

Rapport : Projet Graphes et Réseaux

Rapport : Projet Graphes et Réseaux.....	1
Algorithmes Implémentés.....	3
Algorithme de Dijkstra	3
Algorithme de Kruskal.....	8
Méthodes de JGraphT.....	12
Algorithmes : HawickJamesSimpleCycles et CycleDetector	12
Algorithme: BoruvkaMinimumSpanningTree	14
Algorithmes : TwoApproxMetricTSP.....	15
Affichage du graphe.....	17
Récupération des données.....	20
Traitement des données.....	22

Notre groupe a travaillé sur un projet portant sur les réseaux routiers reliant plusieurs villes en France. Nous avons conçu un web scraper afin de récupérer les données sur le site web bonnesroutes.com ainsi qu'un parser chargé de les traiter afin de pouvoir les rendre utilisables.

Les données récupérées contiennent une ville de départ, une ville d'arrivée, ainsi qu'une distance en kilomètres. A partir de cela nous avons instancié des graphes, avec pour sommet des villes, reliées par des arrêtes symbolisant les routes sans détours (routes "directes"). Nous avons ensuite pu appliquer plusieurs méthodes présentes dans la librairie JGraphT permettant de traiter les données et nous avons aussi implémenté plusieurs algorithmes dont le fonctionnement et les résultats sont détaillés dans ce rapport.

L'annexe regroupe des éléments superflus des dossiers tels qu'une modélisation graphique des données ainsi que le détail du processus de récupération et traitement des données en de les rendre utilisables par les méthodes et classes explicitées dans le rapport.

Nous avons utilisé deux sets de données : Un "petit" contenant 9 villes et 36 arêtes, et un "gros" contenant 64 villes et 350 arêtes. Le gros set peut être utilisé avec la plupart des méthodes mais ne peut être efficacement affiché, ou utilisé en raison de la complexité de certains algorithmes. Il sera donc utilisé à la place un jeu de données réduit permettant néanmoins d'apprécier la modélisation. Le projet est divisé en plusieurs packages :

- Application : contenant une classe d'exécution et l'affichage du graphe
- UtilityGraphe, contenant les classes se rapportant à l'algorithmique et implémentation des fonctionnalités liées à la programmation orientée-objet et aux Graphes : Algorithmes, Node, OutilsGraphe.
- Scraper : chargée de récupérer et traiter les données, elle utilise la librairie Jsoup pour récupérer le code source des pages internet.

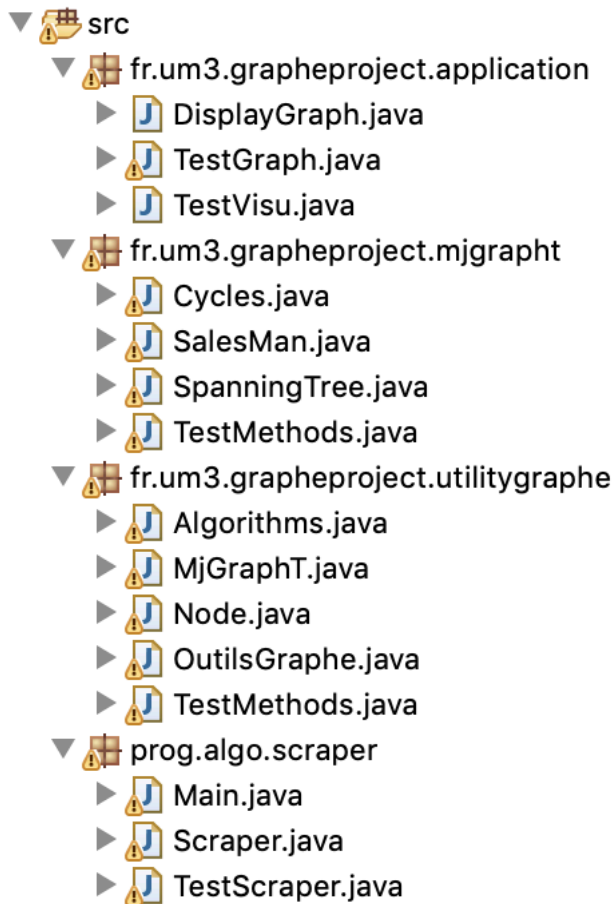
Plusieurs classes ont été nécessaires à la conception du projet.

Le projet se centre autour de l'analyse d'un réseau routier reliant plusieurs villes de France. Les données du réseau récoltées font l'objet ici d'un traitement algorithmique sur des structures de graphes. Les mises en œuvre de 5 algorithmes, dont 2 seront détaillés, feront l'objet de ce rapport en débouchant sur les intérêts qu'ils peuvent porter.

La structure des graphes seront adaptés selon les algorithmes utilisés dans le but de permettre leur bonne fonctionnalité.

