NOZAHIC Morvan
PELOUS Thomas

Rapport de Be Graphe

Introduction

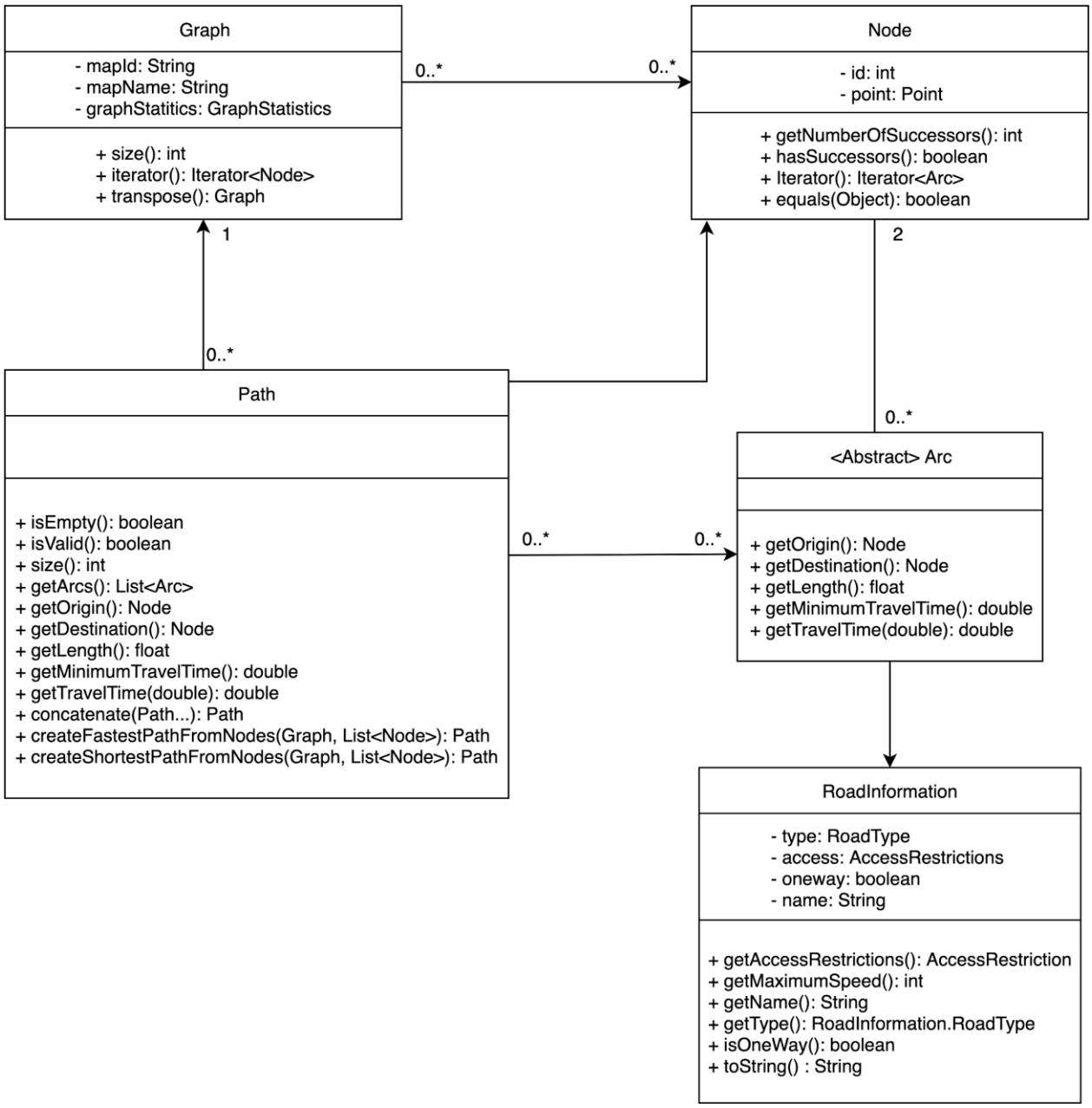
Dans le cadre du cours de Graphes et de Programmation Objet, nous avons pu mettre en pratique nos connaissances acquises dans un bureau d'étude en java.
Nous avions comme objectif de comprendre et d'implémenter deux algorithmes de parcours de graphes ainsi que de réfléchir à un problème ouvert utilisant ces algorithmes de parcours. Les algorithmes de parcours sont l'algorithme de Dijkstra et l'algorithme A*. Le problème ouvert que nous avons choisi est celui du covoiturage.

I. Implémentation

Ce bureau d'étude se veut le plus proche possible d'un projet que l'on pourrait avoir à développer en entreprise. Afin de pouvoir coder sur le même projet en même temps, nous avons utilisé un logiciel de gestion de version, git.
La structure du programme étant déjà donnée, il ne nous restait qu'à implémenter les algorithmes et les quelques classes dont ils avaient besoin.
Nous avons implémenté la fonction "Remove" d'un tas pour ensuite pouvoir développer l'algorithme de Dijkstra.
Nous avons pu découvrir le concept de design pattern Observer qui permet de notifier des objets d'autres classes sans devoir multi-threader notre programme.
Lors des tests sur la carte « carré dense », nous pouvons voir le comportement de l'algorithme de Dijkstra qui se répand dans toutes les directions. A contrario, l'algorithme A* suit une ligne directrice dans la direction de la destination.

