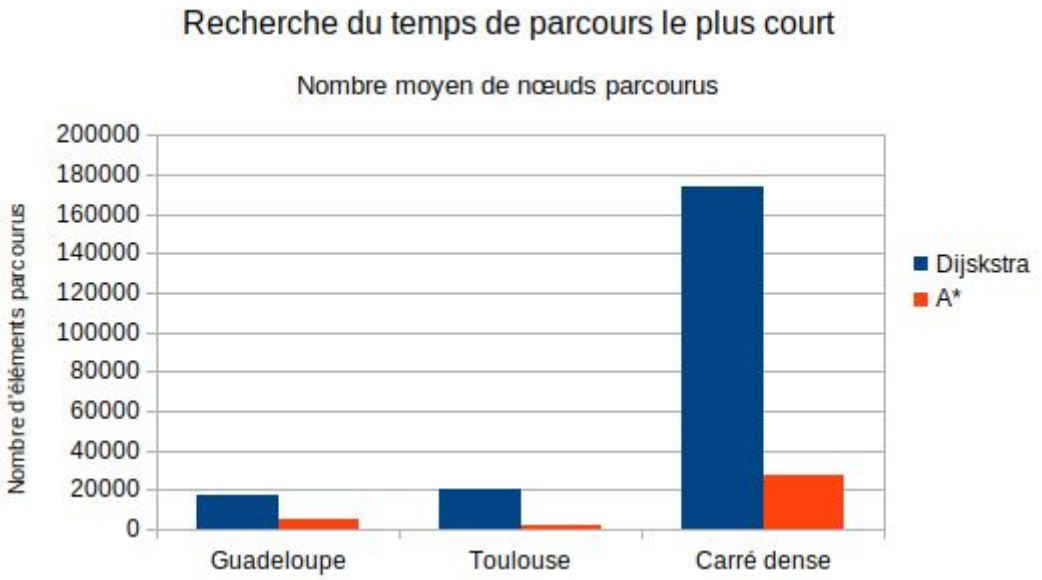
Préparation des tests

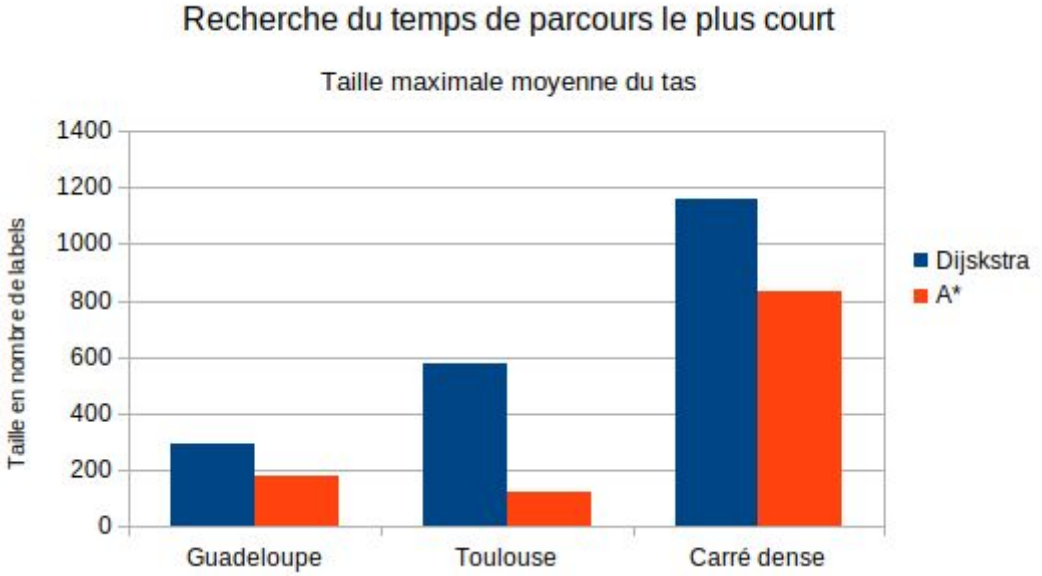
Chacun des tests ont été réalisés sur les cartes Guadeloupe, Toulouse, et carré dense. Ces trois cartes ont été choisis pour le caractéristiques propres : La carte Guadeloupe représentant une archipel possède une multitude de chemins inaccessibles, la carte Toulouse dispose de plusieurs autoroutes et la quasi totalité des chemins sont accessibles, la carte carré dense dispose d'un très grand nombre d'arcs et tous ses chemins sont accessibles.