L'implémentation et l'utilisation des algorithmes est réalisé en JAVA sous l'IDE Eclipse dont le programme est divisé en quatre packages :

- Application : qui contient les classes d'exécution et d'affichage du graphe
- UtiliyGraphe qui contient les classes Algorthims, OutilsGraphe pour les opérations utiles sur les graphes et Node relatives aux sommets
- Mjgrapht est consacrée aux méthodes présentes dans la librairie jGraphT
- Package Scraper, chargé de récupérer et parser les données



Algorithmes Implémentés

Algorithme de Dijkstra

(Codé par Maximilien Servajean)

Problème : Plus court chemin

Figure 1: Illustration des Classes importées des différentes librairies pour les algorithmes implémentés

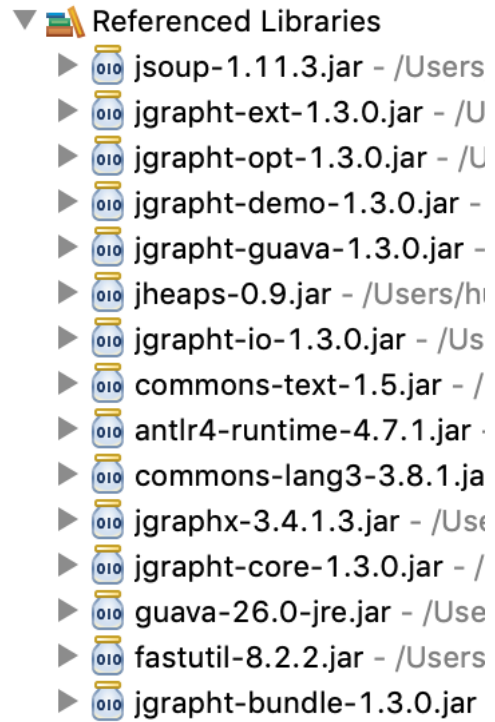


Figure 2 : Librairies utilisées dans le cadre du projet

Nous cherchons à nous rendre à une ville, nous allons y aller en utilisant le réseau routier, nous allons chercher le chemin le plus court à emprunter.

L'algorithme de Dijkstra permet de trouver le plus court chemin dans un graphe valué sans arête de valeur négative.

Structure du graphe

Graphe simple G , orienté valué : $G(V,E,c)$ (`SimpleDirectedWeightedGraph<V,E>`).

V = l'ensemble des villes (= sommets / `Node`)

E = l'ensemble des lignes (= arcs / `DefaultWeightedEdge`) reliant une ville à une autre.

$c(\text{arc})$ = valuation / poids positif d'un arc.

L'algorithme *ShortestPaths* prend en paramètre un graphe simple orienté valué (= structure de graphe géré par `JGraphT`) ainsi que l'identifiant d'un sommet source et affiche l'ensemble des potentiels minimaux des sommets (= le coût du chemin le plus court du sommet source au sommet considéré)

```
public static void ShortestPaths(SimpleDirectedWeightedGraph<Node, DefaultWeightedEdge> g, int id) {
```

Préparatifs

Instanciation d'une liste de sommets S pour vérifier lesquelles ont déjà été analysé, ainsi que d'un pointeur p référençant le sommet courant que l'on étudie.

1er Phase : Initialisation

Une boucle *for* est lancé sur l'ensemble des sommets du graphe vérifiant pour chacun d'entre eux si son *id* est celui du sommet source, dans ce cas on référence le pointeur p sur ce sommet étant le sommet source, ainsi si présent on l'ajoute dans notre liste de sommets S car il est le premier à être analysé, on met son potentiel à 0 et on le marque comme visité avec la méthode *toggleVisnbVilled()*. Tous les potentiels des autres sommets sont à *Double.MAX_VALUE* c'est à dire à $+\infty$ et seront mené à être modifier au cours de l'algorithme. Ainsi d'une part on initialise les potentiels et d'autre part nous vérifions si l'*id* du sommet source correspond bien à un sommet du graphe dans le cas contraire l'algorithme s'arrête.

```
List<Node> S = new ArrayList<Node>();
Node p = null;
for (Node n : g.vertexSet()) {

    if (n.getId() == id) {
        p = n;
        S.add(p);
        p.setPotentiel(0);
        p.toggleVisnbVilled();
    } else {
        n.setPotentiel(Double.MAX_VALUE);
    }
}
if (p == null) return;
```

2ème Phase : Parcours du graphe

On entre dans une boucle *while* ne s'arrêtant que si la liste S ne contient pas tous les sommets du graphe.

```
while (g.vertexSet().size() != S.size()) {
```

Nous affectons le potentiel du sommet courant à la variable *potentielActuel*. A l'aide d'une boucle *for* on étudie, en utilisant l'ensemble des arcs sortant du sommet courant, l'ensemble de ses successeurs, puis pour chacun d'entre eux, s'il n'a pas déjà été visité (*isVisnbVilled()* ?),

```
double potentielActuel = p.getPotentiel();
for (DefaultWeightedEdge edge : g.outgoingEdgesOf(p)) {
    Node x = g.getEdgeTarget(edge);
    if (!x.isVisnbVilled()) {
        double weight = g.getEdgeWeight(edge);
        if (x.getPotentiel() > potentielActuel + weight) {
            x.setPotentiel(potentielActuel + weight);
            x.setPredecesseur(p);
        }
    }
}
```

car le potentiel de ceux visités ne peut plus être amélioré), on regarde si son potentiel peut être amélioré. En outre si le potentiel du successeur est supérieur au potentiel du prédécesseur + le coût de l'arc les reliant alors on modifie son potentiel car il existera un plus court chemin passant par le prédécesseur (stocké dans l'attribut prédécesseur du Node successeur) du sommet source jusqu'au successeur lui-même.

```
Node x = null;
double smallestPotentiel = 0;
for (Node n : g.vertexSet()) {
    if (!n.isVisnbVilled() && (x == null || n.getPotentiel() < smallestPotentiel)) {
        x = n;
        smallestPotentiel = n.getPotentiel();
    }
}
S.add(x);
x.toggleVisnbVilled();
p = x;
```

Une fois le sommet étudié, pour la sélection du prochain nous parcourons les autres sommets du graphe n'ayant pas encore été analysés (*!n.isVisnbVilled()* ?) puis est sélectionné parmi eux le sommet ayant le potentiel le moins élevé, ainsi ce dernier devient le nouveau sommet courant et on l'ajoute à la liste *S*, on le marque comme visité et on le référence par le pointeur *p*.