A. Documents de conception: Diagramme UML

Afin de bien comprendre la documentation créée par Javadoc et le code qui nous a été fourni, nous avons dû concevoir un diagramme UML synthétique des classes Graph, Node, Arcs, Path et RoadInformation:



Une fois ce diagramme fait, nous avons pu commencer à implémenter les différentes méthodes demandées en sachant à chaque fois où aller chercher les méthodes des différentes classes.
Suite à cela, nous avons eu à coder toutes les méthodes marquées “deprecated” de la classe Path ainsi que la méthode Remove de la classe BinaryHeap. Ces méthodes sont essentielles pour pouvoir implémenter l’algorithme de Dijkstra.

B. Algorithme de Dijkstra:

1. Code de Label

```java
package org.insa.graph;

public class Label implements Comparable<Label> {
    private Node courant;
    private boolean marque;
    private double cout;
    private Arc pere;
    private boolean insert;

    public Label(Node cour, boolean marq, double cout, Arc papa) {
        this.courant=cour;
        this.marque=marq;
        this.cout=cout;
        this.pere=papa;
        this.insert=false;
    }

    public int compareTo(Label other) {
        if(this.getTotalCost()==other.getTotalCost()) {
            return 0;
        }else if(this.getTotalCost()>other.getTotalCost()) {
            return 1;
        }else{
            return -1;
        }
    }

    public void setMark(boolean B) {
        this.marque=B;
    }

    public double getCost() {
        return this.cout;
    }


    public void setCost(double new_cost) {
        this.cout = new_cost;
    }

    public void setFather(Arc new_father) {
        this.pere = new_father;
    }

    public void setInsert() {
        this.insert = true;
    }

    public boolean getInsert() {
        return this.insert;
    }

    public boolean isMarked() {

    public Node getNode() {
        return this.courant;
    }
    public Arc getFather() {
        return this.pere;
    }

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

}
```

2. Code de Dijkstra

```java
package org.insa.algo.shortestpath;

import java.util.*;



public class DijkstraAlgorithm extends ShortestPathAlgorithm {

    int i=1;
    private double cout;
    private boolean validity=true;
    private int nbIteration=0;


    // verifie la validite du tas dans le Dijsktra
    public boolean IsValid() {
        if (this.validity)
            return true;
        else
            return false;
    }

    //retourne le nombre d'iterration de l'algorithme
    public int nbIter() {
        return this.nbIteration;
    }

    //permet d'incrémentation l'iterration
    public void IncrementIter() {
        this.nbIteration++;
    }

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

    protected Label newLabel(Node cour, boolean marq, double cout, Arc papa,ShortestPathData data) {
        Label L=new Label(cour, marq, cout, papa);
        return L;
    }

    public double getCout() {
        return this.cout;
    }
```

```java
@Override
protected ShortestPathSolution doRun() {

    ShortestPathData data = getInputData();
    Graph graph = data.getGraph();
    notifyOriginProcessed(data.getOrigin());
    final int nbNodes = graph.size();

    //Chemin: liste noeuds mini qui sortent du tas (il faut ensuite choisir le bon noeud pour remonter le chemin)
    //un noeud sortant du tas n'est pas forcément impliqué dans le chemin le plus court
    //tablab: liste noeuds rencontre pendant l'algorithme

    Label tablab[]=new Label[nbNodes];

    //Tas_label: tas nécessaire à l'alorithme de Dijkstra
    BinaryHeap<Label> Tas_label= new BinaryHeap<Label>();

    Label x=newLabel(data.getOrigin(),false,0.0,(Arc)null, data);
    tablab[x.getNode().getId()]=x;
    tablab[x.getNode().getId()].setInsert();
    Tas_label.insert(x); //insere le label du point d'origine
```

```
77          while(!Tas_label.isEmpty() && x.getNode()!=data.getDestination()) {
78
79              //Test de la validité du Tas mis en commentaire car ralentit l'algorithme
80              /*if(!Tas_label.IsValid()) {
81                  this.validity=false;
82              }*/
83
84              this.IncrementIter();
85              x = Tas_label.deleteMin();
86
87              //informe les observateurs qu'un noeud a ete marque
88              notifyNodeMarked(tablab[x.getNode().getId()].getNode());
89
90              tablab[x.getNode().getId()].setMark(true);
91
92              for(Arc successeur : tablab[x.getNode().getId()].getNode().getSuccessors()){
93
94                  //y n'appartienenet pas a tablab on l'ajoute
95
96                  if(tablab[successeur.getDestination().getId()]==null) {
97
98                      tablab[successeur.getDestination().getId()]=newLabel(successeur.getDestination(),false,Double.MAX_VALUE,successeur, data);
99
100                     //informe les observateurs qu'un noeud a ?t? visit? pour la premi?re fois
101                     notifyNodeReached(tablab[successeur.getDestination().getId()].getNode());
102
103                     tablab[successeur.getDestination().getId()].setInsert();
104                     Tas_label.insert(tablab[successeur.getDestination().getId()]);
105                 }
106
107                 // test si y marked on saute le if
108                 if(!tablab[successeur.getDestination().getId()].isMarked()){
109
110                     if(tablab[successeur.getDestination().getId()].getCost()>tablab[x.getNode().getId()].getCost() + data.getCost(successeur)) {
111
112                         //Cost(y)=cost(x)+W(x,y)
113                         if(tablab[successeur.getDestination().getId()].getInsert()) {
114                             Tas_label.remove(tablab[successeur.getDestination().getId()]);
115                         }
116
117                         //pour faire la mise a jour:
118                         //on supprime et on remet avec le getCost a jour
119                         tablab[successeur.getDestination().getId()].setCost(tablab[x.getNode().getId()].getCost() + data.getCost(successeur));
120                         tablab[successeur.getDestination().getId()].setFather(successeur);
121                         Tas_label.insert(tablab[successeur.getDestination().getId()]);
122                     }
123                 }
124             }
125         }
126
127         ShortestPathSolution solution = null;
128
129         //Destination has no predecessor, the solution is infeasible...
130         if (tablab.length==0 || tablab[data.getDestination().getId()]==null) {
131             solution = new ShortestPathSolution(data, Status.INFEASIBLE);
132         }
133         else {
134
135             // The destination has been found, notify the observers.
136             notifyDestinationReached(data.getDestination());
137
138             // Create the path from the arrayList of Node...
139             ArrayList<Arc> arcs = new ArrayList<>();
140
141             Label courant=tablab[data.getDestination().getId()];
142
143             while(courant.getNode().getId()!=data.getOrigin().getId()) {
144                 arcs.add(courant.getFather());
145                 i++;
146                 //cout+=courant.getCost();
147                 courant=tablab[courant.getFather().getOrigin().getId()];
148             }
149         Collections.reverse(arcs);
150
151         // Create the final solution.
152         solution = new ShortestPathSolution(data, Status.OPTIMAL, new Path(graph, arcs));
153         }
154
155             return solution;
156     }
157 }
158
```

