PROJET D'ALGORITHMIQUE ET PROGRAMMATION OBJET

L'ITINÉRAIRE

Année scolaire 2020-2021

Groupe 2/12

Louis BERTHIER, Colin BERTHIOL et Johanna BERLIET

Sommaire

I. Synthèse du projet

Dans la vie de tous les jours, nous sommes toujours confrontés à des choix de chemins pour se diriger d'un point A à un point B. Notre décision finale est guidée par diverses raisons: vouloir prendre son temps et passer par un chemin beau et verdoyant, emprunter le chemin le plus court quitte à passer par des péages, passer par le chemin le plus rapide même si la consommation sera probablement plus importante... Finalement il existe de nombreux itinéraires disponibles et comme le dit cette vieille expression: "Tous les chemins mènent à Rome". Ainsi se pose notre problème, quel itinéraire suivre pour se diriger d'un point à un autre parmi les chemins, les villes et les lieux proposés?

Si nous avons décidé de coder un algorithme proposant différents itinéraires, c'est surtout pour apprécier l'évolution visuelle de notre code à travers l'interface graphique et le résultat qui affiche le chemin le plus favorable selon les conditions renseignées. De plus, cela permet de se renseigner sur de nombreux algorithmes que l'on doit chercher à comprendre et mettre en application comme l'algorithme de voisin le plus proche ou encore l'algorithme de Dijkstra. Nous avons hésité avec l'algorithme qui s'occupe du traitement des images notamment avec le changement de couleur des pixels, le floutage... Cependant l'un d'entre nous avait déjà eu l'occasion de coder un programme similaire en terminale au lycée, nous avons donc préféré proposer quelque chose de nouveau pour chacun des membres ! De plus ce projet nous permet d'avoir une certaine satisfaction personnelle avec l'évolution des algorithmes et d'autant plus avec l'interface graphique, puis cela nous permet de coder des algorithmes qui se posent plutôt sur la réflexion que sur une logique et des principes mathématiques. Ce projet nous apparaissait donc comme l'aboutissement du savoir théorique que nous avions accumulé jusqu'à présent.

Le développement de notre algorithme d'itinéraire s'est déroulé sur plusieurs mois et il permet de choisir le chemin d'un point de départ à un point d'arrivée à

travers le chemin le plus rapide ou le plus court et en empruntant ou non un péage. Nous avons aussi développé plusieurs cartes parfois à l'échelle d'un pays et parfois à l'échelle d'une région afin de modifier les destinations et de rendre l'expérience plus intéressante chez l'utilisateur. Avec le visuel de la carte, on peut essayer de deviner le chemin en amont pour aller d'un point A à un point B - cette dernière respecte une échelle - et se rendre compte que le chemin attendu n'est pas toujours le même selon nos critères. En effet certaines routes peuvent être rallongées pour les cols de montagne ou plus rapide si l'on a accès à une autoroute ! Toutefois notre algorithme n'est pas un jeu et n'est pas aussi distrayant qu'une modification d'image à laquelle on peut appliquer n'importe quel paramètre pour toujours avoir un résultat différent.

Au cours du développement de notre projet, nous nous sommes confrontés à plusieurs difficultés que nous avons surmontées à travers de la recherche documentaire et des problèmes similaires ou via l'aide de notre professeur. Avant de commencer à coder, nous avons réfléchi au diagramme de classes et aux relations entre les différentes classes. Finalement, nous avons utilisé 5 classes afin de représenter et créer les chemins, les villes, les lieux, la carte et la fenêtre graphique. Au premier abord, l'interface graphique proposée ainsi que l'idée du code ne semblaient pas trop complexes mais cela n'était qu'une impression, même si nous avons eu une bonne vision d'ensemble pour essayer de décortiquer les problèmes ! Au début nous avons eu de "petits" soucis comme la création des points, distinguer les villes des lieux, ou encore lire correctement nos cartes afin de les importer et de les afficher. Cependant, l'algorithme qui nous a posé le plus de problèmes au cours de notre projet reste le développement de l'algorithme du chemin le plus court ou du chemin le plus rapide - les deux sont identiques, il suffit de changer l'indentation des valeurs lues par l'algorithme sur la carte. Essayer d'éviter l'ensemble des erreurs possible en jeu est apparu comme un réel challenge puisqu'il fallait penser à toutes les éventualités en cherchant à être le plus exhaustif possible: que se passe-t-il si l'on renseigne des points qui n'existent pas? Que se

passe-t-il si un chemin n'est accessible qu'à travers un péage et que l'utilisateur décide de ne pas emprunter de péage? Par ailleurs, nous avons été confrontés à une intense phase de débogage aussi longue que difficile. Effectivement, lorsque l'on réalise un projet de la sorte, nous travaillons de façon à écrire des choses qui fonctionnent. C'est pour cela que la détection des soucis d'un programme est difficile. Cette phase de correction des différentes erreurs nous a appris aussi bien l'intérêt du travail en équipe que celui de la réalisation de tests individuels des méthodes écrites. Effectivement, le fait d'être en équipe permet d'apporter son point de vue sur le travail réalisé par l'autre. Ainsi, à l'aide d'une approche différente, il est possible de trouver des solutions aux soucis rencontrés par nos collègues. Concernant le fait de tester individuellement les méthodes, cela est très utile si l'on veut éviter de perdre du temps ! En effet, cela nous a permis d'identifier très rapidement d'où venait les problèmes. Généralement il s'agissait d'une erreur liée à la création de la nouvelle méthode mais parfois il pouvait y avoir des problèmes entre différentes classes et méthodes qui se faisaient appel. Même si cela nous prenait du temps, cette rigueur nous a probablement permis d'éviter de perdre encore plus de temps.

Nous mettons en avant uniquement les mauvais problèmes, toutefois il est aussi important de mettre en exergue que ce projet nous a apporté de bons moments: notre joie lorsque nous avons réussi à lire la carte même s'il n'y avait pas les points; notre euphorie lorsque nous avons réussi à afficher les chemins, les lieux et tout le reste de notre interface graphique même si notre méthode du chemin le plus court n'était pas encore développée; notre satisfaction lorsque l'ensemble de notre programme fonctionnait en nous indiquant quel chemin nous devions choisir… Ce sont ces bons moments qui nous font apprécier la programmation, quand après des heures de problèmes nous parvenons à trouver une solution et à corriger notre erreur qui peut se révéler parfois toute bête. Effectivement, nous n'étions pas les meilleurs codeurs mais nous avons tout de même compris en quoi cela peut être aussi satisfaisant que plaisant.

Ce projet a été une expérience à la fois intéressante et enrichissante puisque nous avons pu appliquer et comprendre des leçons étudiées en classes avec un objectif plus concret, personnel et ludique. De plus, nous avons suivi l’évolution de notre travail et de nos progrès depuis septembre, ce qui s’est avéré très satisfaisant.