Nous réitérons ce processus jusqu'à ce que tous les sommets soient visités et ajoutés à la liste *S*, ce qui nous assurera d'avoir l'ensemble des potentiels minimaux ainsi que l'ensemble des plus courts chemins du sommet source vers les autres (avec une petite modification de l'algorithme).

Remarque* : Les sommets restant à $+\infty$ seront donc des sommets impossibles à atteindre depuis le sommet source

3ème Phase : Affichage des potentiels

Pour conclure nous affichons les potentiels.

```
for (Node n : g.vertexSet()) {
    if (n.getPotentiel() < 1000000) {
        System.out.println("nId: "+UtilsGraphe.getBigSommetVilles().get(n.getId()-1).getId()+
            "\nVille: "+UtilsGraphe.getBigSommetVilles().get(n.getId()-1).getVille()+"\nPotentiel: "
            +n.getPotentiel()+"\n"); /*on affiche tous les potentiels*/
    } else if (n.getPotentiel() > 1000000) {
        System.out.println("nId: "+UtilsGraphe.getBigSommetVilles().get(n.getId()-1).getId()+
            "\nVille: "+UtilsGraphe.getBigSommetVilles().get(n.getId()-1).getVille()+
            "\nPotentiel: +\infty\n ne peut accéder à "+UtilsGraphe.getBigSommetVilles().get(n.getId()-1).getVille()+
            " depuis "+UtilsGraphe.getBigSommetVilles().get(n.getId()-1).getVille()); /*on affiche tous les potentiels*/
    }
}
```

Intérêts de l'algorithme de Dijkstra :

Les usages de cet algorithme sont divers et variés s'axent généralement autour d'une notion de distance relative à un point du graphe. Ainsi l'on peut par exemple savoir au voisinage d'une ville celle qui est la plus proche, ou déterminer un itinéraire optimal menant à ville précise.

Résultats du test :

```
<terminated> TestGraph [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101-jre/bin/java
On ne peut accéder à Perpignan depuis Verdun

Id: 61
Ville: Villeurbanne
Potentiel: 674.0

Id: 62
Ville: Vitré
Potentiel: +∞
On ne peut accéder à Perpignan depuis Vitré

Id: 63
Ville: Yssingeaux
Potentiel: 769.0

Id: 64
Ville: Étaples
Potentiel: +∞
On ne peut accéder à Perpignan depuis Étaples
Le sommet d'arrivée n'existe pas ou est inatteignable
```

Si le potentiel est marqué $+\infty$ alors il n'est pas accessible depuis la ville susmentionnée.

Algorithme de Kruskal

(Codé par les membres du projet)

Problème : Arbre de poids minimum

L'Etat souhaite rénover son réseau routier, délabré à de nombreux endroits, et encombré en période de vacances. Cependant il doit dans le même temps procéder à des économies, il souhaite investir aussi efficacement que possible ses ressources dans le chantier sans laisser aucune ville de côté : Il va rénover de telle sorte que chaque ville soit reliée au nouveau réseau autoroutier, par une seule route, la plus courte possible.

L'algorithme de Kruskal construit un arbre de poids minimum : il assure une inter-connectivité avec l'ensemble des points du réseau mais également un coût minimal en poids

Structure du graphe

Graphe simple G' , non orienté valué : $G'(V,E,c)$
(DefaultUndirectedWeightedGraph<V,E>).

V = l'ensemble des villes (= sommets / Node)

E = l'ensemble des lignes (= arêtes/ DefaultWeightedEdge) reliant une ville à une autre.

$c(\text{arête})$ = valuation / poid positif d'un arc.

L'algorithme KruskalAlgo prend en paramètre un graphe non orienté valué (= structure de graphe géré par JGraphT) et affiche l'ensemble des arêtes constituant l'arbre de poids minimum, c'est à dire un graphe connexe et sans cycle (= chaîne fermée et simple) dont la somme du poids des arêtes est minimale.

```
Graph<Node,DefaultWeightedEdge> util = new DefaultUndirectedWeightedGraph<>(DefaultWeightedEdge.class);
List<Double> Weights = new ArrayList<Double>();
List<DefaultWeightedEdge> ForTri = new ArrayList<DefaultWeightedEdge>();
List<DefaultWeightedEdge> Res = new ArrayList<DefaultWeightedEdge>();
```

```
public static void KruskalAlgo(DefaultUndirectedWeightedGraph<Node, DefaultWeightedEdge> g)
```

Préparatifs :

Instanciation d'un graphe non orienté *util* annexe qui sera utile lors de la deuxième phase de l'algorithme relatif à la construction de l'arbre, ainsi que *ForTri* une liste d'arêtes et *Weights* une liste contenant la taille des arêtes, tous deux utilisés lors de la première phase concernant le tri. Et enfin *Res*, liste qui contiendra l'ensemble des arêtes finales.

