CR Projet Graphes

Polytech lyon

Recherche de plus court chemin

Kpamy mark – OGLI FABIEN – TAGGUEjou CHristian

2017

Table des matières

I- Structure du Graphe :

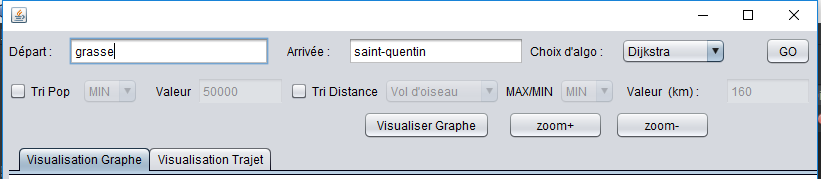
1- Liste de Communes

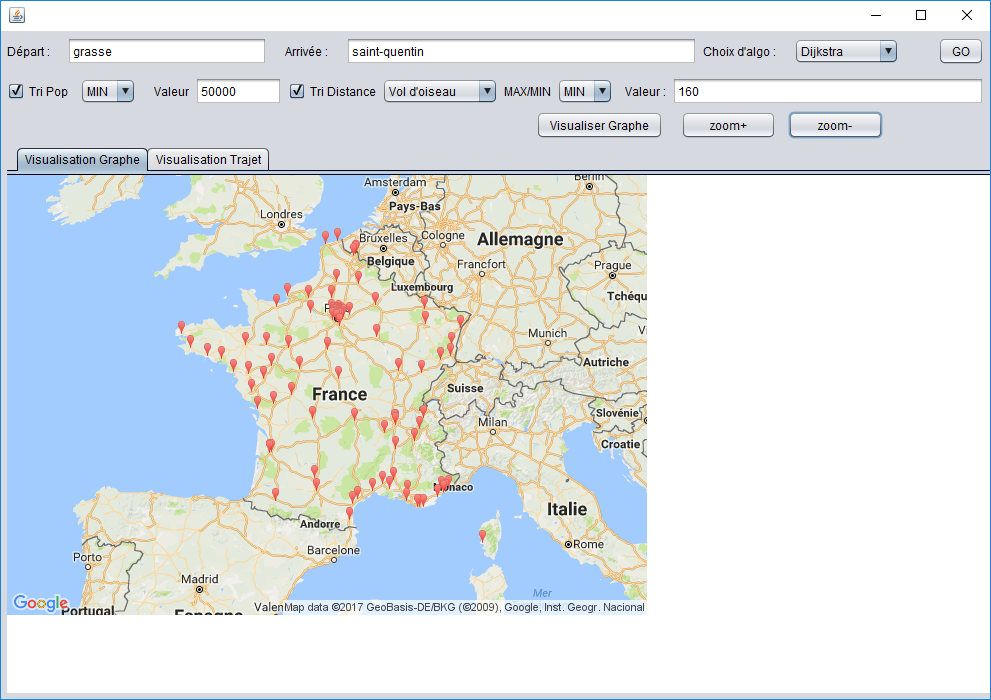
La première étape du projet consistait à ouvrir le fichier contenant les communes. Pour cela, nous avons commencé par créer une classe Communes contenant les mêmes paramètres que les communes présentent dans le fichier CommunesFrance.csv. Ensuite, nous avons importé ce fichier dans notre projet, sous forme de liste de Communes.