3. Explication du code

Pour implémenter l'algorithme de Dijkstra, nous avons eu besoin d'une nouvelle classe "Label" permettant de sauvegarder les caractéristiques de chaque sommet. Un Label contient donc:

- Le sommet courant qui sera le sommet associé à ce Label
- Un attribut marque qui est un boolean indiquant si le sommet a été marqué ou non
- Son coût
- Le père qui correspondra au sommet précédent sur le chemin correspondant au plus court chemin.

Une fois la classe Label créée, nous avons pu coder Dijkstra. Nous nous sommes inspirés du pseudo-code présent dans le polycopié du cours. Néanmoins, nous avons choisi de ne pas initialiser en début d'algorithme le coût de tous les sommets à l'infinie. Nous initialisons les sommets seulement lorsque l'algorithme les rencontres.
Nous disposons de plusieurs structures de données. Un tas dans lequel les Labels sont rangés par ordre de coût ainsi qu'un tableau qui recense tous les Labels qui ont été rencontrés pendant l'algorithme.
Tant que le tas n'est pas vide ou que le noeud minimum du tas n'est pas le noeud de Destination, alors, on sort le noeud de coût minimum, on le marque, on met à jour tous ses successeurs si l'ancien coût est supérieur au nouveau.
Lorsque l'algorithme se termine, on cherche le prédécesseur du Label de Destination et ainsi de suite jusqu'à arriver au Label d'Origine. On a plus qu'à inverser la liste obtenue pour avoir le chemin optimal.

C. Algorithme de A*

1. Code de LabelStar

```java
package org.insa.graph;

import org.insa.algo.AbstractInputData.Mode;
public class LabelStar extends Label {

    private double coutEstime;
    public LabelStar(Node cour, boolean marq, double cout, Arc papa, ShortestPathData data){
        super(cour,marq,cout,papa);
        setEstimation(data);
    }


    public double getTotalCost() {
        return this.getCost()+this.coutEstime;
    }

    public void setEstimation(ShortestPathData data) {
        if (data.getMode()==Mode.LENGTH) {
            this.coutEstime=Point.distance(this.getNode().getPoint(), data.getDestination().getPoint());
        }else if(data.getMode()==Mode.TIME) {
            this.coutEstime=Point.distance(this.getNode().getPoint(), data.getDestination().getPoint())/(Math.max(data.getGraph().getGraphInformation().getMaximumSpeed(), data.getMaximumSpeed())/3.6);
        }
    }
}