Donc nous ajoutons à *util* l'ensemble des nœuds du graphe, passés en paramètre. En quelque sorte, une copie sans les arêtes, à *Weights* l'ensemble des poids du graphe puis enfin à *ForTri* les arêtes eux-mêmes.

```
for(Node n : g.vertexSet()) {  
    util.addVertex(n);  
}  
  
for(DefaultWeightedEdge e: g.edgeSet()) {  
    Weights.add(g.getEdgeWeight(e));  
    ForTri.add(e);  
}
```

1^{ère} Phase : Tri des longueurs en ordre croissant :

La liste *Weights* des poids est ainsi triée pour faire en sorte ensuite d'ajouter à *Res* l'ensemble des arêtes dans l'ordre adéquat. Pour cela on utilise une boucle *for* parcourant *Weights* de manière à ce que pour chaque poids on trouve son arête correspondante dans *ForTri*, grâce à une boucle *while*, pour ensuite l'ajouter dans *Res* tout en veillant à le supprimer de *ForTri* pour éviter tout problème de doublons d'arêtes afin de ne pas ajouter deux fois la même arête dans *Res* (le graphe obtenu à partir des données étant arêtes multiples). Une fois l'arête trouvée, la valeur du *boolean* passe alors à *true*, ce qui arrête la boucle et ainsi de suite.

```
Collections.sort(Weights);  
  
for(int i = 0; i < Weights.size();i++) {  
    boolean flag = false;  
    int k = 0;  
    while(k < ForTri.size() && flag == false) {  
        if(g.getEdgeWeight(ForTri.get(k)) == Weights.get(i)) {  
            flag = true;  
            Res.add(ForTri.get(k));  
            ForTri.remove(k);  
        }else  
            k++;  
    }  
}
```

2^{nde} Phase : Analyse successives des arêtes et détection de cycle

```
PatonCycleBase<Node,DefaultWeightedEdge> cycleDetector = new PatonCycleBase<Node, DefaultWeightedEdge>(util);  
int j = 0;  
while(j < Res.size()) {  
    util.addEdge(g.getEdgeSource(Res.get(j)), g.getEdgeTarget(Res.get(j)),Res.get(j));  
    if(cycleDetector.getCycleBasis().getWeight() != 0) {  
        util.removeEdge(Res.get(j));  
        Res.remove(j);  
    }else  
        j++;  
}
```

Instanciation d'un détecteur de cycle *cycleDetector* (*PatonCycleBase*<V,E> géré par la librairie JGraphT) spécialisé dans les graphes non orienté prenant en paramètre *util* et retournant si présent un cycle (chaîne fermée simple) du graphe

Remarque : erreur *IllegalArgumentException* est lancé si le graphe passé en paramètre contient des arêtes multiples c'est-à-dire s'il existe plusieurs arêtes entre deux même sommets.

Ainsi avec une boucle *while* nous parcourrons l'ensemble des arêtes rangées en ordre croissant dans *Res* que l'on ajoutera progressivement au graphe non orienté *util* tout en effectuant ensuite un test à l'aide du détecteur de cycle sur le graphe courant. Donc si

un cycle est présent dans le graphe, nous supprimons l'arête dans le graphe *util* et aussi dans la liste des arêtes *Res*.

Enfin, nous avons l'ensemble des arêtes satisfaisant d'un côté le critère de somme de poids minimum et de l'autre le respect des propriétés d'un arbre (en partant de l'hypothèse que le graphe non orienté en paramètre soit connexe).

3^{ème} Phase : Affichage des arêtes

```
144 /
148     double poidsTotal = 0;
149     for(int k = 0; k < Res.size(); k++) {
150         int vdp = g.getEdgeSource(Res.get(k)).getId();
151         int vda = g.getEdgeTarget(Res.get(k)).getId();
152         if(vda > 63)
153             System.out.println("vda: "+vda);
154         if(vdp > 63)
155             System.out.println("vdp: "+vdp);
156         System.out.println("\nArête n°"+k+":\nId du sommet de Départ: "+g.getEdgeSource(Res.get(k)).getId()+"\nId du sommet d'arrivée: "+g.getE
157         System.out.println("Ville du sommet de départ: "+OutilsGraphe.getBigVilles().get(vdp-1)+"\nVille du sommet d'arrivée: "+OutilsGraphe.ge
158         poidsTotal+= g.getEdgeWeight(g.getEdge(OutilsGraphe.getBigSommetVilles().get(vdp-1), OutilsGraphe.getBigSommetVilles().get(vda-1)));
159     }
160     System.out.println("\nPoids Total: "+ poidsTotal);
161 }
162 }
163 }
164 }
```

Problems @ Javadoc Declaration Search Console

<terminated> TestMethods [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/bin/java (Apr 19, 2019, 1:34:30 AM)

Ville du sommet d'arrivée: Montpellier

Arête n°58:
Id du sommet de Départ: 3
Id du sommet d'arrivée: 7
Ville du sommet de départ: Annecy
Ville du sommet d'arrivée: Besançon

Arête n°59:
Id du sommet de Départ: 43
Id du sommet d'arrivée: 3
Ville du sommet de départ: Nîmes
Ville du sommet d'arrivée: Annecy
Poids Total: 3779.0

Le graphe affiche chaque arrête faisant partie de l'arbre de poids minimum en précisant son numéro, l'id du sommet de départ, d'arrivée ainsi que leur dénomination.

Il affiche aussi à la fin le poids total de l'arbre de poids minimum.

Intérêts :

Ses intérêts se centrent sur les propriétés du graphe qu'il construit. En effet il assure une inter-connectivité avec l'ensemble des points du réseau mais également un coût minimal en poids. Il peut par exemple servir à l'élaboration d'un nouveau réseau de ligne routier permettant une accessibilité à l'ensemble des villes doublé d'un coût minimal de mise en place.

