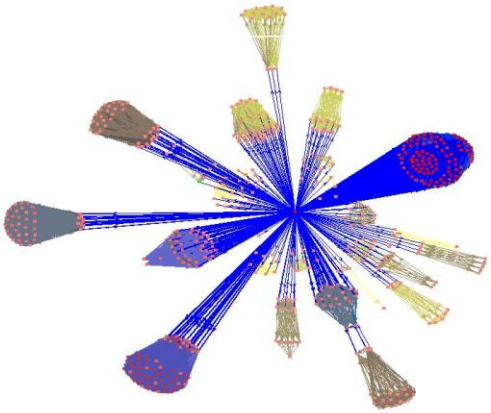
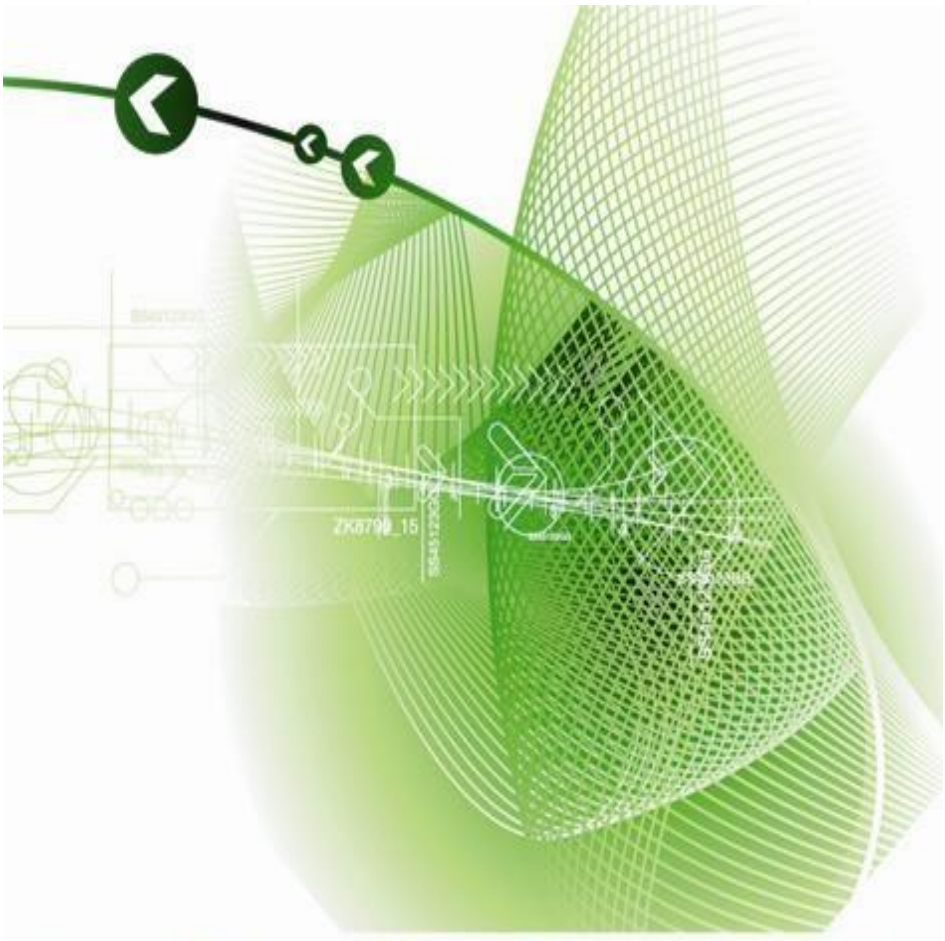
GAYRAUD Grégory
KLEIN Eléonore
MARTIN Mathieu
MIETLICKI Pascal

GRAPHES - RAPPORT





Groupe A

Calcul d'itinéraires sans et avec contraintes pour la société MonTrajetElectrique.com

Graphes - Rapport

1. INTRODUCTION

Chaque litre de gazole consomme l'oxygène de l'air et charge, inexorablement, l'atmosphère en gaz carbonique, oxyde d'azote, particules toxiques… qui accroissent sans cesse l'effet de serre. C'est pourquoi il a fallu, afin de préserver notre belle planète bleue, se tourner vers des véhicules moins polluants tels que les véhicules électriques. Un bon nombre de stations ont donc du répondre à cette nouvelle demande, notamment en permettant la recharge ou l'échange de batteries pour les usagers car, malheureusement, les batteries ont une autonomie limitée et ne permettent que de courts trajets inférieur à 150km.