```

2. Code de A*

```java
package org.insa.algo.shortestpath;

import java.util.ArrayList;

public class AStarAlgorithm extends DijkstraAlgorithm {

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

    @Override
    protected Label newLabel(Node cour, boolean marq, double cout, Arc papa, ShortestPathData data) {
        Label LS = new LabelStar(cour, marq, cout, papa, data);
        return LS;
    }

}
```

3. Explication de A*

A* est une extension de l'algorithme de Dijkstra ce qui explique que son code est court. C'est pourquoi la classe de A* hérite de la classe DijsktraAlgorithm. Dans A*, nous ne rangeons plus les sommets par seul ordre de coût mais avec une fonction de deux coûts: coût par rapport à l'origine plus coût estimé jusqu'à la destination.
Tout comme pour Dijkstra, nous avons dû créer une classe LabelStar pour sauvegarder nos informations sur les noeuds visités. LabelStar contient en plus des propriétées héritées de Label, un attribut "Coût estimé". Il correspond à la distance entre le noeud courant et la destination finale à vol d'oiseau dans le cas d'une recherche de chemin le plus court. Dans le cas de la recherche du chemin le plus rapide, c'est la distance restante à vol d'oiseau divisé par la la vitesse maximum autorisée. Ces deux estimations, appelés heuristiques, permettent de guider l'algorithme vers la destination.
Pour que l'algorithme de Dijkstra puisse s'adapter en celui de A* en fonction des besoins, nous créons une méthode newLabel dans l'algorithme de Dijkstra que nous redéfinissons dans celui de A*. Cette méthode nous permet ainsi de créer un Label si on est dans Dijsktra et un LabelStar si nous sommes dans A*.

II. Tests de correction

Nous avons pu prendre en main le Framework JUnit qui permet de tester notre programme de manière automatique et non pas de seulement faire des affichages de nos résultats. Nous avons réalisé tous nos tests grâce à ce Framework.
Nous avons mené divers tests de corrections portants sur des cartes fictives et réelles. Ces tests ont été validés. Les tests portaient sur des chemins les plus courts en distance ainsi que sur la recherche de chemin les plus rapides en temps. Nous les avons faits pour l'algorithme de Dijkstra et A*.

Pour réaliser nos tests de correction nous avons utilisés le graphe (cf fig.1) proposés dans l'énoncé du BE (il a ensuite été retiré mais nous l'avions fini avant):

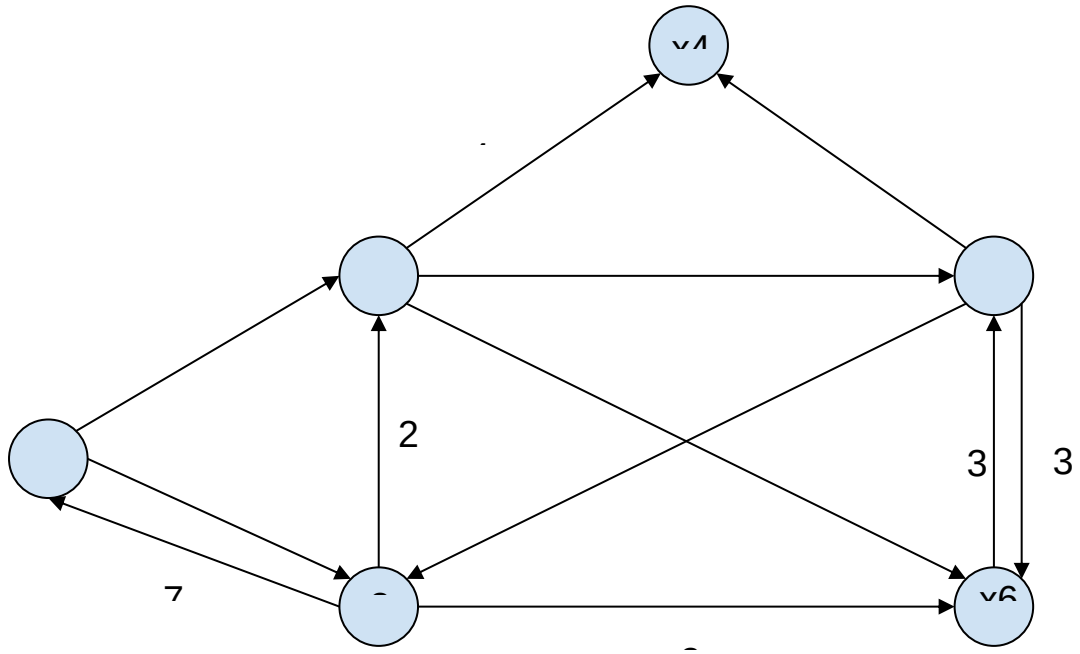


fig. 1: Graphe de test

Résultat attendu:

	x1	x2	x3	x4	x5	x6
x1	-	7,x1	8,x1	10,x5	8,x2	10,x3
x2	10,x3	-	3,x5	3,x5	1,x2	4,x5
x3	7,x3	2,x3	-	5,x5	3,x2	2,x3
x4	∞	∞	∞	-	∞	∞
x5	9,x3	4,x3	2,x5	2,x5	-	3,x5
x6	12,x3	7,x3	5,x5	5,x5	3,x6	-