Méthodes de JGraphT

Algorithmes : HawickJamesSimpleCycles et CycleDetector (Package : org.jgrapht.alg.cycle)

Problème : Détecteur de cycles

Nous souhaitons organiser une course de voiture sur le circuit routier. 63 villes pourraient servir d'étapes, avec 350 routes les reliant les unes aux autres. Malheureusement, c'est la période des vacances et beaucoup de gens sur les routes, rendant chacune de ces routes, empruntable seulement à sens unique si l'on veut organiser une course dessus. Nous souhaitons partir d'une ville, parcourir une distance puis revenir à notre point de départ pour terminer la course.

Pour cela allons chercher des circuits dans le graphe des villes.

Structure du graphe :

Graphe simple « grapheCycleSimple », orienté valué : grapheCycleSimple(V,E,c).

V = l'ensemble des villes (sommets/Node)

E = l'ensemble des chemins (arcs/DefaultWeightedEdge) reliant une ville à une autre

c(arc) = valuation de la distance entre les villes.

L'algorithme *HawickJamesSimpleCycles* prend en paramètre un graphe orienté G(V,E) et retourne les circuits simples du graphe G. « grapheCycleSimple » représente G.

```
Graph<Node,DefaultWeightedEdge> graphe = OutilsGraphe.getDirectedData();  
  
HawickJamesSimpleCycles<Node,DefaultWeightedEdge> grapheCycleSimple =  
    new HawickJamesSimpleCycles<Node, DefaultWeightedEdge>(OutilsGraphe.getBigGraphe());  
List<List<Node>>>listeCycleSimple = grapheCycleSimple.findSimpleCycles();  
System.out.println(listeCycleSimple);  
// on listeCycleSimple est vide donc notre graphe ne contient pas de cycle simple.  
// pour bien verifier que la liste est vide, je vais compter le nombre de cycle present dans le graphe.
```

A l'aide de la méthode « *findSimpleCycle ()* » on a cherché à récupérer sur une liste «listeCycleSimple» de liste de villes tous les circuits simples trouvés. On s'est aperçu que la liste « listeCycleSimple » est vide : il n'y a pas de circuit simple dans « grapheCycleSimple ».

```
double nombreCycleSimple = grapheCycleSimple.countSimpleCycles();  
System.out.println(nombreCycleSimple);  
/** on a bien 0 cycle simple dans notre graphe.  
 * je cherche maintenant des cycles pas simple dans mon graphe  
 */
```

Nous avons ensuite cherché à savoir s'il existait un circuit quelconque dans `grapheCycleSimple` grâce à l'algorithme ***CycleDetector***. Cet Algorithme prend en paramètre un graphe orienté $G(V,E)$ et retourne un booléen qui indique si G contient un circuit ou pas.

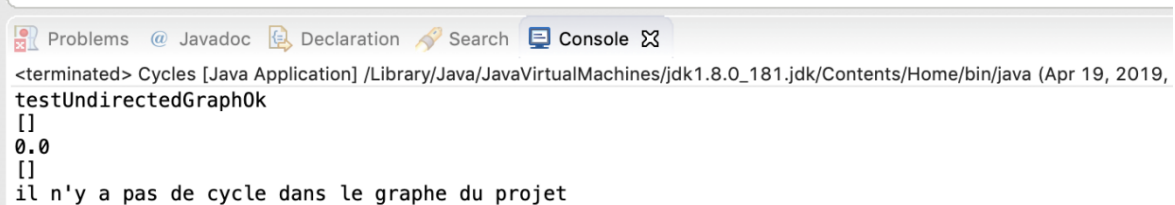
On a instancié la classe ***CycleDetector*** avec les mêmes données que celles du graphe « `grapheCycleSimple` ». À l'aide de la méthode « ***findCycle*** () » on a fait un test qui indique si le set de villes « `setCycles` » qui récupère les nœuds participants à au moins un circuit (dans le

```
CycleDetector<Node,DefaultWeightedEdge>CycleComplexe =
    new CycleDetector<Node,DefaultWeightedEdge>(OutilsGraphe.getBigGraphe());
//System.out.println(CycleComplexe); // again i know i cant just print this type like that
Set<Node>setCycle = CycleComplexe.findCycles();
System.out.println(setCycle);
// apparament pas de cycle

if(!CycleComplexe.detectCycles()){
    System.out.println("il n'y a pas de cycle dans le graphe du projet");
}
```

cas où il existe bien de circuit dans le graphe).

```
44         if(!CycleComplexe.detectCycles()){
45             System.out.println("il n'y a pas de cycle dans le graphe du projet");
46         }
47     }
48 }
49 }
```



Problems @ Javadoc Declaration Search Console

<terminated> Cycles [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/bin/java (Apr 19, 2019, testUndirectedGraph0k
[]
0.0
[]
il n'y a pas de cycle dans le graphe du projet

Conclusion du problème :

Le test a donné comme résultat une non-existence de circuit dans notre graphe de 63 villes avec 350 chemins (le calcul a pris du temps). Il n'est donc pas possible dans ce contexte (graphe orienté) de partir d'une ville (sommet) et d'y retourner en passant par les autres villes et routes.

Algorithme: BoruvkaMinimumSpanningTree
(Package: org.jgrapht.alg.spanning)

Problème : recherche d'un arbre de poids minimum.

Comme on l'a vu précédemment avec les algorithmes de cycles, notre graphe de 63 villes et 350 chemins ne présente pas de circuit. On a alors un graphe connexe sans cycle. On voudrait réaliser des travaux de rénovations pour fluidifier la circulation entre ces 63 villes. On décide de rénover que les axes routiers qui permettent d'atteindre ces villes avec une distance minimum possible. Contrainte supplémentaire par rapport à l'algorithme de Kruskal utilisé plus tôt, dans cet exemple est utilisé un graphe orienté.

On se ramène alors à la recherche d'un arbre de poids minimum.