Ces trois cartes nous permettrons d'analyser l'impact de l'inaccessibilité de chemins, la différence de temps de parcours des algorithmes, le nombre de nœuds atteints et ceux marqués et la taille maximale du tas. Afin de faciliter l'obtention et l'utilisation des données, chaque tests exécutera un nombre élevé de recherche de chemins et les résultats seront stockés dans un fichier .csv, exploitable par un tableur.

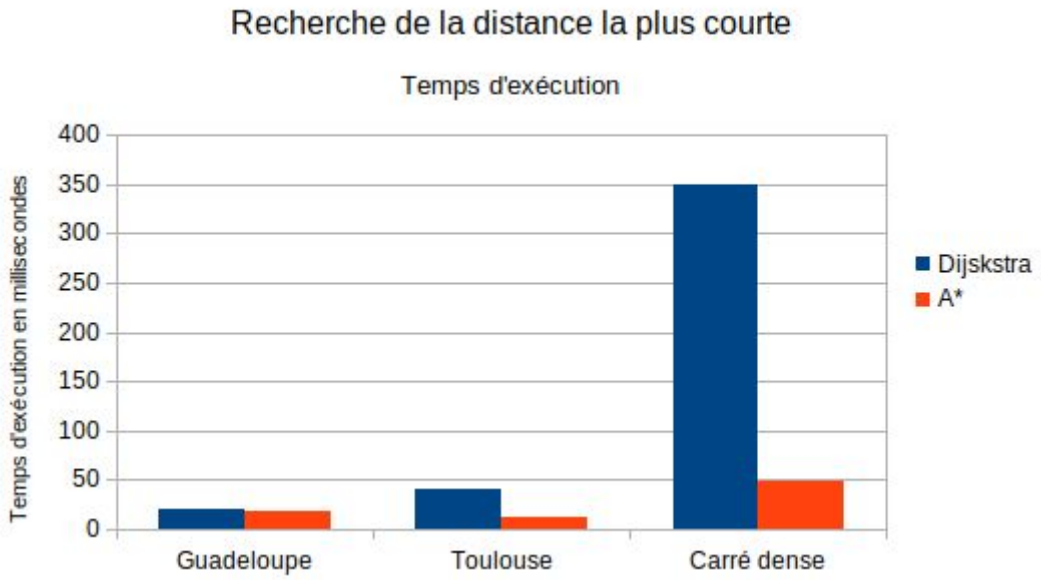
Tests de performances en temps







Tests de performances en distance



Le nombre de nœuds parcourus et la taille maximale du tas ne varie pas que l'on utilise une recherche du plus court chemin en distance ou en temps.

Analyse des résultats

On remarque en faisant ces tests que l'algorithme de A* prend nettement moins de temps et stock beaucoup moins dans le tas quelque soit la carte utilisé. De plus, on observe que plus une carte possède de nœuds, plus l'écart entre l'algorithme A* et Dijkstra sera grand. Le mode de parcours , en distance ou en temps n'ont pas d'impact significatifs sur le temps, le nombre de nœuds parcourus et la complétion du tas.

Pour améliorer l'algorithme de Dijkstra, il faudrait donc réduire la complexité spatiale de l'algorithme qui est trop élevé (ce qui se voit par la très forte augmentation du nombre maximum d'éléments mis en pile lors que le nombre de nœuds augmente). Cela permettra aussi de réduire la complexité temporelle car en ayant un nombre de nœuds à parcourir moins élevé le nombre d'arcs à parcourir sera plus restreint.