Ce marché émergent et en nette croissance a permis à des sociétés tel que MonTrajetElectrique.com de proposer des solutions adaptées aux utilisateurs de véhicule électriques afin de répondre à leurs attentes en leur proposant des trajets adaptées à leurs contraintes et, notamment, leur permettant de ne plus jamais tomber en panne de batteries.

Ce projet a pu être mené à bien par une équipe de 4 jeunes étudiants ingénieurs qui ont su répondre au cahier des charges en implémentant des algorithmes optimisés. Algorithmes permettant de calculer les itinéraires les plus courts sans ou avec contraintes telle que la gestion des batteries du véhicule afin d'indiquer une station de ravitaillement. Ils ont aussi pu implémenter un programme pour la représentation graphique des cartes s'appuyant sur les utilitaires graphviz et neato. Ceci pour le plus grand bonheur des clients de MonTrajetElectrique.com !

Ce rapport décrit le fruit de leur travail avec les différentes difficultés auxquelles ils ont pu être confrontés ainsi que la manière dont elles ont été résolues.

2. REPRESENTATION DU GRAPHE

a. Structure de données

Tout d'abord, afin d'implémenter le graphe, nous avons évidemment suivi le cahier des charges ainsi que les indications qui nous ont été fournies. C'est pourquoi nous avons d'abord utilisé un vecteur pour stocker les valeurs des différents fichiers représentant le graphe. Malheureusement, pendant le déroulement du projet, nous nous sommes rendu compte que c'était une erreur car cela induisait une grande lenteur dans l'exécution du programme.

Problème des vecteurs

La classe Vector est d'un emploi pénible (initialisation nécessaire des cases vides, confusion entre taille et capacité…). D'autre part, elle contient des méthodes à ne surtout pas utiliser (par exemple indexOf), car d'une complexité en O(n) plutôt qu'en temps constant.

En effet, sous java, il est préférable d'utiliser des tableaux plutôt que des vecteurs car cela économise les appels de méthode, la classe vecteur étant trop générique.

Solution : les ArrayList

C'est pourquoi, nous avons finalement décidé d'utiliser des ArrayList afin de stocker notre graphe. Le principal avantage de la structure de donnée « ArrayList » est qu'elle a des fonctions similaires à un vecteur.

En effet, la classe ArrayList est une encapsulation du tableau avec la possibilité de le rendre dynamique en taille. C'est pourquoi, il est grandement préférable d'utiliser des ArrayList plutôt que des Vecteurs car ces 2 structures de données sont très similaires en ce qui concerne les fonctionnalités mais aussi très différents en ce qui concerne le temps d'exécution. En effet, l'ArrayList est beaucoup plus rapide à l'exécution car moins générique que la classe Vector.

```
// La liste des Sommets
private ArrayList<Sommet> listeSommets;
// La liste des successeurs
private ArrayList<ArrayList<Successeur>> listeSucc;
// On initialise le nb de sommets
private int nbSommet = 0 ;
```

FIGURE 1 : STRUCTURE DE DONNEE POUR LA REPRESENTATION DU GRAPHE

b. Affichage du graphe

Nous avons, pour cela, utiliser l'outil graphviz (disponible sur www.graphviz.org). Nous avons tout d'abord récupérer une classe spécifique permettant d' « interfacer » notre programme avec graphviz en générant directement des images au format gif. Malheureusement, cette classe Java s'est avérée limitée dans ses fonctionnalités car nous ne pouvions pas, par exemple, choisir la position des sommets sur le graphe. Cette classe avait effectivement pour but de placer les sommets de la meilleure façon possible afin d'être agréable à l'affichage. C'est pourquoi, bien qu'étant particulièrement utile pour la représentation rapide et efficace d'un graphe, cela s'est avéré inadapté à l'affichage d'une carte routière.