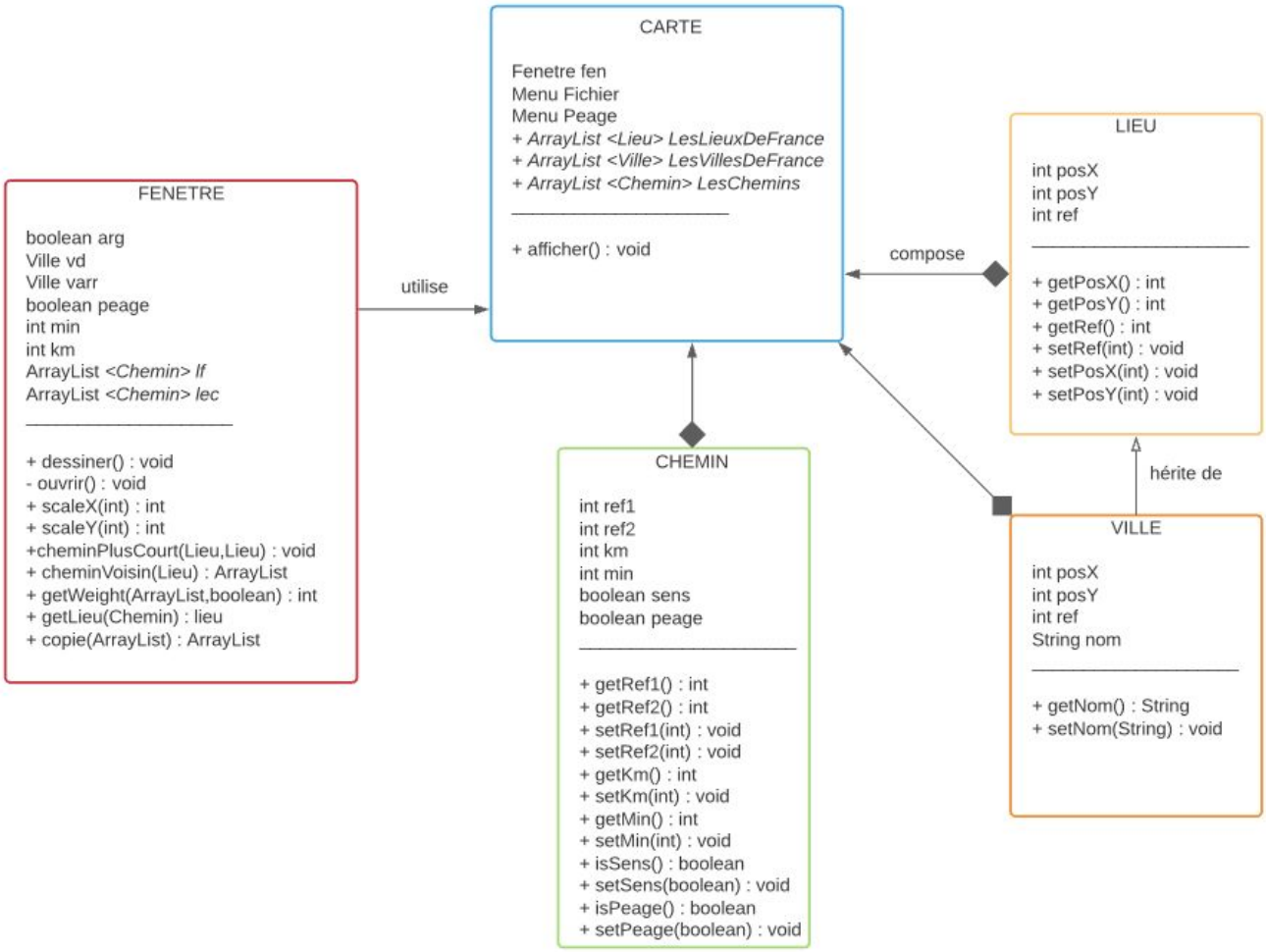
II. Analyse

Nous avons créé 5 classes: Lieu, Ville, Chemin, Carte et Fenêtre. Les trois premières sont assez simples et définissent simplement ce qu'est un lieu, une ville ou un chemin. La classe fenêtre correspond à notre interface graphique et elle permet de définir les différentes méthodes pour choisir son itinéraire alors que notre classe carte correspond à la classe qui nous permet de lire la carte choisie, de vérifier que la lecture est bonne ainsi que de créer et placer des points . Un chemin est caractérisé par différents attributs à savoir: le point de départ et le point d'arrivée, la distance et la durée ainsi que la présence d'un péage ou non et s'il s'agit d'un chemin à sens unique ou non. Toutes les villes sont des lieux mais tous les lieux ne sont pas des villes, ainsi Ville hérite de Lieu est est donc caractérisée par sa référence, ses coordonnées ainsi que son nom alors qu'un lieu n'a pas de nom! La classe Fenêtre est clairement la classe la plus complexe de notre programme et celle qui nous a posé le plus de soucis.

Contrairement à une bataille navale il ne s'agit pas d'un jeu donc l'interaction homme-machine n'est pas très intéressante ou enrichissante et contrairement à un algorithme qui permet de modifier les photos, nos modifications sont restreintes et doivent être proposées par le développeur et non l'utilisateur (notamment avec les cartes dématérialisées sous forme de fichier texte).

Dans l'état actuel des choses, l'utilisateur n'a pour choix que la carte, l'absence ou la présence d'un péage et s'il veut passer par le chemin le plus court ou le plus rapide. En réfléchissant et en cherchant à développer un peu plus le code, on aurait pu proposer une fenêtre de navigation où l'utilisateur peut choisir un point essentiel de passage avant d'arriver à destination, le passage par la mer ou encore éviter les routes montagneuses avec présence d'un certain danger... Cet algorithme itinéraire constitue donc les prémices d'un GPS.

III. Diagramme de classes



IV. Algorithmes

1. Algorithmes principaux

A. Déterminer le chemin le plus court/rapide

Cette méthode a été le cœur de notre sujet mais aussi de nos problèmes. En effet, le principe qui s'y applique est facilement compréhensible: parmi toutes les possibilités de chemins, il faut sélectionner à travers différents critères celui qui est le plus court ou le plus rapide. Toutefois, même si l'idée est simple, le procédé s'avère beaucoup plus complexe à mettre en place. En effet l'algorithme de Dijkstra ou encore de l'AEtoile témoigne de cette complexité.