Ci-dessus le tableau des plus courts chemins depuis un point de départ (colonne de gauche) jusqu'à la destination (ligne du haut). Nous avons ainsi fait les tests de notre Dijkstra sur ce graphe et tous les résultats ont été justes (le fichier des résultats édité est en annexe p.16). Ensuite nous avons comparé les résultats avec ceux de l'algorithme de Bellman-Ford fourni pour vérifier que nous obtenions bien exactement le même chemin, voici quelques exemples:

Chemin nul : Carré dense

Algorithme	Mode	Origine	Destination	Coût
Dijkstra	Shortest	133444	133444	0
A*	Shortest	133444	133444	0
Dijkstra	Fastest	12970	12970	0
A*	Fastest	12970	12970	0

Chemin inexistant : Carte INSA

Algorithme	Mode	Origine	Destination	Coût
Dijkstra	Shortest	1037	955	No path found
A*	Shortest	1037	955	No path found
Dijkstra	Fastest	1037	955	No path found
A*	Fastest	1037	955	No path found

Carte midi-pyrénées:

Algorithme	Mode	Origine	Destination	Coût
Bellman-Ford	Shortest	50911	526981	157,240km

Dijkstra	Shortest	50911	526981	157,240 km
Bellman-Ford	Fastest	50911	526981	1h48min27s
Dijkstra	Fastest	50911	526981	1h48min27s
Bellman-Ford	Shortest	514015	459008	203,568 km
Dijkstra	Shortest	514015	459008	203,569 km
Bellman-Ford	Fasttest	514015	459008	2h39min0s
Dijkstra	Fasttest	514015	459008	2h39min0s

Carte Carré-dense:

Algorithme	Mode	Origine	Destination	Coût
Bellman-Ford	Shortest	201949	324703	41,754 km
Dijkstra	Shortest	201949	324703	41,754 km
Bellman-Ford	Fastest	33185	306613	1h21min56s
Dijkstra	Fastest	33185	306613	1h21min56s

Suite à ces différents tests nous pouvons être sur que notre algorithme de Dijkstra fonctionne correctement et trouve le bon chemin à chaque fois.

III. Tests de performances

Nous avons repris les tests de corrections auxquels nous avons ajouté deux appels système pour connaître le temps d'exécution des tests pour chaque algorithme. Nous avons pu remarquer que l'algorithme de Bellman-Ford est le plus lent suivi de l'algorithme de Dijkstra, enfin l'algorithme A* est le plus rapide. Nous avons aussi vérifié en même temps que les chemins soient égaux entre Dijkstra et A*.

Voici quelques uns des résultats obtenus:

Carte Bretagne:

Algorithme	Mode	Origine	Destination	Coût	Temps exécution
Dijkstra	Shortest	1504	450393	189,1934 km	655 ms
A*	Shortest	1504	450393	189,1934 km	595 ms
Dijkstra	Shortest	185614	85086	128.3032 km	540 ms
A*	Shortest	185614	850393	128.3032 km	372 ms
Dijkstra	Fastest	185614	85086	84.6632 min	747 ms
A*	Fastest	185614	85086	84.6632 min	629 ms

Carte Midi-Pyrénées:

Algorithme	Mode	Origine	Destination	Coût	Temps exécution
Dijkstra	Fastest	185614	85086	2h6min14s	1289 ms
A*	Fastest	185614	85086	2h6min14s	1279 ms
Dijkstra	Fastest	18	85086	1h36min54s	1380 ms
A*	Fastest	18	85086	1h36min54s	1279 ms
Dijkstra	Shortest	18	886	4.1474 km	171 ms
A*	Shortest	18	886	4.1474 km	44 ms
Dijkstra	Shortest	12000	15641	13.8563 km	112 ms
A*	Shortest	12000	15641	13.8563 km	51 ms
Dijkstra	Shortest	4219	15	28.4625 km	159 ms
A*	Shortest	4219	15	28.4625 km	79 ms

Carte Belgique:

Algorithme	Mode	Origine	Destination	Temps exécution
Dijkstra	Fastest	966454	8836	1430 ms
A*	Fastest	966454	8836	1454 ms

Dijkstra	Shortest	538885	12354	714 ms
A*	Shortest	538885	12354	204 ms
Dijkstra	Shortest	15641	124	579 ms
A*	Shortest	15641	124	133 ms

Nous remarquons que l'algorithme A* est bien plus rapide que l'algorithme de Dijkstra, cependant l'écart se réduit lorsqu'il s'agit du mode fastest, A* semble donc plus adapté pour les plus courts chemins (mode Shortest).