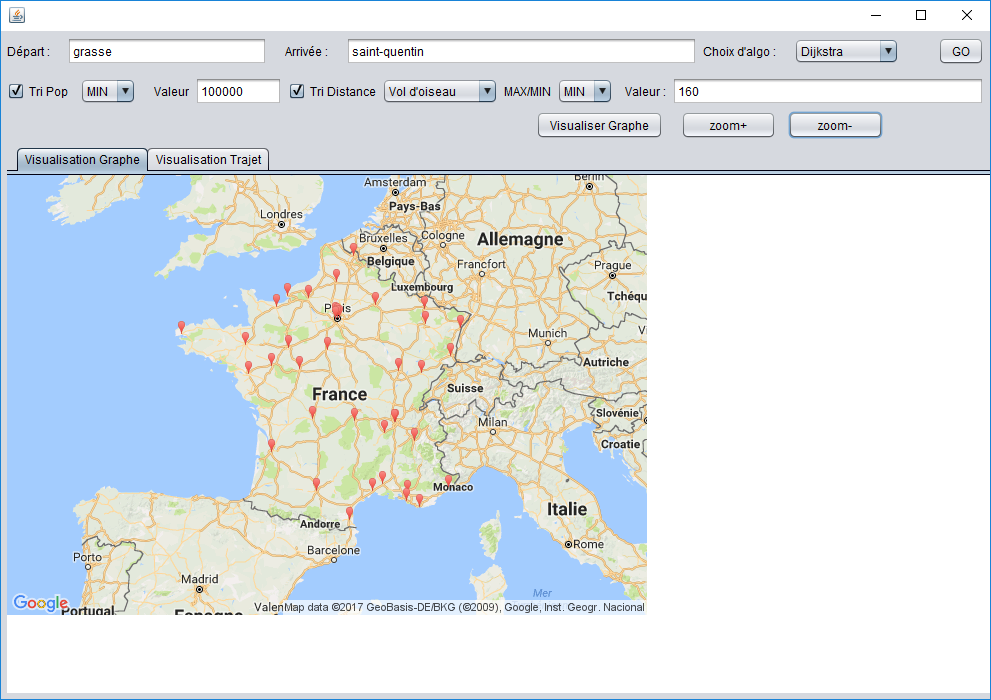
2- Structure

Avant d’implémenter la structure du graphe, nous avons créé la classe Sommet, qui représentera les différents sommets du graphe. Cette classe prend en paramètre une commune, une liste de successeur et un coût.

Pour avoir la structure d’un graphe, il suffit d’instancier la classe Graphe. Lors de l’instanciation, l’utilisateur à la possibilité de choisir quel type de graphe il veut construire. Par exemple, il pourra effectuer un tri sur la population (Supérieur ou inférieur à un certain nombre d’habitants) ou créer un graphe dont les sommets sont à une certaine distance les uns des autres.







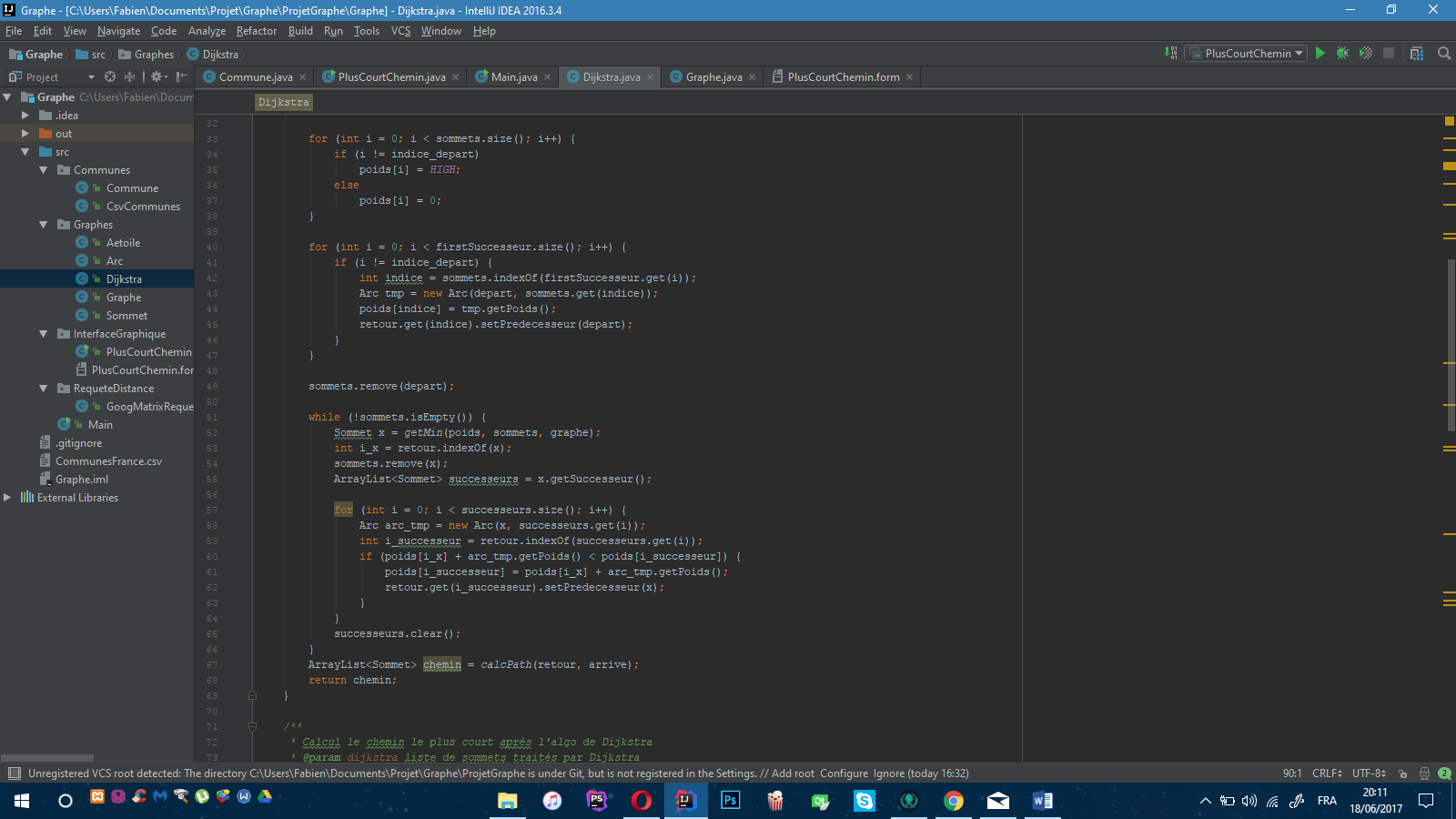
A gauche, nous avons un graphe contenant les communes de France ayant une population supérieure à 50 000 habitants, et à droite, un graphe avec les communes ayant plus de 100 000 habitants.