Structure du graphe :

Graphe simple « graphePoidsMinimum », orienté valué : graphePoidsMinimum(V,E,c).

V = l'ensemble des villes (= sommets / Node)

E = l'ensemble des chemins (= arcs / DefaultWeightedEdge) reliant une ville à une autre.

c(arc) = Distance entre les villes.

L'algorithme ***BoruvkaMinimumSpanningTree*** prend en paramètre un graphe G(V,E) et retourne le poids de l'arbre de poids minimum du graphe G et les arêtes qui constituent cet arbre de poids minimum.

Dans notre cas « grapheCycleSimple » représente G.

```
27 public static void main(String[] args) {
28
29     ArrayList<String> sources = Scraper.getSources();
30     ArrayList<String> data = Scraper.getRoads2(sources);
31     ArrayList<String> villes = Scraper.setVilles(data);
32     ArrayList<Node> sommetVilles = Scraper.setCityObject(villes);
33     Scraper.addToGraph(graphe, sommetVilles);
34     ArrayList<DefaultWeightedEdge> distances = Scraper.setDistances(graphe, data, sommetVilles);
35
36     // on cherche ici l'arbre de poids (distance) minimum dans la graph du noeud.
37     BoruvkaMinimumSpanningTree<Node, DefaultWeightedEdge> graphPoidMinimum = new BoruvkaMinimumSpanningTree<Node, DefaultWeightedEdge>(graphe);
38     SpanningTreeAlgorithm.SpanningTree<DefaultWeightedEdge> arbrePoidsMin = graphPoidMinimum.getSpanningTree();
39     System.out.println(arbrePoidsMin);
40
41 }
42
43 }
```

Console X

```
<terminated> Spanning Tree (1) [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/bin/java (17 avr. 2019 23:37:21)
Spanning-Tree [weight=7558.594, edges=([
Ville: Marseille
Id du Noeud: 104
:
Ville: Toulon
Id du Noeud: 130
), (
Ville: Nimes
Id du Noeud: 116
:
Ville: Mauguio
Id du Noeud: 105
), (
Ville: Ajaccio
Id du Noeud: 74
:
Ville: Besançon
Id du Noeud: 80
), (
Ville: Ajaccio
```

Résultat au problème :

Grâce à la méthode « *getSpanningTree()* » cet algorithme on a pu récupérer un arbre de poids minimum de 7558, 594 km de distance et la liste des axes (chemins) à rénover

Algorithmes : TwoApproxMetricTSP (Package : org.jgrapht.alg.tour)

Problème : Algorithme du voyageur de commerce

Une famille souhaite faire un tour de France pendant les vacances. Elle veut passer par une liste de villes qui l'intéresse, sans contrainte de temps, mais en voulant faire le moins de trajet possible puis retourner à son point de départ.

On doit trouver le plus court cycle Hamiltonien, donc résoudre le problème du voyageur de commerce.

Structure du graphe :

Les algorithmes du problème du voyageur de commerce ne fonctionnent que sur des graphes non orientés complets.

Pour des raisons de complexité, nous avons limité la liste à 9 villes présentes dans notre grand jeu de données pour faire un petit jeu de données.

On a alors généré à partir de notre graphe départ, un sous-graphe complet non orienté de 9 villes : Ajaccio, Annecy, Besançon, Marseille, Nîmes, Paris, Perpignan, Toulouse et Toulon.

```
14 public class SalesMan {  
15     static Graph<Node,DefaultWeightedEdge> graphe = OutilsGraphe.getSmallDirectedData();  
16     static DefaultUndirectedWeightedGraph<Node, DefaultWeightedEdge> grapheNonOriente = OutilsGraphe.createUndirectedGraph(graphe);  
17 }
```

Graphe simple «grapheNonOriente», non-orienté valué : grapheNonOriente(V,E,c).

V = l'ensemble des villes (= sommets / Node)

E = l'ensemble des chemins (= arêtes / DefaultWeightedEdge) reliant une ville à une autre.

c(arêtes) = Distance entre les villes.

L'algorithme *TwoApproxMetricTSP* ne prend rien paramètre. On instancie d'abord un graphe

```
public static void main(String[] args) {  
    TwoApproxMetricTSP<Node,DefaultWeightedEdge>grapheSMA = new TwoApproxMetricTSP<Node,DefaultWeightedEdge>();  
    GraphPath<Node, DefaultWeightedEdge>test = grapheSMA.getTour(grapheNonOriente);  
    System.out.println(test);  
}
```

de type *TwoApproxMetricTSP*.

La méthode « *getWeight()* » a permis d'avoir la somme de la distance que fait le plus court cycle Hamiltonien de notre graphe, elle est de 2294.297 km.

```
24     double distanceAParcourir = test.getWeight();  
25     System.out.println("la distance totale a parcourir est: "+ distanceAParcourir+" km");  
26 }  
27
```

Problems @ Javadoc Declaration Search Console

<terminated> SalesMan [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/bin/java (Apr 19, 2019, 1:5
testUndirectedGraph0k
testUndirectedGraph0k
la distance totale a parcourir est: 2294.0 km

