



Année universitaire 2019-2020

# Projet Informatique : Itinéraires pour se déplacer dans le métro parisien

## Livrable 1

## 12/02/2020

Sous la direction de Pierre Sutra

Membres du projet :

- Manjing Mao
- Maximilien Melan
- Kexin Biao
- Paul Orluc
- Shicheng Liu

## Sommaire

<b>Cahier des charges</b> .....	3-4
Fonctionnalités de l'application.....	3
Objectifs Principaux .....	3
Objectifs additionnels .....	3
Détails techniques du projet .....	4

<b>Regroupement modulaire/Tâches à réaliser .....</b>	<b>5-7</b>
Regroupement modulaire des fonctionnalités .....	5
Ensemble des tâches à effectuer.....	6-7
Ce qui a déjà été fait.....	7

<b>Possibles ajouts et tests.....</b>	<b>7</b>
---------------------------------------	----------

<b>Rapport livrable 3 .....</b>	<b>8-20</b>
Fonctionnement du programme.....	8-13
Guide utilisateur.....	13-19

# 1.Cahier des charges

L'objectif de notre projet informatique est de réaliser une application permettant à l'utilisateur de trouver l'itinéraire le plus court (ou avec le moins de changements de ligne) entre 2 stations du métro parisien. L'application a pour objectif d'être performante : calcul rapide de l'itinéraire. De plus, elle doit être fiable. Nous devons faire en sorte qu'il ne soit pas possible à l'utilisateur de saisir le nom d'une station incorrecte de façon à éviter les cas d'erreur.

Rentrons maintenant plus en détail dans les fonctionnalités de notre application. Grâce à notre application, l'utilisateur doit donc pouvoir, en entrant une station de métro A et une autre station B, trouver l'itinéraire le plus rapide en métro pour aller de A à B. Il s'agit là de l'objectif principal de notre application. Cependant, nous souhaitons aussi ajouter à celle-ci plusieurs autres fonctionnalités, non primordiales mais qui pourrait aider l'utilisateur dans sa recherche et lui permettre d'affiner sa requête. Nous souhaiterions par exemple que l'utilisateur puisse aussi avoir accès, s'il le souhaite, à l'itinéraire avec le moins de changements (certains utilisateurs privilégiant le confort à la durée préféreront emprunter un itinéraire peut-être plus long mais avec moins de changements. Cela peut donc apporter une véritable plus-value à notre programme).

En outre, même si notre application ne prendra pas en compte les grèves et divers incidents sur les lignes, l'utilisateur pourra s'il le souhaite éviter une(des) ligne(s) (parce qu'il sait qu'il y a des perturbations sur cette ligne par exemple) ou une(des) station(s) (stations fermées pour cause de travaux par exemple), il pourra saisir sa requête dans l'application et celle-ci lui calculera l'itinéraire le plus court (ou avec le moins de changement) sans passer par cette ligne ou par cette station. Enfin, si le temps nous le permet, l'utilisateur pourra aussi, au lieu de saisir des stations de métro, saisir certains lieux publics que lui proposera l'application. Par exemple, s'il souhaite aller depuis Châtelet à la Tour Eiffel, l'application lui calculera le plus court itinéraire en métro pour atteindre son but (au lieu qu'il cherche lui-même les stations proches de la Tour Eiffel pour les saisir ensuite dans l'application).

Passons maintenant au point de vue graphique pour notre application qui est aussi au cœur de notre projet. Notre application comportera donc en fond une carte de Paris. Lorsque l'utilisateur saisira deux stations de métro (ou 2 lieux publics), son itinéraire sera calculé et lui sera affiché. Nous aimerions aussi que l'itinéraire soit tracé sur la carte. Enfin, nous ajouterons une complétion automatique dès que l'utilisateur commencera à saisir le nom d'une station. Cela lui évitera de devoir taper l'ensemble du nom de la station.

L'affiche final de l'itinéraire devrait ressembler à ceci :



Il y a donc deux grandes parties dans notre application :  
 une partie très algorithmique : implémenter l'algorithme du plus court chemin, adapter notre algorithme pour qu'il puisse prendre en compte les contraintes saisies par l'utilisateur... ainsi qu'une partie plus graphique pour que l'application soit agréable d'utilisation (nous revenons sur cette subdivision dans la **2<sup>ème</sup> partie**).