Les méthodes qui permettent d’effectuer le tri des communes sont simples. Elles parcourent la liste des communes, et selon leurs caractéristiques, elles sont ajoutées ou pas à la liste de communes triés qui sera affichée.

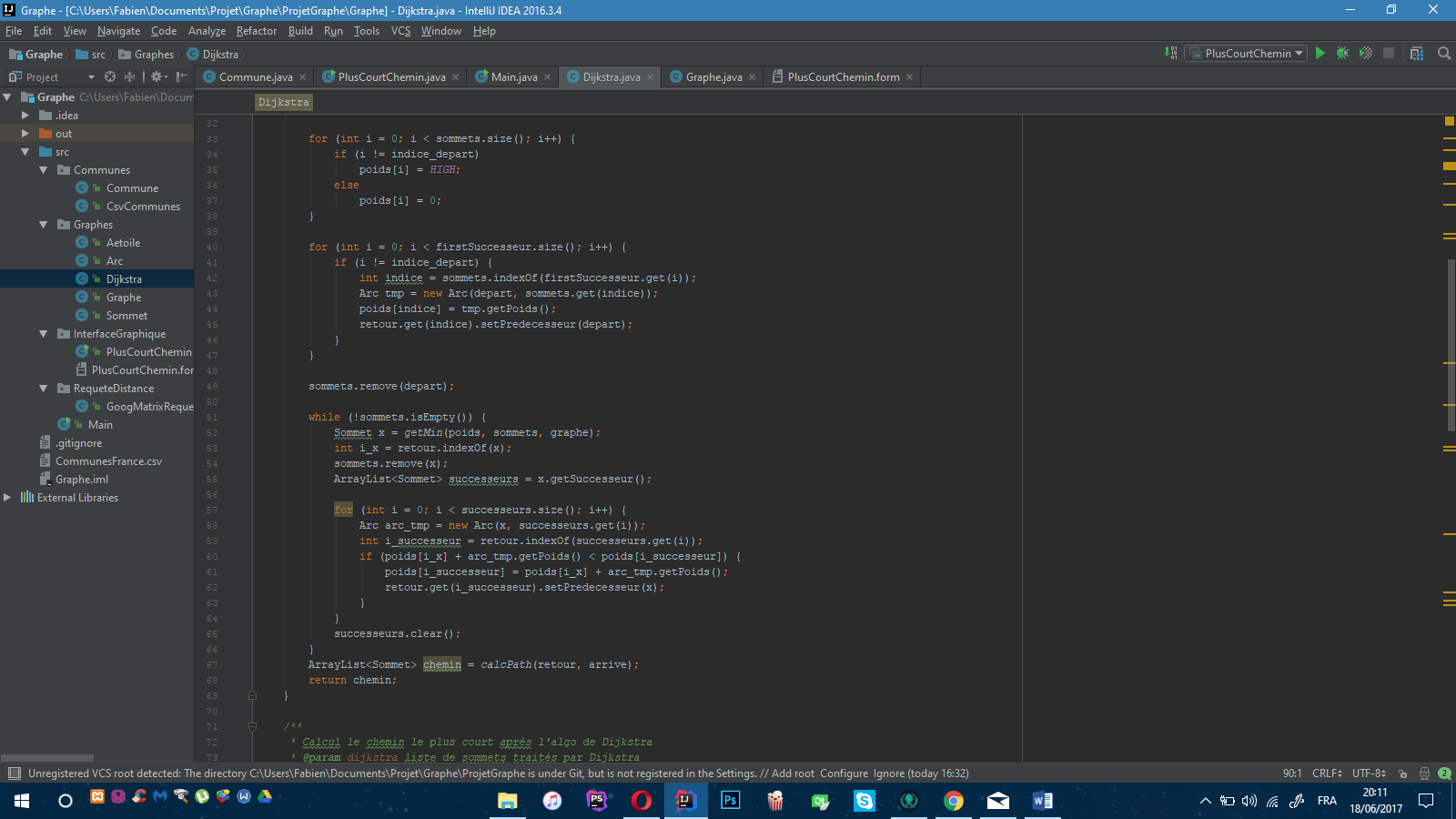
La méthode trieVolDoiseau() compare toutes les villes entre elles afin de vérifier si elles sont à la bonne distance. Elle a donc une complexité en O(n²). Sachant qu’il s’agit d’une option, elle peut être désactivé par l’utilisateur s’il le souhaite. On peut l’améliorer un peu faiblement de manière à avoir n(n-1) mais il n’apporte pas une modification remarquable au temps d’exécution.