Nous avons donc finalement opté pour la génération d'un fichier intermédiaire au format .dot afin, par la suite de générer un gif visualisable grâce à l'outil neato. Cela nous permet de contrôler par nous-mêmes et de manière sûre la représentation du graphe avec, notamment, des options spécifiques pour la position des sommets sur le graphe généré tel que « pos= » .

```
/*
 * Dessine un graphe (soit avec la classe graphviz, soit en generant un fichier dot qui est ensuite
 * charge grace a  la commande neato)
 * option (varargs) est une liste des options affectees Ã  la fonction
 */

public void dessinerGraphe(String...option) throws IOException{

GraphViz gv = new GraphViz();
gv.addln(gv.start_graph());
gv.addln("graph [bgcolor=lightyellow2, splines=true];");
gv.addln("edge [color=blue, arrowsize=0.5, arrowhead=open, labelfontsize=7, labelfontname=Verdana, color=green];");
gv.addln("node [color=lightgray, style=filled, fontname=\"Verdana\", " +
        "fontsize=8, shape=box, width=0, margin=\"0.0,0.0\", height=0, pack=0, fontcolor=red];");

// Pour la representation du graphe (avec pos pour le placement des sommets)
for(int i=0; i<this.getListeSommets().size();i++){
    if(this.getListeSommets().get(i) != null){
        Sommet som1 = ((Sommet)(this.getListeSommets().get(i)));
        gv.addln("\""+som1.getNom()+"\" [pos=\" "+(int)(som1.getLongitude()*100)+", "
                +(int)(som1.getLatitude()*100)+"\"];");
        for(int j=0; j<((this.getListeSucc().get(i))).size();j++){
            if((this.getListeSucc().get(i)).get(j) != null){
                Successeur succ1 = ((Successeur)((this.getListeSucc().get(i)).get(j)));
                Sommet som2 = ((Sommet)(this.getListeSommets().get(succ1.getNumSommet())));
                gv.addln("\""+som1.getNom() + "\" -> \"" + som2.getNom()+"\";");
            }
        }
    }
}
gv.addln(gv.end_graph());
```

FIGURE 2 : METHODE DESSINER GRAPHE POUR LA GENERATION DU GRAPHE

Afin d'effectuer la génération de l'image pour le graphe, nous avons crée une fonction dessinerGraphe dans la classe graphe. Cette fonction prend en argument une liste d'option (String) que l'on récupère afin de connaître quelle méthode l'utilisateur souhaite effectuer, il peut indiquer « graphviz » ou « neato » au choix.

```
for (String current : option)
{
    // methode en generant le fichier dot
    if (current.compareTo("neato")==0) {
        FileWriter out = new FileWriter("notreGraphe.dot");
        out.write(gv.getDotSource());
        out.flush();
        String cmd = "neato -s -n2 -Tjpg -ocarte.jpg notreGraphe.dot";
        try {
                Runtime r = Runtime.getRuntime();
                Process p = r.exec(cmd);
                p.waitFor();//si l'application doit attendre a ce que ce process fini
            }catch(Exception e) {
                System.out.println("erreur d'execution " + cmd + e.toString());
            }
    }
    // methode en utilisant graphviz
    else if (current.compareTo("graphviz")==0) {
        File out = new File("Graphe.gif");
        gv.writeGraphToFile(gv.getGraph(gv.getDotSource()), out);
    }
}
```

FIGURE 3 : METHODE DESSINER GRAPHE, CHOIX DES OPTIONS

3. CALCUL DES COMPOSANTES CONNEXES

Pour calculer le nombre de composantes connexes du graphe, on part d'un sommet auquel on attribue un numéro de groupe connexe. A partir de ce sommet, on teste alors si les autres sommets lui sont connexes ou non. Lorsque tous les sommets liés au sommet en cours sont traités, on passe au sommet suivant en incrémentant le numéro du groupe connexe.

```
public static int[] calculConnexite(Graphe graph){
    int nbSommets = graph.getNbSommet();
    int[] tableauConnexite = new int[nbSommets];
    int i=0;
    int numGpeConnexe = 1;
    while(i<nbSommets){
        if((tableauConnexite[i] == 0) && (graph.getListeSommets().get(i) != null)){
            testConnexite(tableauConnexite, ((Sommet) graph.getListeSommets().get(i)).getIdSom(), numGpeConnexe, graph);
            numGpeConnexe++;
        }

        i++;
    }

    return tableauConnexite;
}
```