IV. Problème ouvert

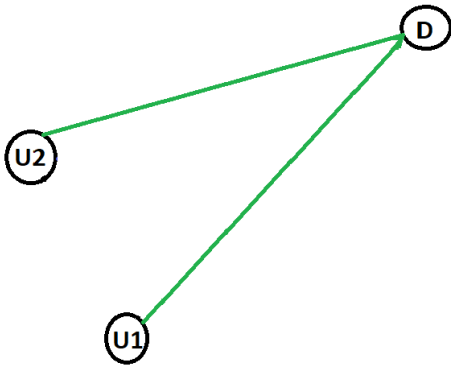
Nous avons choisi le problème du covoiturage. Deux usagers veulent covoiturer, nous devons trouver le point de rencontre optimal pour qu'ils effectuent le moins de kilomètres. Ils peuvent faire le trajet chacun de leur côté dans le pire des cas. Nous avons choisi le problème dans sa version la plus simple lorsqu'il n'y a que deux covoitureurs qui ont la même destination.

Voici l'algorithme utilisé:

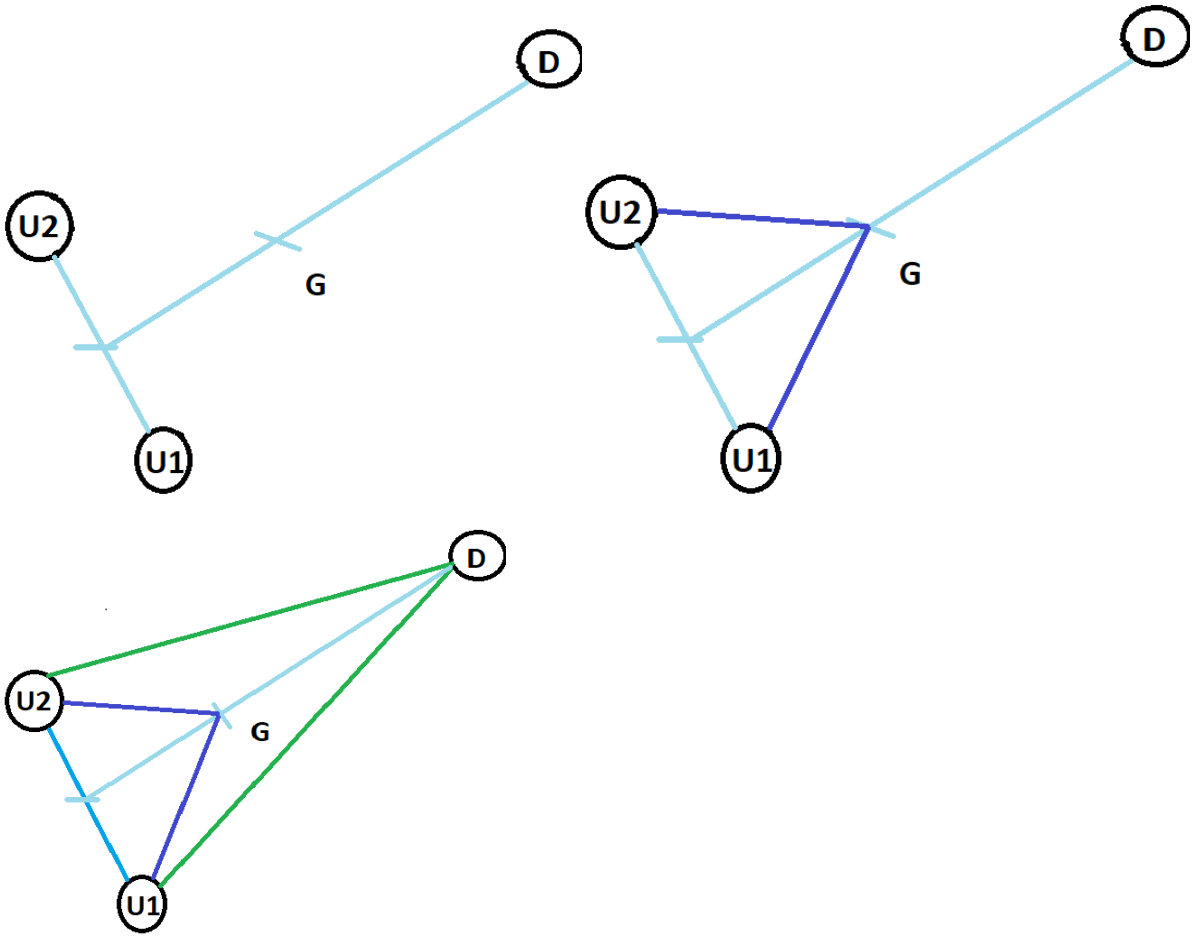
Faire un A* entre U1 et D: A*1, et entre U2 et D: A*2
Faire un A* entre U1 et U2: A*3
Trouver le milieu du trajet U1-U2 : I
Faire un A* entre I et D :A*4
Trouver le point situé à ⅓ de la distance en partant de I : G
Faire un A* entre U1 et G, A*5, et entre U2 et G, A*6
Faire un A* entre G et D: A*7

If((A*1+A*2)<(A*5+A*6+A*7){
Path=A*1+A*2
}else{Path=A*5+A*6+A*7}

On a représenté ce problème sous la forme d'un triangle, avec pour sommets les usagers et la destination. Nous avons trouvé que le point de rencontre optimal s'il existe est le centre de gravité de ce triangle.
Nous trouvons donc le chemin le plus rapide entre les deux usagers. S'il est plus long que le chemin de la somme des trajets entre les usagers et la destination alors nous décidons que le covoiturage n'est pas une solution optimale.