II - Dijsktra

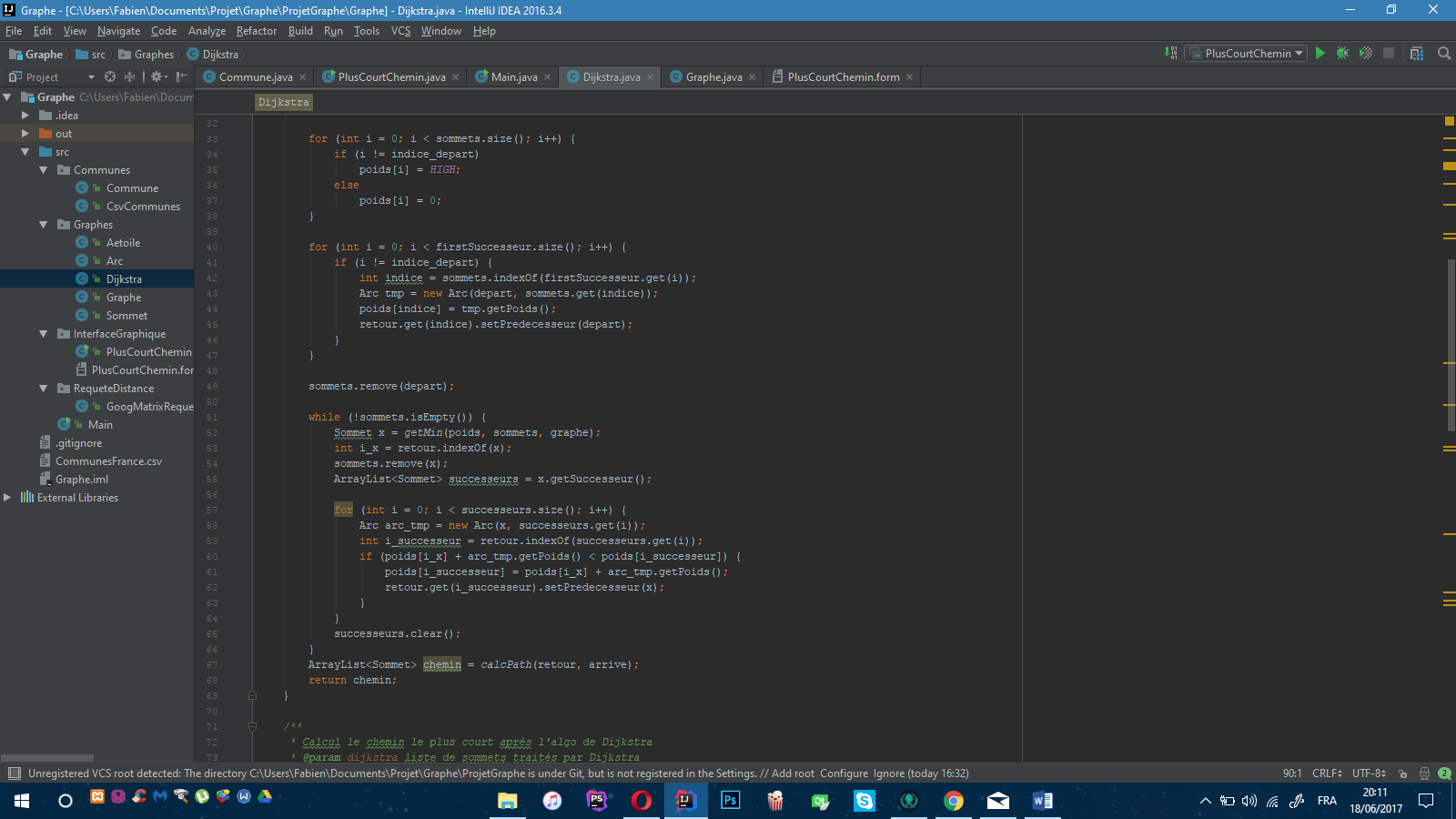
Nous avons implémenté l’algorithme de Dijsktra présent dans le cours. Voici le code de la méthode Dijsktra qui prend le graphe, le sommet d’arrivée et le sommet de départ/



Tout d’abord, nous initialisons le tableau de valuation des sommets à une grande valeur(“HIGH”) sauf pour le sommet de départ que l’on initialise à 0.