Voici notre code :

```
public void cheminPlusCourt(Lieu a, Lieu b) {
    int ba = 0; // Initialise sile chemin n'est pas emprunté
    for (Chemin c : cheminVoisin(a)) { // On regarde tous les voisins proches de notre lieu de départ A
        if (peage == false && c.isPeage() == false) { // c.isPeage == false indique la présence d'un péage
            return; // On ne compte pas ce chemin dans la liste des chemins voisins
        }
        for (Chemin ch : lec) { // On regarde tous les chemins en cours
            ba = 0;
            if (c.getRef2() == ch.getRef1() || c.getRef2() == ch.getRef2()) { // Si l'arrivée du chemin traitée (c)
                //correspond à la ville de départ OU d'arrivée de tous les chemins en cours (ch)
                ba = 1; // Indique que le chemin est emprunté pour regarder les autres chemins voisins
                break; // On sort de la boucle pour éviter une boucle infinie
            }
        }

        if (ba == 0) { // Si le chemin n'a pas été emprunté
            lec.add(c); // On ajoute le chemin à la liste des chemins en cours
            if (c.getRef2() == b.getRef()) { // Si ville d'arrivée du chemin traité = ville d'arrivée souhaitée
                if (lf.isEmpty()) { // Si on a pas de chemins finaux
                    lf = copie(lec); // On ajoute ce chemin
                } else {if (getWeight(lf, arg) > getWeight(lec, arg)) { // On compare les "poids" donc soit
                    //le temps soit la distance des différents chemins finaux possibles
                        // getWeight est une méthode decrite plus tard
                        lf = copie(lec); // On prend le chemin dont le temps ou la distance est le plus "petit"
                    }
                }
            }
            else {cheminPlusCourt(getLieu(c), b); // Appel récursif à partir de la ville d'arrivée du chemin traité
                //si on a pas atteint la ville souhaitée
            }
            lec.remove(c); // On retire le chemin choisi parmi les chemins en cours pour éviter de repasser dessus
        }
    }
    km = getWeight(lf, true); // Permet de récuperer la valeur de la distance du chemin final
    min = getWeight(lf, false); // permet de récupérer la valeur de la durée du chemin final
}
```

On reviendra sur la méthode du cheminVoisin juste après mais il suffit de savoir qu'elle permet de savoir tous les chemins possibles selon le lieu choisi. Ainsi Lieu a correspond au lieu de départ et Lieu b à celui d'arrivée. Pour chaque chemin voisin du lieu de départ, on regarde s'il y a la présence d'un péage et si la demande de l'utilisateur valide ou non la présence de ce dernier. Si les deux sont en adéquation, alors on conserve le chemin sinon on le sort de la liste.

Ensuite pour chacun des chemins en cours on vérifie si la ville atteinte suite au chemin correspond à la destination finale souhaitée, si c'est le cas on donne à notre variable "ba" la valeur de 1 pour la suite du code et on sort afin d'éviter un problème de "boucle infinie".

On vérifie la valeur de notre variable, et si le chemin n'a pas été emprunté (ba==0) on va à nouveau vérifier si la ville atteinte est celle d'arrivée souhaitée. Si ce n'est pas le cas, on fait appel de façon récursive à la méthode du chemin le plus court. Une fois la ville d'arrivée atteinte, on crée une copie (pour éviter les problèmes liés aux adresses des variables) de notre chemin en cours qui finalement correspond à un chemin final et ensuite chaque chemin final sera comparé selon le poids des variables que sont km pour la distance ou min pour le temps. Finalement, une fois le chemin final comparé, on le retire de la liste afin d'éviter de le traiter à nouveau.

B. Trouver les chemins voisins à partir d'un lieu

Comme énoncé précédemment, la méthode du chemin voisin est primordial pour le bon fonctionnement de la méthode du chemin le plus court. A nouveau, le principe est très simple et très clair: trouver tous les chemins issus d'un lieu de départ.

Voici notre code:

```
public ArrayList<Chemin> cheminVoisin(Lieu l) {

    ArrayList<Chemin> lcv = new ArrayList(); // Liste des chemins voisins
    for (Chemin c : carte.LesChemins) {
        if (l.getRef() == c.getRef1()) { // On forme la liste des chemins voisins,
            //tous les chemins qui partent de se lieu
            lcv.add(c);
        }
    }
    return lcv;
}
```

Pour chaque chemin de la carte on vérifie si les chemins du lieu de départ permettent de conduire à la ville d'arrivée désirée. Si c'est le cas alors on ajoute ce chemin dans une liste qui correspondra à nos différents chemins possibles.

C. Créer la carte avec les lieux, villes et chemins

Cette méthode est indispensable pour mener à bien notre projet. En effet, ce dernier repose sur une carte composée des différents lieux, villes et chemins du fichier sélectionné. La méthode se charge de lire une à une les lignes du fichier découpées à la présence de chaque "espace".

Voici notre code:

```
public Carte(String fic){
    try {
        BufferedReader br = new BufferedReader(new FileReader(fic)); //pour lire un fichier
        String lu = br.readLine(); //pour lire chaque ligne du fichier
        while(lu!=null){
            String [] dec = lu.split(" "); //décompose la ligne en X éléments à chaque espace
            switch(dec[0]){

                case "P": // le cas où on a P = Point
                    Lieu lieu = new Lieu(Integer.parseInt(dec[1]),Integer.parseInt(dec[2]),Integer.parseInt(dec[3]));
                    // parseInt transforme un string en entier
                    // On crée un lieu avec numéro du point, coordonnée x, coordonnée y
                    LesLieuxDeFrance.add(lieu);
                    break;

                case "V": // le cas où on a V = Ville
                    Ville ville = new Ville(Integer.parseInt(dec[1]),Integer.parseInt(dec[2]),Integer.parseInt(dec[3]),dec[4]);
                    // On crée une ville avec numéro de la ville, coordonnée x, coordonnée y, nom de la ville
                    LesVillesDeFrance.add(ville);
                    LesLieuxDeFrance.add(ville);
                    break;
```

```
                case "C": // le cas où on a C = Chemin
                boolean doubleSens = true; // On initialise  la présence d'un double sens
                boolean gratuit = true; // On initialise l'absence d'un péage
                if("u".equals(dec[5])){
                    doubleSens=false; // On indique que c'est à sens unique
                }
                if(dec.length==7 && "$".equals(dec[6])){
                    gratuit=false; // On indique qu'il y a un péage
                }
                Chemin chemin = new Chemin(Integer.parseInt(dec[1]),Integer.parseInt(dec[2]),Integer.parseInt(dec[3]),
                        Integer.parseInt(dec[4]),doubleSens,gratuit);
                /* On crée un chemin avec point de départ, point d'arrivée, la longueur, la durée,
                le sens et s'il y a un péage */
                LesChemins.add(chemin);

                if(doubleSens){
                 Chemin cheminInv = new Chemin(Integer.parseInt(dec[2]),Integer.parseInt(dec[1]),Integer.parseInt(dec[3]),
                        Integer.parseInt(dec[4]),doubleSens,gratuit);
                   LesChemins.add(cheminInv);
                }
            }


            lu=br.readLine();
    }

    } catch (IOException | NumberFormatException ex) {
        ex.printStackTrace();
    }
    afficher();
}
```

On cherche à "créer" les différents points et passages de notre carte, et pour ça on vient lire notre fichier texte qui correspond à notre carte "dématérialisée". Par la suite, l'utilisation d'un switch nous permet d'identifier la validité de la lettre renseignée. On crée un nouveau lieu, chemin ou une nouvelle ville en convertissant les chaînes de caractères en valeur numérique. Une fois l'élément créé, on l'ajoute à sa liste correspondante. Pour le chemin on vérifie la présence ou l'absence d'un péage et d'un double sens. En cas de double sens, on crée un chemin inverse.