Pour tester la connexité de deux sommets, on regarde si le premier sommet fait partie d'un groupe connexe. Si ce n'est pas le cas on utilise le numéro de groupe connexe courant, sinon on utilise le numéro du groupe dont le sommet fait partie. On récupère ensuite la liste des successeurs de ce sommet, et on leur attribue le même numéro de groupe connexe.

```
private static void testConnexite (int[] tableauConnexite, int sommet, int numGpeConnexe, Graphe graph){

    if(tableauConnexite[sommet] == 0) {
        tableauConnexite[sommet] = numGpeConnexe;
        ArrayList<Successeur> successeurs = graph.getListeSucc().get(sommet);
        if (!successeurs.isEmpty()){

            int j=0;

            while ((j < successeurs.size()) && (successeurs.get(j) != null)) {
                testConnexite(tableauConnexite, ((Successeur) successeurs.get(j)).getNumSommet(), numGpeConnexe, graph);
                j++;
            }
        }
    }
}
```

4. CALCUL DU PLUS COURT CHEMIN

a. Sans contrainte

Dijkstra

L'algorithme de Dijkstra sert à résoudre le problème du plus court chemin entre deux sommets d'un graphe connexe dont le poids lié aux arêtes est positif ou nul.

Initialisation de l'algorithme

```
Initialisation(G,sdeb)
1 pour chaque point s de G
2    faire d[s] := infini              /* on initialise les sommets autres que sdeb à 0 */[3]
3       prédecesseur[s] := 0           /* car on ne connaît au départ aucun chemin entre s et sdeb */
4 d[sdeb] := 0                         /* sdeb étant le point le plus proche de sdeb */
```

Recherche du nœud le plus proche

On recherche le nœud le plus proche (relié par l'arête de poids le plus faible) de sdeb parmi l'ensemble Q de ses successeurs.

Mise à jour des distances

On met à jour des distances entre sdeb et s1 en se posant la question : vaut-il mieux passer par s2 ou pas ?

```
maj_distances(s1,s2)
1 si d[s1] > d[s2] + Poids(s1,s2)
2    alors d[s1] := d[s2] + Poids(s1,s2)
3         prédecesseur[s1] := s2          /* on fait passer le chemin par s2 */
```

```
if(((tab.getValeurAt(numSommet) == 0) || (tab.getValeurAt(numSommet) > valeur)) && (numSommet != sommetTravail)) {
    //si le sommet est marque avec une valeur superieure, on la met a jour
    tab.setValeurAt(numSommet,valeur);
    chemin[numSommet]=sommetTravail; //on sauvegarde le sommet de travail comme pr?decesseur direct du sommet courant
}
```

Fonction principale

```
Dijkstra(G,Poids,sdeb)
1 Initialisation(G,sdeb)
2 Q := ensemble de tous les nœuds sauf sdeb
3 tant que Q n'est pas un ensemble vide
4       faire s1 := Distance_minimum(Q)
5          Q := Q privé de s1
```

```
6           pour chaque nœud  s2 voisin de s1
7               faire maj_distances(s1,s2)
```

Le plus court chemin de somDepart à somFin est ensuite affiché à l'envers (du sommet de fin à celui du début) grâce au chemin stocké et mis à jour au fur et à mesure de l'algorithme :

```
//etablissement du trajet du plus court chemin
System.out.println("Plus court chemin : "+tab.getValeurAt(somFin));

int i=1;
sommetTravail = somFin;
System.out.println("Arrivee : "+sommetTravail+" ("+ graph.nom(sommetTravail)+")");
while(sommetTravail != somDepart){
    System.out.println("etape - "+i+" : "+chemin[sommetTravail]+" ("+ graph.nom(chemin[sommetTravail])+")");
    i++;
    sommetTravail = chemin[sommetTravail];
}
```