Au moyen de la méthode « *getTour()* » qui elle prend en paramètre un graphe non-orienté complet et retourne l'itinéraire du plus court cycle Hamiltonien.

```
12 import fr.univ.graphproject.utiltygraphe.UtiltyGraphe;
13
14 public class SalesManAlgo {
15     static Graph<Node,DefaultWeightedEdge> graphe = OutilsGraphe.getSmallDirectedData();
16     static DefaultUndirectedWeightedGraph<Node, DefaultWeightedEdge> grapheNonOriente = OutilsGraphe.createUndirectedGraph(Graphe);
17
18     public static void main(String[] args) {
19         TwoApproxMetricTSP<Node,DefaultWeightedEdge>grapheSMA = new TwoApproxMetricTSP<Node,DefaultWeightedEdge>();
20         GraphPath<Node, DefaultWeightedEdge>test = grapheSMA.getTour(grapheNonOriente);
21         System.out.println(test);
22         //Set<DefaultWeightedEdge> cheminsGraphe = grapheNonOriente.getAllEdges(Node sourceVertex,Node targetVertex);
23         //System.out.println(grapheNonOriente);
24         double distanceAParcourir = test.getWeight();
25         System.out.println("la distance totale a parcourir est: "+ distanceAParcourir+" km");
26     }
27 }
28
29
```

Problems Javadoc Declaration Console x

<terminated> SalesManAlgo [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/bin/java (18 avr. 2019 00:34:12)

Ville: Perpignan
Id du Noeud: 71
,
Ville: Toulouse
Id du Noeud: 73
,
Ville: Marseille
Id du Noeud: 68
,
Ville: Toulon
Id du Noeud: 72
,
Ville: Paris
Id du Noeud: 70
,
Ville: Ajaccio
Id du Noeud: 65
]
la distance totale a parcourir est: 2294.297 km

Conclusion du problème :

Pour passer toutes les villes qui l'intéresse, tout en faisant le minimum de trajet, et en retournant à son point de départ, cette famille devra parcourir les villes suivantes dans cet ordre :

Ajaccio ➡ Besançon ➡ Annecy ➡ Nîmes ➡ Perpignan ➡ Toulouse ➡ Marseille
Toulon ➡ Paris ➡ Ajaccio.

Et elle parcourra au total 2294km.

Annexe

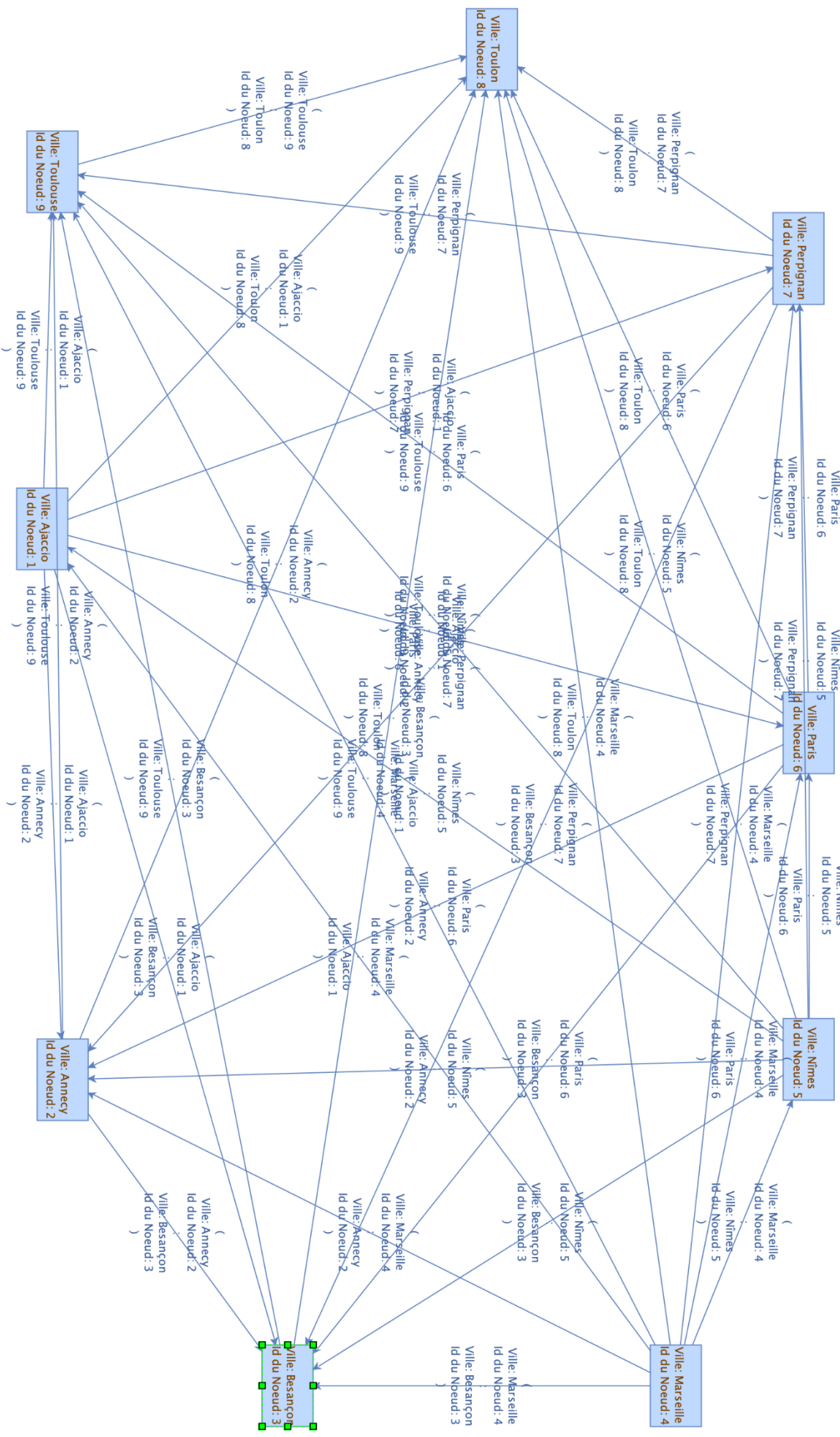
Affichage du graphe

Modélisé à l'aide de la classe TestVisu observée en classe.