Problème ouvert :

Comme problème ouvert, nous avons choisi le problème 1 (covoiturage).

Le but de ce problème est de trouver un trajet de covoiturage pour deux automobilistes U1 et U2 partant respectivement des points O1 et O2 et voulant arriver à une destination D en faisant du covoiturage à partir d'un point à déterminer.

Pour réaliser cela, on peut commencer par trouver le plus court chemin des trajets O1-D et O2-D. On sait que le plus long de ces deux trajets sera donc le trajet minimal de covoiturage.

Imaginons que le plus long chemin des deux soit le chemin O1-D. Le chemin optimal de covoiturage sera donc O1-D.

On peut commencer par voir si les deux chemins O1-D et O2-D ont des noeuds en commun. Si c'est le cas, les automobilistes U1 et U2 n'ont qu'à se rencontrer au premier noeud commun et le reste des noeuds des deux chemins sont forcément identiques (car un sous-chemin de plus court chemin est un plus court-chemin) s'il n'y a pas plusieurs plus courts chemins égaux entre le noeud de rencontre et D. Et donc U2 devra attendre U1 au noeud de rencontre.

Si les deux chemins n'ont aucun noeud en commun, l'automobiliste U1 fait son plus court chemin (car c'est lui qui a le plus long des deux), et pour chaque noeud X par lequel il passe on calcule le plus court chemin entre O2 et X pour l'automobiliste U2. Si le chemin O2-X est plus court que le chemin O1-X, alors X peut être un lieu de rencontre.

Si il n'y a aucun X pouvant être un lieu de rencontre, alors il est préférable de ne pas covoiturer car ça rallongerait la durée de voyage des deux automobiliste. Mais si ce n'est pas un problème, on pourrait prendre le deuxième plus court chemin entre O1 et X mais où la moyenne en distance à vol d'oiseau entre les noeuds de ce chemin et O2 soit inférieure à celle du plus court chemin, puis on répète l'étape du paragraphe précédent. S'il n'y a toujours pas de point de rencontre, on prend le troisième plus court chemin respectant les mêmes conditions etc. (pour optimiser on pourra tester seulement les noeuds qui ont changé entre chaque chemin).

Conclusion

Lors de ce bureau d'études en Graphe, nous avons pu implémenter, à l'aide du langage de programmation Java, des algorithmes de plus court chemin ainsi que les outils (comme le tas binaire et les labels) servant à leur implémentation. Pour cela nous avons dû passer par un travail de réflexion et de conception tout en appliquant les notions vues en graphe cette année. Nous avons aussi appris à utiliser des outils de programmation en groupe comme git (et le dépot github), qui ont permis de programmer à plusieurs sur un même programme beaucoup plus simplement et efficacement, ainsi que d'archiver, versionner les programmes afin de pouvoir les récupérer facilement si besoin. Nous avons pu aussi apprendre à réaliser des tests sur les algorithmes implémentés à l'aide de programmes que nous avons fait et d'outils mis à disposition, ce qui est essentiel lors de gros projet (en entreprise par exemple) et qui nous à donc permis d'acquérir de l'expérience dans ce domaine là sachant que c'est quelques chose que nous avions très peu fait.

Nous avons eu beaucoup de problèmes lors des implémentations des algorithmes que nous avons réussi à régler pour la plupart. Il en reste certains (comme l'algorithme A*) mais nous pensons qu'avec plus de temps nous aurions trouvé une solution et d'où venait le problème. Nous aurions aussi aimé avoir le temps d'implémenter les problèmes ouverts car nous les trouvons intéressant.

Nous avons apprécié ce projet car il était complexe mais concret et s'approchait beaucoup de ce à quoi aurait pu ressembler l'implémentation de programmes de géolocalisation, et nous a permis de découvrir l'organisation de projets complexes en groupe.