Dijkstra – implémentation :

```
public static Trajet grandDijsktra(int somDepart, int somFin, Graphe graph,String param){

        Trajet trajet = new Trajet();
        int sommet_min;
        double valeur_min;
        double valeur;
        int sommetTravail;

        int[] chemin = new int[graph.getNbSommet()];

        //tableau contenant pour chaque sommet du graphe, un boolean et la valeur calculee
        tabDij tab =  new tabDij(graph.getNbSommet());

        sommetTravail = somDepart; //pour la premiere iteration
        while(!tab.getMarqueAt(somFin)){ //tant que le sommet qui nous interesse n'est pas marque, on continue
            //on prend les successeurs de notre sommet
            ArrayList<Successeur> succ = graph.getListeSucc().get(sommetTravail);

            //on met a jour les distances des successeurs non marques a true
            if(!succ.isEmpty()){ //si y a des successeurs

                for(int j = 0; j < succ.size(); j++){
                    if(!tab.getMarqueAt(((Successeur) succ.get(j)).getNumSommet())){ //si le successeur n'a pas ete marque
                        Successeur leSucc = (Successeur) succ.get(j);
                        int numSommet = leSucc.getNumSommet();

                        if(param.equals("distance"))
                            valeur = leSucc.getValuationArc().getDistance() + tab.getValeurAt(sommetTravail);
                        else if(param.equals("temps"))
                            valeur = leSucc.getValuationArc().getDistance(); //il faut calculer le temps de parcours !!
                        else valeur = -1;

                        if(((tab.getValeurAt(numSommet) == 0) || (tab.getValeurAt(numSommet) > valeur)) && (numSomm
                            tab.setValeurAt(numSommet,valeur);
                            chemin[numSommet]=sommetTravail; //on sauvegarde le sommet de travail comme pr?decesseur direct du sommet courant
                        }
                    }
                }
            }
            //on marque le sommet lui mm
                tab.setMarqueAt(sommetTravail);

            //selection du prochain sommet a marquer -- on le selectionne parmis les sommets non marques mais ayant deja une distance allou?e
                valeur_min = 0;
                sommet_min = 0;
                for(int i = 0;i<tab.getLenght();i++){
                    if((!tab.getMarqueAt(i) && tab.getValeurAt(i) > 0) && (valeur_min == 0 || tab.getValeurAt(i) <
                        valeur_min = tab.getValeurAt(i);
                        sommet_min = i;
                    }
                }
            //on passe au prochain sommet
                sommetTravail = sommet_min;
        }//Fin du marquage du graphe


            sommetTravail = somFin;
            trajet.addEtape(sommetTravail);
            while(sommetTravail != somDepart ){
                trajet.addEtape(chemin[sommetTravail]);
                sommetTravail = chemin[sommetTravail];
            }
            trajet.setDistance(tab.getValeurAt(somFin));
            return trajet;

}
```

Dans notre implémentation, l'algorithme retourne un objet de type Trajet définit ainsi :

```
public class Trajet {

    Stack<Integer> pileEtape ;
    private double distance;
    private int taille;
    private int deb;
    private int fin;

    public Trajet(){
        pileEtape = new Stack<Integer>();
        taille = 0;
    }

    public void addEtape(int etape){
        pileEtape.push((Integer) etape);
        if(taille == 0)
            deb = etape;
        else
            fin = etape;
        taille++;

    }
```

```
//affiche un trajet
public static String afficheTrajet(Trajet t, Graphe g){
    String retour = "";
    while(!t.isEmpty())
        retour += g.nom(t.etape())+"\n";
    retour += t.taille()+" Etapes - "+t.getDistance()+"Km\n";
    return retour;
}
```

C'est une classe simple, englobant une pile qui stocke les étapes les unes après les autres, l'avantage de la pile, c'est lors de l'affichage du trajet, on a directement les étapes dans l'ordre (l'affichage se fait grâce à la méthode afficheTrajet).

Optimisé avec un tas

L'algorithme de Dijkstra peut être aussi mis en œuvre de manière plus efficace en utilisant un tas comme une file à priorités pour réaliser la fonction de recherche du minimum. Pour un graphe possédant m arcs et n nœuds, en supposant que les comparaisons des poids d'arcs soient à temps constant, alors la complexité de l'algorithme devient : $\sigma[(m+n) * \ln(n)]$.

STRUCTURE DE TAS

Un tas est un arbre binaire complet. On dit qu'un arbre est ordonné en tas lorsque les nœuds sont ordonnés par leurs clés respectives, et pour tous A et B nœuds de l'arbre tels que B soit un fils de A alors clé(A) ≥ clé(B).

Cette propriété implique que la plus grande clé soit située à la racine du tas. Ils sont ainsi très utilisés pour implémenter les files à priorités car ils permettent des insertions en temps logarithmique et un accès direct au plus grand élément. L'efficacité des opérations effectuée sur des tas est très importante dans de nombreux algorithmes sur les graphes. Le fait qu'un tas soit un arbre binaire complet permet de le représenter d'une manière intuitive par un tableau unidimensionnel.