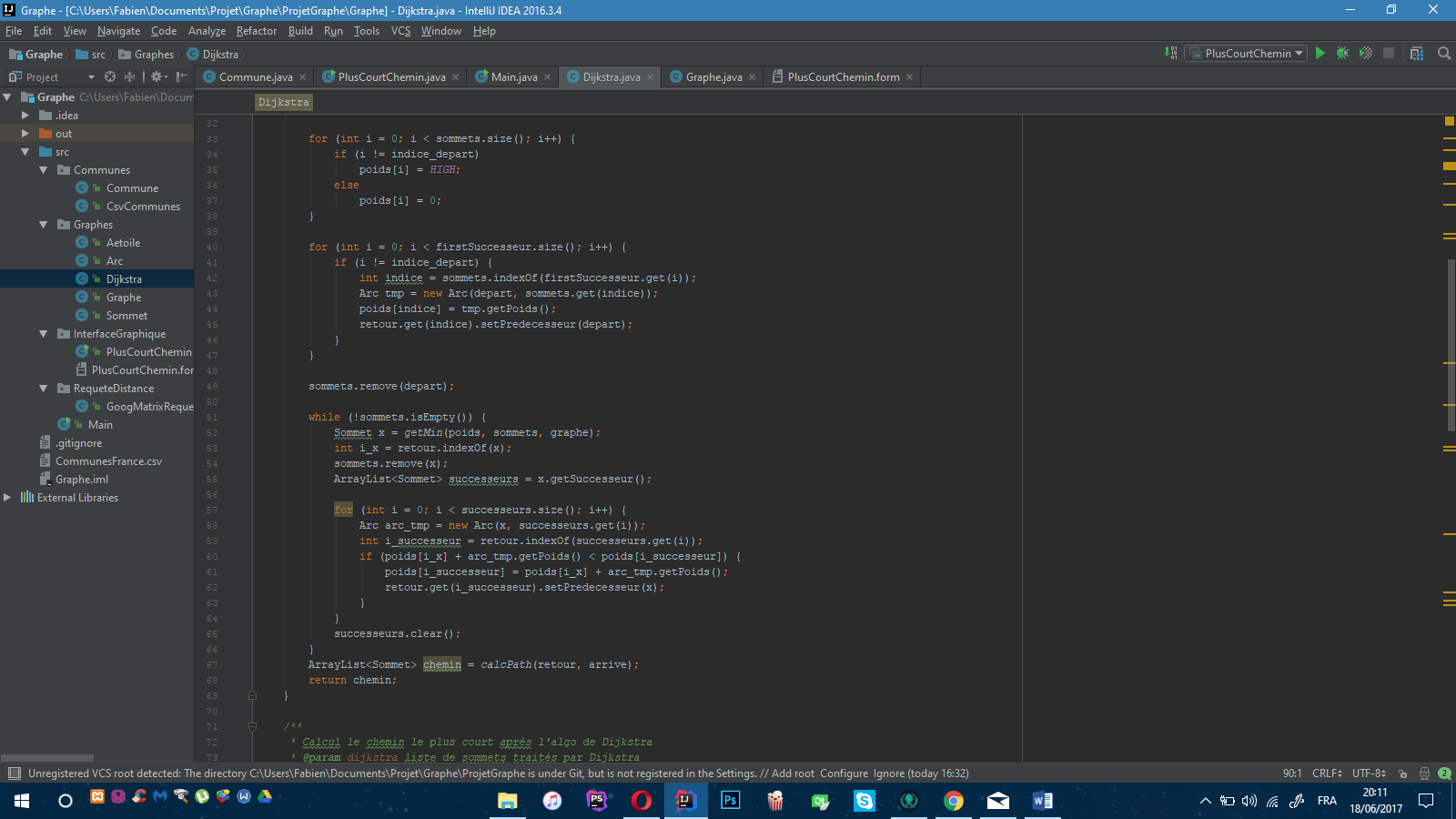


On parcourt ensuite tous les successeurs du sommet de départ et attribuons une valeur à tous ces sommets qui correspond à la distance entre le point de départ et ses successeurs. On définit ensuite le prédécesseur des successeurs qui est le sommet d’arrivée. Le prédécesseur nous servira à retracer le chemin entre le départ et l’arrivée. Nous enlevons ensuite le sommet de départ.



Nous rentrons à présent dans une boucle “Tant que” qui s’arrête lorsque la liste “sommets” est vide. Elle attribue au sommet x le sommet ayant la plus petite valuation dans la liste des sommets “sommets” soit la liste des sommets qu’ils nous restent à traiter.

Nous enlevons le sommet x de “sommets”.



Nous parcourons ensuite les successeurs du sommet x pour déterminer qui prend la valeur minimale entre sa valeur dans le tableau de valuation ou la somme de celle-ci et de la distance avec son successeur.

Si la somme est plus petite, on définit le prédécesseur du successeur.

III – A étoile

L’algorithme A\* permet la recherche du chemin de coût minimal dans un graphe orienté.