Le graphe avec 350 arêtes n'était pas assez clair alors nous montrons ici le graphe réduit avec 9 villes et 36 arêtes.

Ce graphe étant encore brouillon, nous avons une version réorganisée, plus lisible à la page suivante.





Récupération des données

Nous avons récupéré les données sur le site bonnesroutes.com :

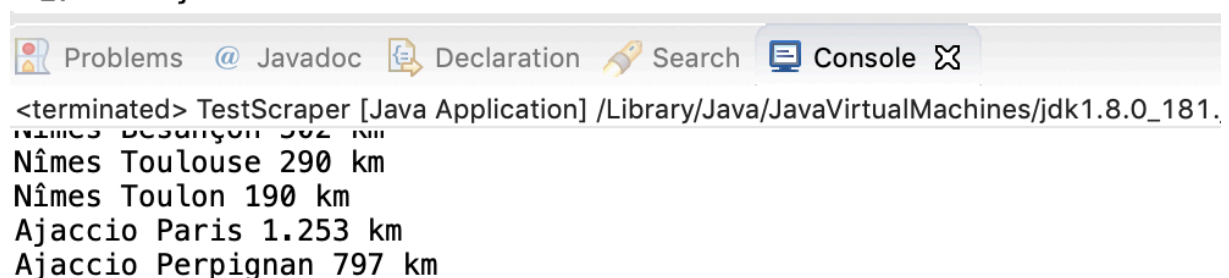
De	À	Distance	De	À	Distance
Marseille	Nîmes	122 km	Ajaccio	Besançon	1 044 km
Marseille	Ajaccio	521 km	Ajaccio	Toulouse	883 km
Marseille	Paris	773 km	Ajaccio	Toulon	471 km
Marseille	Perpignan	317 km	Paris	Perpignan	846 km

Nous avons codé une classe Scraper utilisant la librairie Jsoup, nous permettant de récupérer le code source de la page en ciblant certaines parties :

```
public static ArrayList<String> getRoads2(ArrayList<String> urls){
    ArrayList<String> txt = new ArrayList<String>();
    for(int i = 0; i < urls.size(); i++) {
        try {
            Document document = Jsoup.connect(urls.get(i)).get();
            for(Element row : document.select("div.t_row")) {
                String element = row.select("div.t_row").text();
                txt.add(element);
            }
        } catch (Exception exception) {
            exception.printStackTrace();
        }
    }
    purge1(txt);
    return txt;
}
```

La fonction `getRoads2()` nous retourne à partir de l'URL de la page une `ArrayList<String>` contenant dans cet ordre : une ville de départ, une ville d'arrivée, et la distance entre les deux. A partir de cela nous pouvons instancier des sommets, ainsi que des arêtes vallées de la distance entre deux villes.

```
22 public static void showRawData(ArrayList<String> data) {
23     System.out.println("RawData: "+data.toString()+"\n");
24     for(int i = 0; i < data.size(); i++) {
25         System.out.println(data.get(i));
26     }
27 }
```



```
<terminated> TestScraper [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_181.
Nîmes Besançon 502 km
Nîmes Toulouse 290 km
Nîmes Toulon 190 km
Ajaccio Paris 1.253 km
Ajaccio Perpignan 797 km
```

Traitement des données

```
ArrayList<String> sources = Scraper.getSources();
ArrayList<String> data = Scraper.getRoads2(sources);
ArrayList<String> villes = Scraper.setVilles(data);
ArrayList<Node> sommetVilles = Scraper.setCityObject(villes);
Scraper.addToGraph(graphe, sommetVilles);
ArrayList<DefaultWeightedEdge> distances = Scraper.setDistances(graphe, data, sommetVilles);
Scraper.setDistances(graphe, data, sommetVilles);
```

Une fois que la fonction `getRoads2()` a retourné une `ArrayList<String>` contenant les informations, la méthode `setVilles()` identifiera les villes et les mettra dans une `ArrayList<String>` dédiée. Une fois ajoutées, elles triées par ordre alphabétique puis une fonction `purge3()` enlève tous les doublons et retourne la liste.

La fonction `setCityObj(villes)` instancie pour chaque ville dans la liste un `Node` auquel est attribué un id et le nom de la ville :

```
36 public static void showSommetVilles(ArrayList<Node> sommetVilles) {
37     System.out.println("Affichage des Villes-Sommets:");
38     for(int i = 0; i < sommetVilles.size(); i++) {
39         System.out.println("Sommets n°"+sommetVilles.get(i).getId()+"\nVille: "+sommetVilles.get(i).getVille()+"\n");
40     }
41 }
42
```



Les sommets sont ajoutés au graphe, et enfin sont instanciées les arêtes avec la fonction `setDistances()`. Pour chaque ligne dans `RawData` elle va rechercher dans la liste des sommets le nom de la 1^{ère} ville, la rechercher dans la liste des sommets et la désigner comme point de départ, la 2^{ème} comme ville d'arrivée, et donner comme valuation de l'arête la distance entre les deux :

```
43 public static void showDistances(ArrayList<DefaultWeightedEdge> distances) {
44     System.out.println("Affichage des Distances-Arrêtes:");
45     for(int i = 0; i < distances.size(); i++) {
46         System.out.println("Arête n°"+i+"\n"+distances.get(i).toString()+"\n");
47     }
48 }
```