Nous avons donc implémenté notre structure de tas en créant, tout d'abord, la classe InfosTas qui contient les données que nous allons insérer dans notre structure de tas soit le numéro du sommet ainsi que le coût.

```java
public class InfoTas {
    private int idSom;
    private float cout;

    //Constructeur pour les informations affÃ©rentes au tas
    public InfoTas(int som, float cout) {
        super();
        // TODO Auto-generated constructor stub
        this.cout = cout;
        idSom = som;
    }

    //Renvoie le cout
    public float getCout() {
        return cout;
    }

    //Affecte le cout
    public void setCout(float cout) {
        this.cout = cout;
    }

    //Renvoie l'identifiant du sommet
    public int getIdSom() {
        return idSom;
    }
```

FIGURE 4 : INFOSTAS POUR LA STRUCTURE DE TAS

Nous avons aussi implémenté un tableau permettant, à tout moment, de connaître la place dans le tas d'un indice donné. En effet, notre tableau des positions est indicé par le numéro d'un sommet et permet de renvoyer la place de ce sommet dans le tas. Ce qui permet d'accéder à n'importe quel élément contenu dans le tas en temps constant.

```
public Vtas(int nbSommet) {
    vTas = new InfoTas[nbSommet+1];
    pos = new int[nbSommet];
}

//insertion de sommets dans le tas
public void inserer(int numSom, float cout){
    //On le met a la fin temporairement
    vTas[dernierIndice]= new InfoTas(numSom,cout);
    this.pos[numSom] = dernierIndice;
    //On verifie que le sommet est bien a sa place
    if(dernierIndice > 0)
        this.moveUp(dernierIndice);
    //On augmente l'indice de sommet de 1
    dernierIndice++;
}

//retourne l'element en haut du tas
public InfoTas getTete(){
    return vTas[0];
}

//vire la tete et fait remonter la nouvelle tete
public void retirerTete(){
    echange(0,dernierIndice-1); //on remplace la tete du tas par son dernier ?l?ment
    dernierIndice--; // on supprime en quelque sorte l'ancienne tete
    moveDown(0); // et on reconstruit le tas
}
```

FIGURE 5 : STRUCTURE DE TAS, CONSTRUCTEUR ET INSERTION

Djikstra et le tas :

La façon dont on utilise le tas pour optimiser l'algorithme de Dijsktra est la suivante :

On part du sommet de départ, on l'insère dans le tas (avec une distance égale à 0), puis on fait de même avec ses successeurs (distance = 0 + valeur de l'arc), et avec les successeurs des successeurs etc. L'intérêt réside dans le choix du prochain sommet à traiter, en effet, de part les propriétés du tas, celui ci se retrouve automatiquement en tête du tas.

Pour obtenir le sommet à traiter, il suffit de retirer la tête du tas (dans notre implémentation, on envoie ce sommet tout au fond du tas, au delà du dernier élément du tas, ce qui permet de pouvoir y accéder tout de même via son numéro de sommet et le tableau pos[] mais il n'interfère en aucune façon sur le tas), et de récupérer la nouvelle tête du tas. Dans les faits, de façon empirique (nous n'avons pas pu faire des tests plus poussés faute de temps), on observe pour le calcul du plus court chemin entre Toulouse et Paris, pour Dijsktra normal, entre 5 et 10 secondes de calculs, avec le Dijsktra optimisé entre 20 et 150 ms en temps utilisateur (et non en temps effectif). La différence est toutefois suffisamment significative pour bien montrer l'avantage du tas.

b. Avec contrainte

Selon autonomie des batteries (150km)

Trouver un chemin d'un point à un autre en respectant la contrainte des batteries S'est avéré problématique.

Après bien des hésitations et fausses pistes, nous avons décidé d'implémenter une méthode qui bien que non exacte dans l'absolu, fournit tout de même de très bons résultats, très proche de la réalité.

Cette méthode consiste à créer un autre graphe, qui contient uniquement les sommets correspondants aux stations listés dans les fichiers .stations.