A chaque étape d’itération de l’algorithme, on choisit le meilleur nœud en évaluant le coût du nœud actuel vers la source puis le coût du nœud actuel vers l’arrivée. Le coût du nœud actuel est vers la source est connu : c’est la somme du coût du nœud précédent + le coût du nœud précédent vers la source. Pour le coût vers la destination, on utilise une heuristique donc une estimation en passant soit par exemple par la distance à vol d’oiseau, la distance de Manhattan ou encore mieux la distance réelle par requêtes vers les API de géolocalisation.

Voici le pseudo-code de l’A \* (tiré de wikipédia):

Fonction cheminPlusCourt(g:Graphe, objectif:Nœud, depart:Nœud)

closedList = File()

openList = FilePrioritaire(comparateur=compare2Noeuds)

openList.ajouter(depart)

tant que openList n'est pas vide

u = openList.depiler()

si u.x == objectif.x et u.y == objectif.y

reconstituerChemin(u)

terminer le programme

pour chaque voisin v de u dans g

si v existe dans closedList avec un cout inférieur ou si v existe dans openList avec un cout inférieur

neRienFaire()

sinon

v.cout = u.cout +1

v.heuristique = v.cout + distance([v.x, v.y], [objectif.x, objectif.y])

openList.ajouter(v)

closedList.ajouter(u)

terminer le programme (avec erreur)

Cet algorithme est fonctionnel et donne de bonnes performances mais toutefois il est possible de faire mieux en implémentant la classe PriorityQueue. Une PriorityQueue est une file qui range des objets à traiter selon un ordre précis d’empilage et de dépilage. La méthode poll() permet de retirer le premier élément de la file. Il faut dans le cas le nôtre structure faire en sorte que le premier élément retirer soit l’élément de coût minimum. Pour cela, on implémente l’interface Comparable sur la classe Sommet :

public class Sommet implements Comparable<Sommet>

Ensuite il faut redéfinir la méthode compareTo():

@Override

public int compareTo(Sommet t) {

Double tmp = this.coutTotal();

return tmp.compareTo(t.coutTotal());

}

La PriorityQueue garantie que l’élément de coût minimal (dans notre cas) ou maximal soit toujours le premier à sortir de la file.

Voici notre implémentation :

public static LinkedList<Sommet> algo(Sommet depart, Sommet arrive, int type) {

PriorityQueue<Sommet> openList = new PriorityQueue<>();

LinkedList<Sommet> closedList = new LinkedList<>();

openList.add(depart);

Sommet courant;

while (!openList.isEmpty() && !openList.contains(arrive)) {

//on retire le 1er element

courant = openList.poll();

//on l'ajoute a liste des sommets deja visités

closedList.add(courant);

//parcours de la liste de successeurs

for (Sommet tmp : courant.getSuccesseur()) {

if (!closedList.contains(tmp)) {

System.out.println("Sommet : " + tmp.getCommune().getNom());

//si ce successeur est deja dans l’openList

if (openList.contains(tmp) && tmp.getgCost() > ((type == 0) ? (tmp.calculateGcost(courant)) : (tmp.calculateGcost1(courant)))) {

openList.remove(tmp);

//on calcule son cout vers la source

tmp.setPredecesseur(courant);

//on set comme predecesseur de chaque successeur le sommet courant

: " + tmp.getPredecesseur().getCommune().getNom());

tmp.sethCost(arrive, type);

//on calcule le cout vers la destination

tmp.setgCost(courant, type);

openList.add(tmp);

} else if (!openList.contains(tmp)) {

tmp.setPredecesseur(courant);

//on set comme predecesseur de chaque successeur le sommet courant

System.out.println("Predecesseur : " + tmp.getPredecesseur().getCommune().getNom());

//on calcule son cout vers la source

tmp.sethCost(arrive, type);

//on calcule le cout vers la destination

tmp.setgCost(courant, type);

openList.add(tmp);

}

} else {

if (openList.contains(tmp) && tmp.getgCost() > ((type == 0) ? (tmp.calculateGcost(courant)) : (tmp.calculateGcost1(courant)))) {

openList.remove(tmp);

System.out.println("Sommet : " + tmp.getCommune().getNom());

//on set comme predecesseur de chaque successeur le sommet courant

tmp.setPredecesseur(courant);

//on met a jour les nouveaux couts

tmp.sethCost(arrive, type);

tmp.setgCost(courant, type);

openList.add(tmp);