2. Algorithmes secondaires

A. Afficher la lecture de notre fichier texte

Au début, nous avions des problèmes d'affichage de carte. En effet, les points étaient placés de façon désordonnée, en contradiction avec les valeurs inscrites sur notre fichier texte représentant notre carte. Cette méthode a pour but de montrer et d'assurer l'utilisateur que le fichier texte est lu correctement et dans le "bon" ordre.

Voici notre code:

```
public void afficher(){

    for(Lieu l:LesLieuxDeFrance){ // vérifie si l'élément a bien été crée comme une ville au début
         if(l instanceof Ville){
             Ville v = (Ville)l; // cast = transtypage où l devient une Ville qui s'appelle v

        System.out.println("V "+v.ref+ " "+v.posX+ " "+v.posY+ " "+v.nom);// On affiche les villes + leurs attributs
         }else{
             System.out.println("P "+l.ref+ " "+l.posX+ " "+l.posY); // On affiche les lieux + leurs attributs
         }
    }
    for (Chemin c:LesChemins){
        System.out.println("C "+c.ref1+ " "+c.ref2+ " "+c.km+ " "+c.min+ " "+c.sens+ " "+c.peage) ;
            // On affiche les chemins + leurs attributs

    }
}
```

Tout d'abord nous voulions distinguer les lieux des villes. On regarde donc si parmi tous nos lieux certains ont été créés en tant que ville au début. Si c'est le cas, on utilise le transtypage afin de reconvertir le lieu en tant que simple ville et ensuite on affiche les attributs de cette dernière. Sinon on considère simplement un lieu que l'on affiche aussi. On fait de même pour les chemins.

B. Dessiner les éléments sur notre carte

Notre projet relève aussi d'une grande partie graphique notamment avec l'affichage et la distinction entre nos différents éléments.

Voici notre code:

```
public void dessiner() {

    int n = carte.LesLieuxDeFrance.size(); // Pour la boucle afin de parcourir l'ensemble des lieux
    int Xmax = 0;
    int Ymax = 0;
    for (int i = 0; i < n; i++) {
        Lieu lieu = carte.LesLieuxDeFrance.get(i);
        if (lieu.getPosX() >= Xmax) {
            Xmax = lieu.getPosX(); // On récupère la valeur de la position maximale en abscisse des lieux lus
        }
    }
    for (int i = 0; i < n; i++) {
        Lieu lieu = carte.LesLieuxDeFrance.get(i);
        if (lieu.getPosY() >= Ymax) {
            Ymax = lieu.getPosY(); // On récupère la valeur de la position maximale en ordonnée des lieux lus
        }
    }

    int scaleX = scaleX(Xmax); // Appel de la méthode scaleX pour redimensionner
    int scaleY = scaleX(Ymax);

    int n2 = carte.LesChemins.size(); // Pour la boucle afin de parcourir l'ensemble des chemins
    for (int j = 0; j < n2; j++) {
        Chemin chemin = carte.LesChemins.get(j);
        Lieu lieu1 = carte.LesLieuxDeFrance.get(chemin.getRef1() - 1); // Problème car les ref commencent à 1 et le compteur à 0
        Lieu lieu2 = carte.LesLieuxDeFrance.get(chemin.getRef2() - 1); // Permet de corriger le problème d'indice
        gc.setStroke(Color.BLUE); // On choisit la couleur bleu "initiale" pour les chemins
        gc.strokeLine(lieu1.getPosX() * scaleX, lieu1.getPosY() * scaleY, lieu2.getPosX() * scaleX, lieu2.getPosY() * scaleY);
        // On applique la couleur aux chemins
    }
```

```
    for (int i = 0; i < n; i++) {
        Lieu lieu = carte.LesLieuxDeFrance.get(i);
        gc.setFill(Color.ORANGE);
        gc.fillRect((lieu.getPosX() * scaleX) - 2, (lieu.getPosY() * scaleY) - 2, 4, 4);
    }
    int n3 = carte.LesVillesDeFrance.size();
    for (int i = 0; i < n3; i++) {
        Ville lieu = carte.LesVillesDeFrance.get(i);
        gc.setFill(Color.RED); // On choisit la couleur de point qui représente la ville
        gc.fillRect((lieu.getPosX() * scaleX) - 2, (lieu.getPosY() * scaleY) - 2, 4, 4);
        gc.setStroke(Color.BLACK); // On choisit la couleur du texte du nom de la ville
        gc.strokeText(lieu.getNom(), (lieu.getPosX() * scaleX) - 2, (lieu.getPosY() * scaleY) - 2);
    }
}
```

Tout d'abord on cherche à redimensionner la carte en trouvant l'abscisse et l'ordonnée maximale avec nos deux premières boucles for et ensuite on applique les méthodes scaleX et scaleY qui viennent retrancher à 630 la valeur de X ou Y max (afin d'avoir une carte possédant tous les points). Ensuite on va colorier nos chemins représentés par une ligne ou nos villes et lieux représentés par des carrés. Pour les villes on va aussi afficher le nom juste au-dessus du point correspondant.

C. Calculer la durée ou la distance du chemin "entier"

La méthode GetWeight est utilisée dans la méthode cheminLePlusCourt. Elle permet de déterminer la distance ou la durée totale d'un trajet allant d'un point de départ à la ville d'arrivée.

Voici notre code :

```
public int getWeight(ArrayList<Chemin> lc, boolean arg) {
    int n = 0;
    for (Chemin c : lc) {
        if (arg) {
            n = n + c.getKm(); // On ajoute la distance successive de chacun des chemins
        } else {
            n = n + c.getMin();// On ajoute la durée successive de chacun des chemins
        }
    }
    return n;
}
```

Pour chaque fragment de chemin sélectionné, la méthode ajoute la durée ou la distance de ce dernier à la durée ou à la distance totale du trajet établi jusqu'à présent.