Ce graphe, un peu différent du graphe utilisé pour l'ensemble des sommets est une instance de la classe petitGraphe (voir sources). Ce graphe a besoin d'être initialisé si ses sommets sont les stations (facilement accessibles), ses arcs correspondent à un calcul de plus court chemin entre les deux sommets (en plus de la valeur en km de l'arc, il faut aussi stocker le chemin parcouru, grâce à une instanciation de la classe Trajet, par arc). Il faut donc calculer tous les chemins entre toutes les stations et sectionner seulement ceux qui sont d'une longueur inférieure à 150 Km (au delà l'arc n'est pas traversable). Sachant que pour la France, il y a une centaine de stations, il faudrait calculer environs 5000 plus courts chemins (5000 * 100 ms = 8 min de calculs !).

Pour simplifier les choses et limiter le nombre de calculs, nous avons choisit de ne calculer que les chemins vraisemblables c'est à dire que pour chaque sommet, on calcule la distance euclidienne (ou distance à vol d'oiseau) entre ce sommet et tous les autres, et on ne cherche des chemins que lorsque cette distance est inférieure à150 Km (méthode construirePetitGraphe dans les sources). L'initialisation du graphe est alors quasi-instantanée.

Dès lors, trouver un chemin entre deux sommets, en respectant cette contrainte est facile, il suffit d'insérer dans ce graphe simplifié (si besoin) les deux sommets à joindre et de trouver le plus court chemin entre ces deux points.

C'est à ce niveau que se trouve notre approximation, en effet, pour des raisons de simplicité lors de l'insertion d'un sommet dans le petit graphe, nous lui « inventons » des arcs entre lui et toutes les stations dans un rayon inférieur à 150Km.

Par manque de temps, nous n'avons pas pu faire afficher le chemin complet résultant de ce calcul, nous n'affichons que les stations traversés (pour afficher la totalité il suffirait de récupérer au fur et à mesure les Trajets correspondants aux différents arcs, et de les mettre bout à bout).

Travaillant sur un graphe d'une centaine de sommets, cette méthode est bien évidemment très rapide (de l'ordre d'1 ou 2ms le calcul), et même les calculs préalables sont rapides (1/2 secondes).

Selon autonomie variable

Nous n'avons pas pu faire cette partie, néanmoins, nous pouvons en expliquer le principe. Pour pouvoir faire des calculs en fonction du temps, il faut modifier plusieurs choses dans notre code :

- La valuation des arcs : prise en compte de l'altitude, calculer la vitesse moyenne par arc et les temps de parcours dans un sens puis l'autre.
- Prise en compte de la valeur temps dans les différents algorithmes Dijsktra.
- Au niveau des classes SommetDij et InfoTas, pas de changement, la variable coût (ou valeur) peut servir indifféremment au temps ou à la distance.

- Pour avoir un calcul en fonction de l'autonomie, il faut reprendre le principe précédent, il est toujours valable, il faut juste remplacer les limites150Km par des limites à 2h.

CONCLUSION

Nous espérons que le fruit de notre travail s'avérerera essentiel pour l'aboutissement du projet de MonTrajetElectrique.com et servira à aider aux mieux les utilisateurs écologistes de ce service afin qu'il dispose d'un service fiable leur permettant d'effectuer un trajet le plus rapidement possible avec un minimum d'encombres et d'inconvénients.

Et ainsi contribuer à une vie meilleure en protégeant notre belle planète grâce à une émission fortement diminué des gaz à effet de serre.

Bilan des apprentissages et du groupe

Ce projet s'est avéré réellement intéressant par le fait qu'il nous a permis de réaliser de manière pratique les connaissances théoriques vues en cours.

Ceci a constitué une grande source de motivation pour chacun des membres du groupe. Nous avons notamment su nous répartir les tâches au fur et à mesure afin de valider les différentes étapes constituant ce projet.

Nous regrettons seulement le fait d'avoir eu un emploi du temps chargé à la fin du de ce semestre ce qui a, malheureusement, constitué un désagrément afin de pouvoir approfondir en détail les fonctionnalités de nos programmes. Mais, nous avons grandement apprécié le fait d'avoir un peu plus de temps car c'est un projet tout de même conséquent.

Sur le plan pédagogique, ce projet nous a permis d'approfondir les concepts de la théorie des graphes en les appliquant au sein d'un travail concret. Nous avons aussi pu découvrir différents outils pour la génération des graphes tels que graphviz.

Globalement, ce projet a constitué une expérience enrichissante et constitue un outil pédagogique motivant.