}

}

}

}

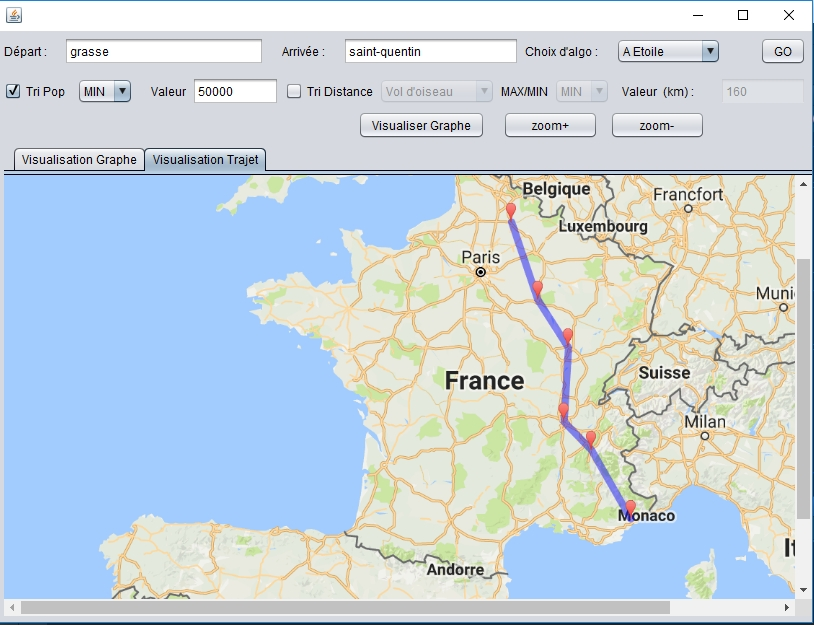
arrive.setPredecesseur(closedList.getLast());

return calcPath(depart, arrive);

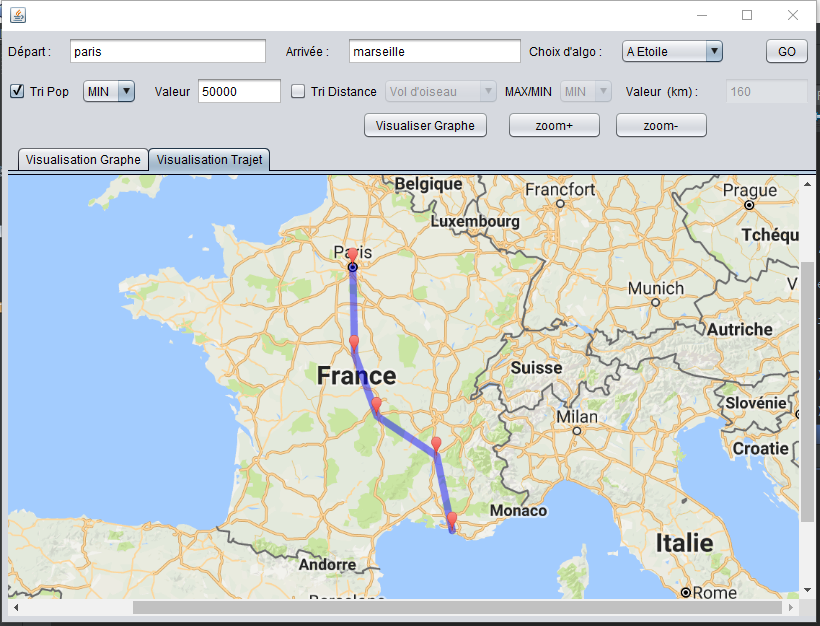
}

Test effectué :

Itinéraire de Grasse à Saint-Quentin:



Itinéraire de Paris à Marseille:



IV-Graphes dynamiques

Dans un environnement où il existe de nombreux facteurs aléatoires, il est intéressant de travailler des graphes dynamiques dont la valuation des arcs dépend des conditions en temps réel. Cette notion de graphe dynamique est très utilisée notamment pour les applications en temps réel et l’amélioration d’un itinéraire routier. Il existe de nombreuses thèses et des études qui ont été faite sur le sujet. Nous discuterons ici de quelques aspects et des algorithmes qui traitent ce genre de graphe.

Définition :

Soit G = (N,A) un graphe orienté et pondéré. G est dit graphe dynamique déterministe si le poids d'un arc est une valeur déterministe mais qui dépend du temps. Par conséquent, le poids d'un arc (i,j) ∈ A est décrit par la fonction p(i,j,t) avec t la date de départ du noeud i.

Dans cette structure de graphe, le temps est supposé soit continu ou discret mais il est courant d’utiliser le temps en discret.

Il existe aussi des sous-familles de graphes dynamiques :

Graphes FIFO :

Les graphes FIFO sont une sous-classe des graphes dynamiques déterministes qui contient des arcs FIFO. Un arc (i,j) ∈ A est appelé arc FIFO si sa fonction de poids est définie de telle manière que partir du noeud i pour le noeud j à t0 ≥ t implique nécessairement d'arriver à j plus tard que t + p(i,j,t).

IV-1 Plus court chemin avec graphes dynamiques

Le problème de calcul de plus court chemin dans un graphe dynamique a été posé la première fois par les chercheurs L. Cooke et E. Halsey. Cependant, ce n’est que le cas avec le temps discret qui a été plus traité.