V. Mode d’emploi

L’interface de GPS que nous avons codée est simple d'utilisation. En voici un mode d’emploi bref et exhaustif :

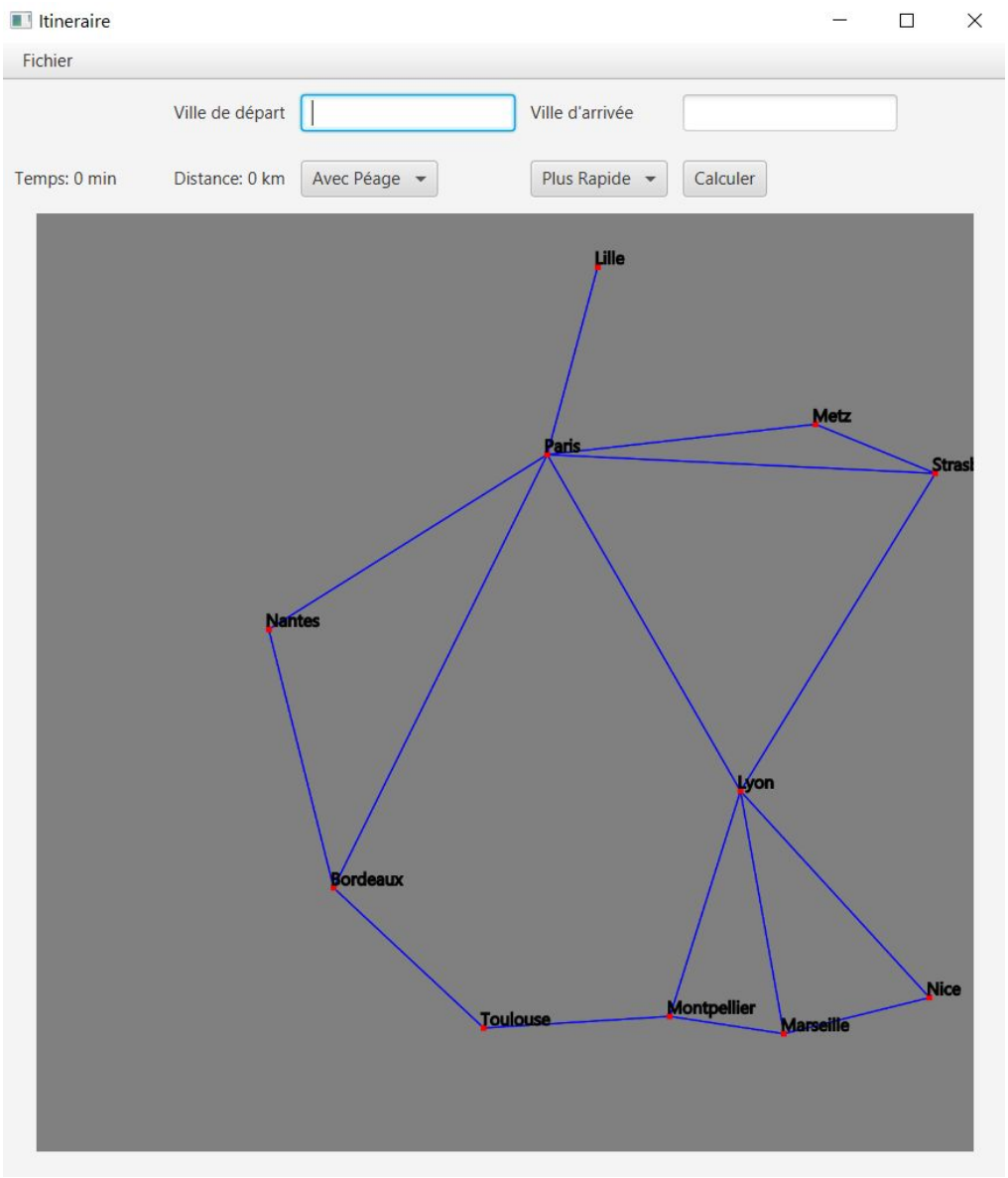
1. Utilisez le ctrl+E afin d’ouvrir votre fichier texte “carte”. Votre carte s’affiche.
2. Renseignez la ville de départ et celle d’arrivée de votre itinéraire.
3. Choisissez à l’aide du menu déroulant si vous acceptez la présence de péage sur votre itinéraire.
4. Choisissez à l’aide du menu déroulant si vous souhaitez emprunter le chemin le plus court (en terme de distance) ou le plus rapide (en terme de durée).
5. Appuyez sur le bouton “Calculer”.
6. Visionnez en rouge l’itinéraire le plus intéressant et adapté à vos contraintes.
7. Visionnez en haut à gauche de l’interface la durée et le kilométrage du trajet.
8. Vous pouvez changer toutes les données renseignées autant de fois que souhaité.
9. Utilisez le ctrl+Q afin de fermer l’interface graphique.

Bon voyage!

VI. Tests

1. Carte de France

Lorsque nous affichons notre carte de France dématérialisée nous obtenons le résultat suivant:

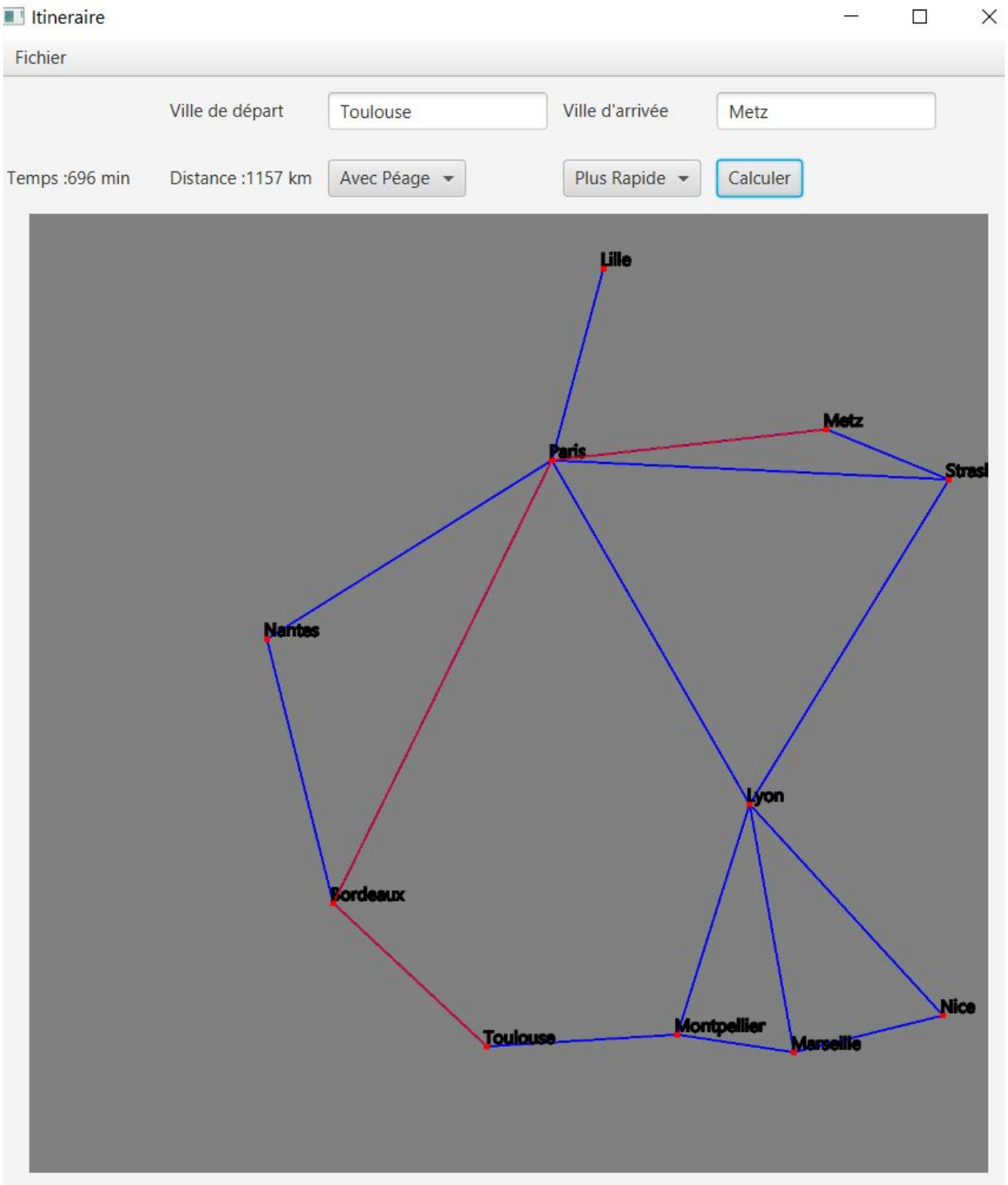


Au début nous n'avions pas traiter certaines erreurs: un nom de ville corrompu ou inexistant, aucune ville renseignée… C'est pourquoi nous avons ajouté le traitement de ces différentes erreurs dans le constructeur de la classe fenêtre:

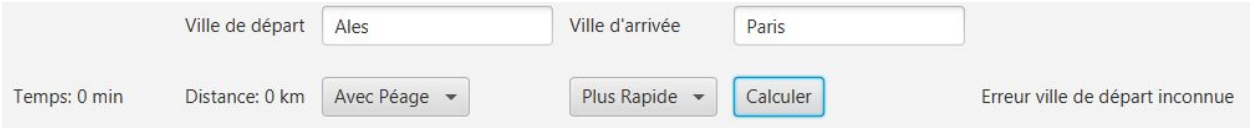
```
if (lu.isEmpty()) {
    err = true;
    error.setText("Erreur ville de départ vide");
}
String lu2 = tfa.getText();
if (lu2.isEmpty()) {
    err = true;
    error.setText("Erreur ville d'arrivée vide");
}
```

```
if (!err) { // On cherche une correspondance entre la ville rentrée et les villes dans le fichier lu
    for (Ville v : carte.LesVillesDeFrance) {
        if (v.getNom().equals(lu)) {
            vd = v;
        }
        if (v.getNom().equals(lu2)) {
            varr = v;
        }
    }
    if (vd == null) {
        err = true;
        error.setText("Erreur ville de départ inconnue");

    }
    if (varr == null) {
        err = true;
        error.setText("Erreur ville d'arrivée inconnue");
    }
}

if (err) {
    return;
}

if(vd.ref==varr.ref){
    error.setText("Erreur ville de départ et arrivée identique ");
    return;
}
```

On peut s'amuser à tester un chemin:



Quelques corrections d'erreurs:

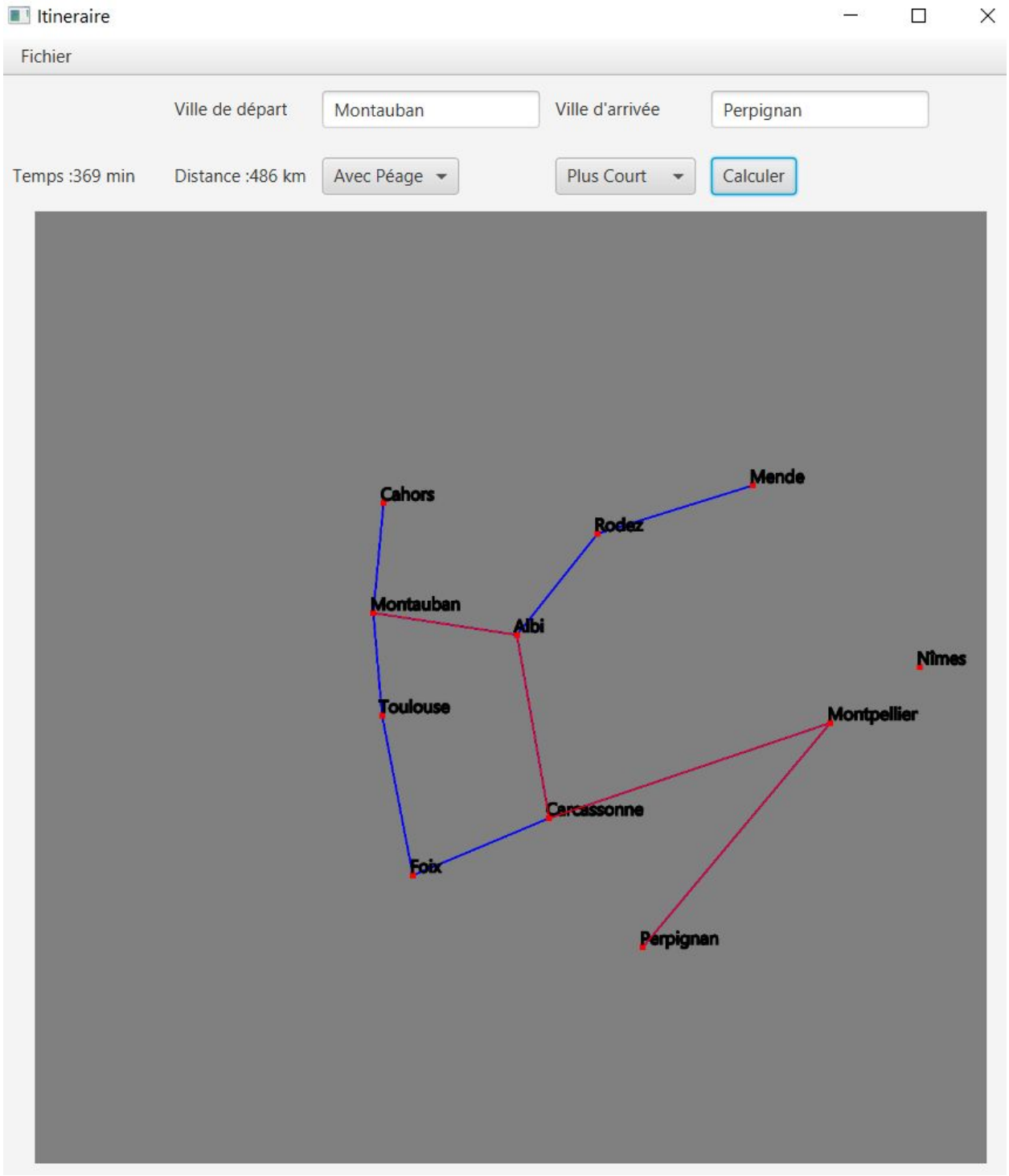


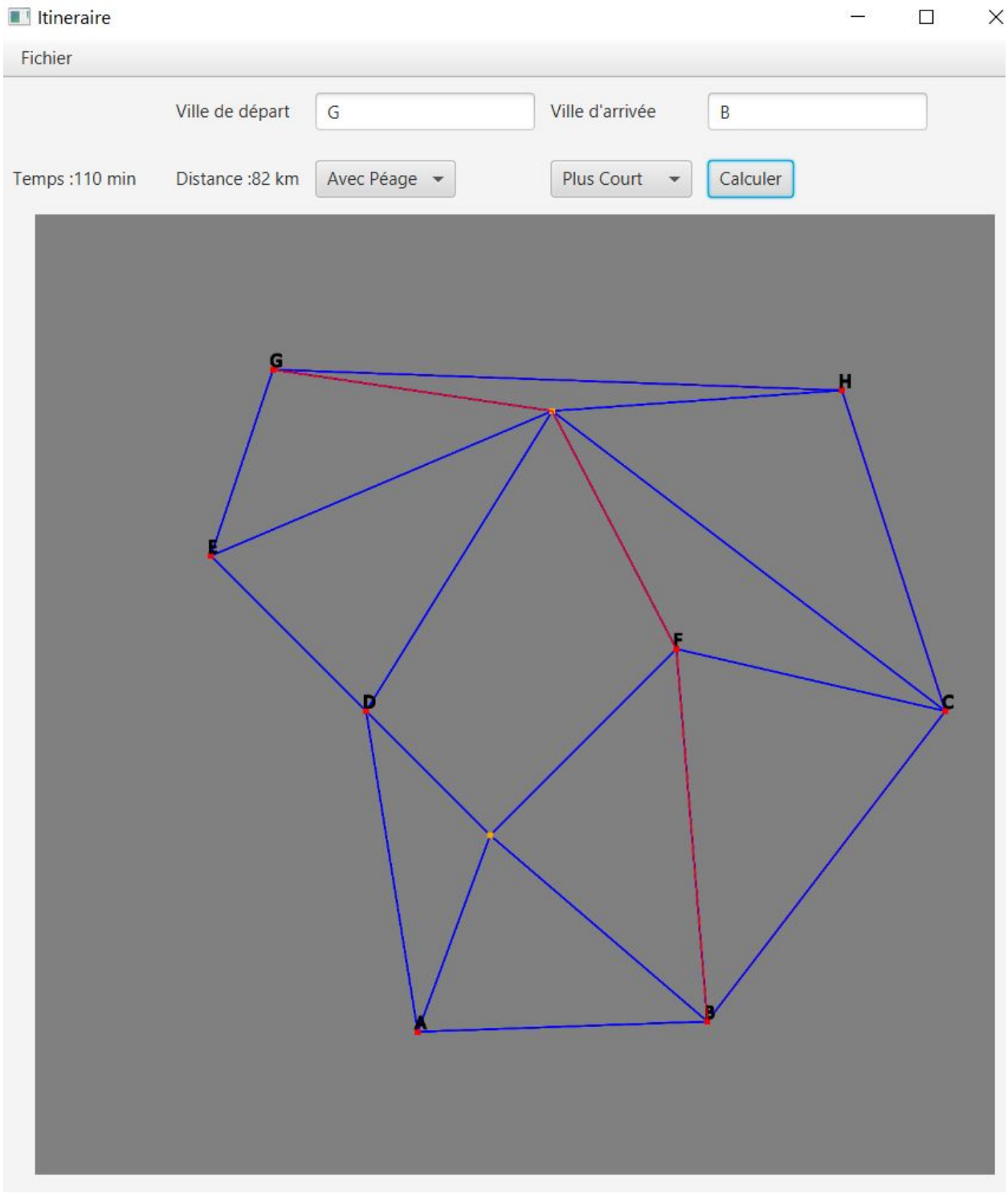
Voici notre carte de France dématérialisée:

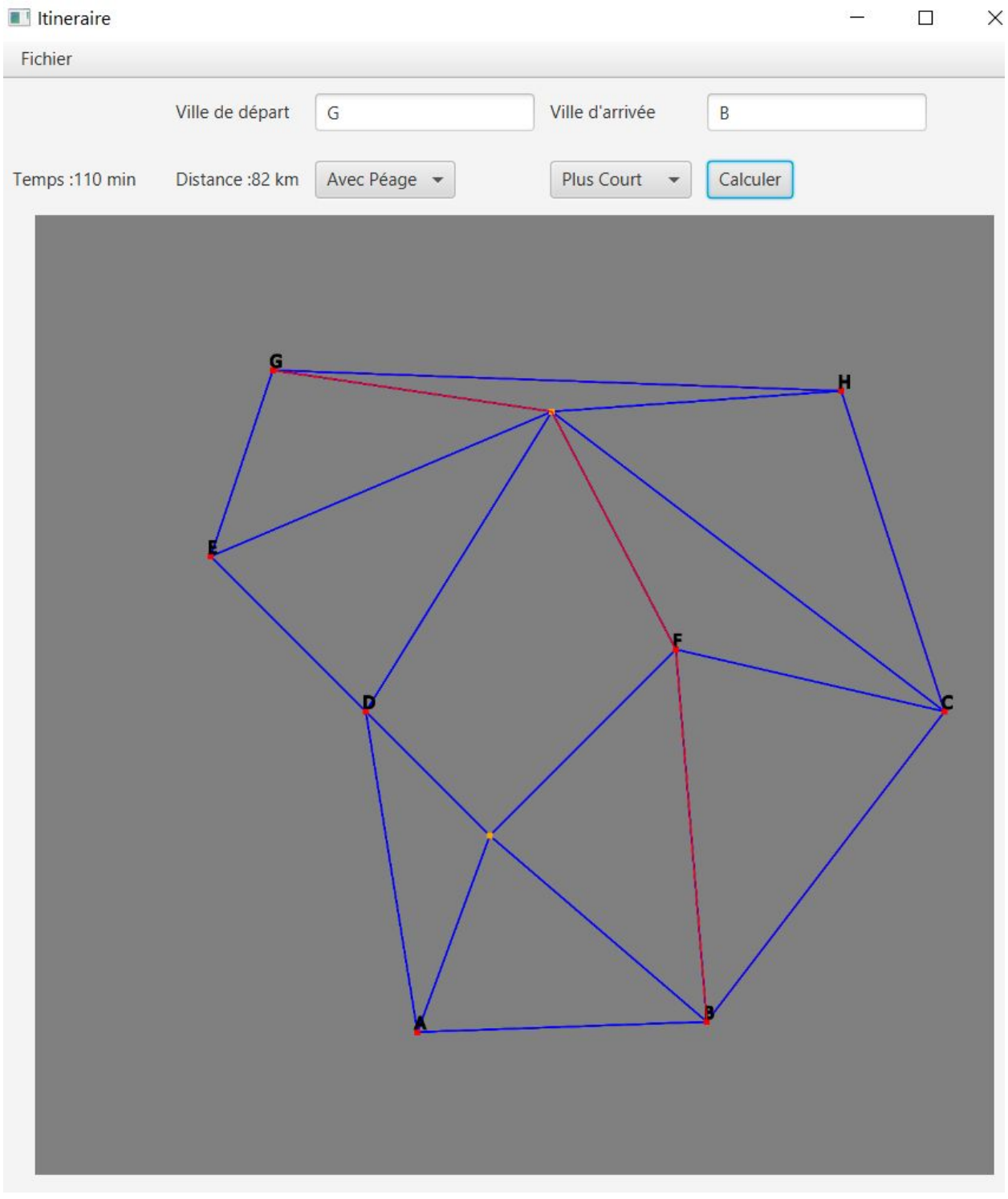
```
V 1 354 167 Paris
V 2 518 568 Marseille
V 3 488 400 Lyon
V 4 310 564 Toulouse
V 5 619 543 Nice
V 6 161 288 Nantes
V 7 623 180 Strasbourg
V 8 439 556 Montpellier
V 9 206 467 Bordeaux
V 10 389 37 Lille
V 11 540 146 Metz
C 1 3 464 310 d
C 1 6 383 324 d $
C 1 7 491 302 d
C 1 9 583 365 d $
C 1 10 220 176 d $
C 3 2 314 185 d
C 3 5 471 258 d $
C 3 7 487 270 d
C 3 8 304 187 d $
C 2 5 199 134 d
C 2 8 170 146 d $
C 4 8 243 255 d
C 9 6 347 206 d $
C 9 4 244 151 d
C 11 1 330 180 d
C 11 7 165 97 d $
```