Ensuite nous calculons le point qui est le plus proche de la moitié du coût du chemin le plus court entre les deux usagers, appelé Intersection.
Nous trouvons le plus court chemin entre intersection et la destination. Le centre de gravité se trouve à 1/3 de distance à partir de Intersection.



Le chemin optimal pour chaque usager est donc le trajet jusqu'au centre de gravité puis jusqu'à leur destination.

```java
public abstract class CarPoolingAlgorithm extends AbstractAlgorithm<CarPoolingObserver> {

    protected CarPoolingAlgorithm(CarPoolingData data) {
        super(data);
    }

    @Override
    public CarPoolingSolution run() {
        return (CarPoolingSolution) super.run();
    }

    @Override
    protected  CarPoolingSolution doRun() {


        CarPoolingData data = getInputData();
        Graph graph = data.getGraph();
        CarPoolingSolution solution=null;
        //Calcul du chemin le plus court entre U1 et U2
        ShortestPathData U1U2=new ShortestPathData(graph,graph.get(data.getOriginU1().getId()),graph.get(data.getOriginU2().getId()),data.getArcInspector());
        AStarAlgorithm AU1U2 = new AStarAlgorithm(U1U2);
        ShortestPathSolution solutionU1U2 = AU1U2.run();

        //Calcul du chemin le plus court entre U1 et D1 et U2 et D1
        ShortestPathData U1D1=new ShortestPathData(graph,graph.get(data.getOriginU1().getId()),graph.get(data.getDestinationD1().getId()),data.getArcInspector());
        AStarAlgorithm AU1D1 = new AStarAlgorithm(U1D1);
        ShortestPathData U2D1=new ShortestPathData(graph,graph.get(data.getOriginU2().getId()),graph.get(data.getDestinationD1().getId()),data.getArcInspector());
        AStarAlgorithm AU2D1 = new AStarAlgorithm(U2D1);

        //on compare si le chemin entre U1 et U2 est plus long que le chemin dans le cas où il ne font pas de bla-bla
        ShortestPathSolution solutionU1D1 = AU1D1.run();
        ShortestPathSolution solutionU2D1 = AU2D1.run();

        if(solutionU1U2.getPath().getLength()>(solutionU1D1.getPath().getLength()+solutionU2D1.getPath().getLength())){
            solution= new CarPoolingSolution(data, Status.OPTIMAL, solutionU1D1.getPath(), solutionU2D1.getPath());

        }

        //on trouve la médiane puis on fait aller U1 et U2 jusqu'au centre de gravité
        else {

            //on trouve le milieu entre U1 et U2
            float cout=0;
```

```
55            Node Intersection;
56            int i=0;
57            while(i<solutionU1U2.getPath().getArcs().size() && cout<(solutionU1U2.getPath().getArcs().get(i).getLength()/2) ) {
58                cout = cout + solutionU1U2.getPath().getArcs().get(i).getLength();
59            }
60            Intersection = solutionU1U2.getPath().getArcs().get(i).getOrigin();
61
62            //Calcul du chemin le plus court entre Intersection et D1 (=médiane du triangle U1 D1 U2)
63            ShortestPathData IntersectionD1=new ShortestPathData(graph,Intersection,graph.get(data.getDestinationD1().getId()),data.getArcInspector());
64            AStarAlgorithm AIntersectionD1 = new AStarAlgorithm(IntersectionD1);
65            ShortestPathSolution solutionIntersectionD1 = AIntersectionD1.run();
66
67            //trouve le noeud représentant le centre de gravité (situé à 1/3 de la médiane en partant de Intersection)
68            cout=0;
69            Node centreGravite;
70            i=0;
71            while(i<solutionIntersectionD1.getPath().getArcs().size() && cout<(solutionIntersectionD1.getPath().getArcs().get(i).getLength()/3) ) {
72                cout = cout + solutionIntersectionD1.getPath().getArcs().get(i).getLength();
73            }
74            centreGravite = solutionIntersectionD1.getPath().getArcs().get(i).getOrigin();
75
76            //Calcul du chemin le plus court entre U1 - centreGravite et U2 - centreGravite
77            ShortestPathData U1G=new ShortestPathData(graph,graph.get(data.getOriginU1().getId()),centreGravite,data.getArcInspector());
78            AStarAlgorithm AU1G = new AStarAlgorithm(U1G);
79            ShortestPathSolution solutionU1G = AU1G.run();
80            ShortestPathData U2G=new ShortestPathData(graph,graph.get(data.getOriginU2().getId()),centreGravite,data.getArcInspector());
81            AStarAlgorithm AU2G = new AStarAlgorithm(U2G);
82            ShortestPathSolution solutionU2G = AU2G.run();
83            solution= new CarPoolingSolution(data, Status.OPTIMAL, solutionU1G.getPath(), solutionU2G.getPath());
84        }
85        return solution;
86
87    }
88
89    @Override
90    public CarPoolingData getInputData() {
91        return (CarPoolingData) super.getInputData();
92    }
93
94 }
```

Nous avons commencé l'implémentation du problème ouvert mais nous n'avons pas eu le temps de la finir. Les observateurs ne fonctionnent pas lorsque nous lançons un test de Car-Pooling. De plus, nous avons plusieurs paths, nous ne savons pas s'il faut les concaténer ou alors si les résultats peuvent être plusieurs paths différents.