Les principales problématiques qui ressortent de cette étude sont :

* Le calcul du plus court chemin d'un noeud source vers tous les autres noeuds du graphe pour une date de départ du noeud source fixée
* Le calcul du plus court chemin d'un noeud source vers tous les autres noeuds du graphe pour toutes les dates de départ possibles du noeud source

La première problématique a été jugé et prouvé comme NP-difficile. La version avec les graphes FIFO quant à elle est polynomiale.

Pour ce dernier, l’algorithme de Dijkstra a été adaptée pour trouver le plus court chemin et il a été montré que le résultat qu’il retourne est optimal

Voici l’algorithme de Dijkstra en version dynamique :

On note :

Soit CD la fonction de coût d'un chemin dans G. CDt0((u0,u1,...,um)) indique le coût du chemin (u0,u1,...,um) en partant du noeud u0 à t0.

Soit G = (N,A) un graphe dynamique déterministe. Le plus court chemin de i ∈ N à j ∈ N en partant de i à l'instant t0 est noté PCCt0(i,j). PCCt0(i,j) = Ch(i,j) avec CDt0(Ch(i,j)) = minP∈ECh(i,j){CDt0(P)}. Au cas où plusieurs chemins avec un coût minimal existent, un parmi eux est choisi aléatoirement.

Debut Algo:

//Initialiser les plus court chemins

PCCt0(Ssource,Ssource) = (Ssource)

∀i ∈ N \ i différent Ssource, PCCt0(Ssource,i) = ∅

//Initialiser l'ensemble Q

Q = {Ssource}

//Initialiser l'ensemble F

F = ∅

Tant que Q différent ∅ faire

Choisir i de Q tel que CDt0(PCCt0(Ssource,i)) soit minimal

Q = Q\{i} et F = F ∪{i}

Développer les successeurs j de i

Pour chaque j faire

Si CDt0(PCCt0(Ssource,i)) + p(i,j,CDt0(PCCt0(Ssource,i))) < CDt0(PCCt0(Ssource,j))

Alors

PCCt0(Ssource,j) = PCCt0(Ssource,i)∪{j}

Q = Q∪{j}

Fin Si

Fin Pour

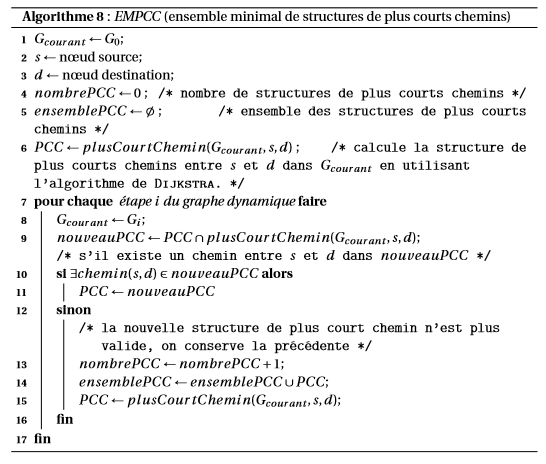
Fin Tant que

Pour les autres graphes, il existe un algorithme appelé ChronoSPT valide que lorsque le temps est discret.

Par ailleurs il existe une algorithme appelée EMPCC(ensemble de plus court chemin) qui permet de construire des structures plus courts chemins au cours de l’évolution d’un graphe dynamique.

Voici son fonctionnement général :(Extrait de thèse)

A partir de la première étape i =0, le sous-graphe de plus courts chemins est calculé et vaut comme structure de référence. De manière itérative, les événements des étapes suivantes sont appliqués et à chaque étape t une nouvelle structure de plus courts chemins est calculée. L’intersection entre la structure de plus courts chemins de référence et la nouvelle est calculée de manière à savoir si le plus court chemin qui vaut pour le nouveau graphe existe dans cette intersection. S’il existe bien un tel chemin, alors cette nouvelle structure devient la structure de référence. Le nombre de plus courts chemins n’augmente pas car la nouvelle structure de référence est incluse dans la précédente. Si au contraire, l’intersection entre la nouvelle structure de plus court chemin et la structure de référence ne permet pas de relier source et destination, cela signiﬁe que la structure de référence n’est plus valide pour l’étape t. On sait en revanche que cette structure est valable jusqu’à l’étape t−1. Dans ce cas, la structure de référence invalide est mémorisée, la nouvelle structure de référence correspond alors à la structure de plus courts chemins de l’étape t et le nombre total de structures de plus courts chemins est incrémenté.



Conclusion

Sources

Wikipedia

Mohamed Mejdi Hizem. Recherche de chemins dans un graphe à pondération dynamique : application à l’optimisation d’itinéraires dans les réseaux routiers. Automatique / Robotique. Ecole Centrale de Lille, 2008. Français. <tel-00344958>

Yoann Pigné. Modélisation et traitement décentralisé des graphes dynamiques Application aux réseaux mobiles ad hoc. R´eseaux et t´el´ecommunications [cs.NI]. Université du Havre, 2008. Français. <tel-00371962>