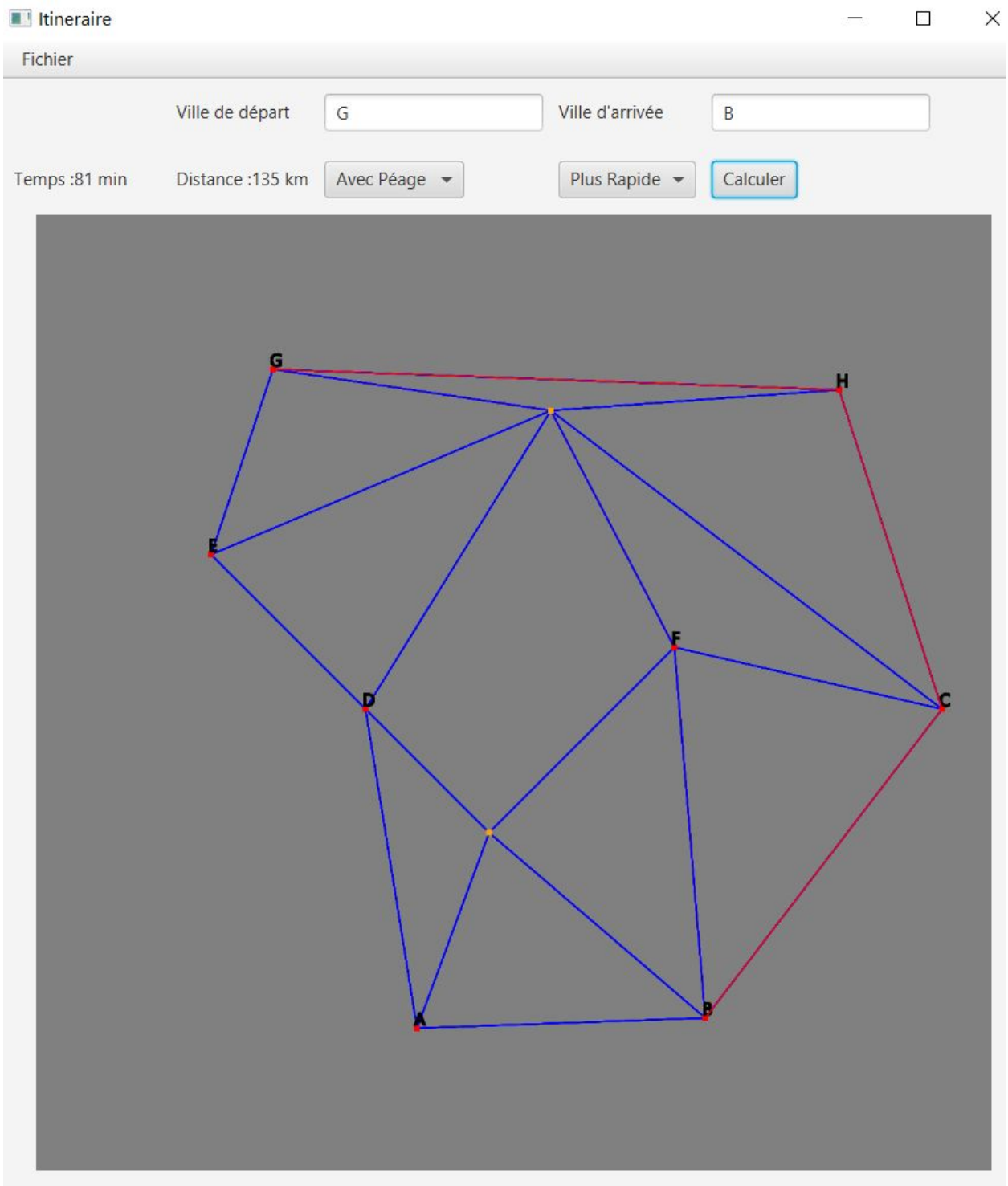
2. Carte d'Occitanie et carte du professeur

Voici notre test pour nos différentes cartes:




Itineraire
Fichier
Ville de départ
G
Ville d'arrivée
B
Temps :110 min
Distance :82 km
Avec Péage
Plus Court
Calculer
G
H
E
F
D
C
A
B


Itineraire
Fichier
Ville de départ
G
Ville d'arrivée
B
Temps :110 min
Distance :82 km
Avec Péage
Plus Court
Calculer
G
H
E
F
D
C
A
B


Itineraire
Fichier
Ville de départ
G
Ville d'arrivée
B
Temps :81 min
Distance :135 km
Avec Péage
Plus Rapide
Calculer
G
H
E
F
D
C
A
B