Conclusion

Pour conclure, nous avons appris au cours de ce bureau d'étude à mettre en pratique nos compétences en programmation Java et en graphe sur des cas concrets. Nous avons codé des algorithmes de plus courts chemins: Dijkstra et A* et ainsi observé leurs différences de comportement que ce soit sur les performances ou le fonctionnement de l'algorithme.
De plus, nous avons appris à réaliser une campagne de tests pour vérifier que les algorithmes fonctionnent dans tous les cas possibles: chemins vides, inexistants, court ou encore long tout en s'assurant que les chemins trouvés soient bien les plus courts.
Enfin nous avons pu commencer à travailler sur le problème ouvert pour proposer une solution malgré le fait que nous n'ayons pas eu le temps de finir de coder l'algorithme complètement.

ANNEXE:

Le document créé lors des tests du graphe proposé dans le BE.

sol: Found a path from node #0 to node #1, 0.0080 kilometers in 0 seconds.
Nombre d'iteration : 2

sol: Found a path from node #0 to node #2, 0.0080 kilometers in 0 seconds.
Nombre d'iteration : 3

sol: Found a path from node #0 to node #3, 0.0110 kilometers in 0 seconds.
Nombre d'iteration : 6

sol: Found a path from node #0 to node #4, 0.0090 kilometers in 0 seconds.
Nombre d'iteration : 4

sol: Found a path from node #0 to node #5, 0.0100 kilometers in 0 seconds.
Nombre d'iteration : 5

sol: Found a path from node #1 to node #4, 0.0010 kilometers in 0 seconds.
Nombre d'iteration : 2

sol: Found a path from node #2 to node #4, 0.0030 kilometers in 0 seconds.
Nombre d'iteration : 4

sol: No path found from node #3 to node #4 in 0 seconds.
Nombre d'iteration : 1

sol: org.insa.algo.shortestpath.DijkstraAlgorithm@2d127a61
Nombre d'iteration : 5

sol: Found a path from node #2 to node #0, 0.0080 kilometers in 0 seconds.
Nombre d'iteration : 6

sol: Found a path from node #5 to node #4, 0.0030 kilometers in 0 seconds.
Nombre d'iteration : 2

sol: Found a path from node #1 to node #0, 0.0110 kilometers in 0 seconds.
Nombre d'iteration : 6

sol: Found a path from node #1 to node #5, 0.0040 kilometers in 0 seconds.
Nombre d'iteration : 5

sol: Found a path from node #1 to node #4, 0.0010 kilometers in 0 seconds.
Nombre d'iteration : 0

sol: No path found from node #3 to node #5 in 0 seconds.
Nombre d'iteration : 1

sol: org.insa.algo.shortestpath.DijkstraAlgorithm@2bbaf4f0
Nombre d'iteration : 4

sol: No path found from node #3 to node #0 in 0 seconds.
Nombre d'iteration : 1

sol: Found a path from node #5 to node #1, 0.0070 kilometers in 0 seconds.
Nombre d'iteration : 5

sol: Found a path from node #1 to node #2, 0.0030 kilometers in 0 seconds.
Nombre d'iteration : 4

sol: Found a path from node #2 to node #1, 0.0020 kilometers in 0 seconds.
Nombre d'iteration : 2

sol: No path found from node #3 to node #2 in 0 seconds.
Nombre d'iteration : 1

sol: Found a path from node #4 to node #2, 0.0020 kilometers in 0 seconds.
Nombre d'iteration : 3

sol: org.insa.algo.shortestpath.DijkstraAlgorithm@11c20519
Nombre d'iteration : 4

sol: Found a path from node #4 to node #0, 0.0100 kilometers in 0 seconds.
Nombre d'iteration : 6

sol: Found a path from node #1 to node #3, 0.0030 kilometers in 0 seconds.
Nombre d'iteration : 3

sol: Found a path from node #2 to node #3, 0.0050 kilometers in 0 seconds.
Nombre d'iteration : 5

sol: No path found from node #3 to node #1 in 0 seconds.
Nombre d'iteration : 1

sol: Found a path from node #4 to node #3, 0.0020 kilometers in 0 seconds.
Nombre d'iteration : 2

sol: org.insa.algo.shortestpath.DijkstraAlgorithm@70beb599
Nombre d'iteration : 3

sol: Found a path from node #5 to node #0, 0.0130 kilometers in 0 seconds.
Nombre d'iteration : 6