Après avoir détaillé les fonctionnalités de notre programme, nous allons maintenant développer les détails plus techniques de notre projet :

Pour notre projet, le langage de développement choisi est Java et nous réaliserons notre application sous un environnement Linux.

Nous n'avons aucune contrainte budgétaire pour notre projet.

Il n'y a pas non plus de public type visé par notre application, elle s'adresse à tout ceux qui souhaitent emprunter le métro parisien.

Enfin, les dates importantes du projet sont :

Le 10 février 2020 avec la remise du « Livrable 1 » qui est ce résumé de notre projet.

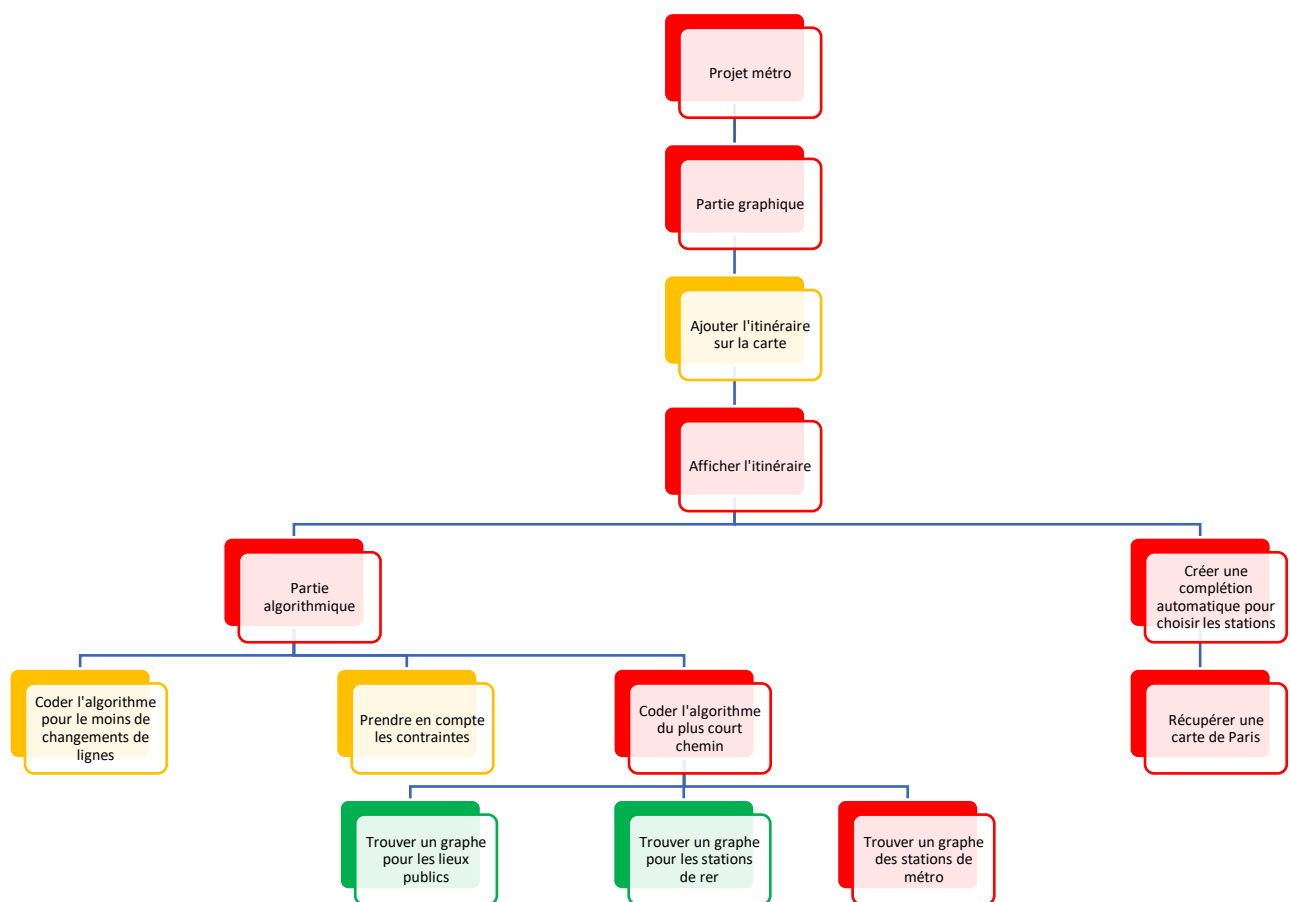
Le 16 mars 2020 avec la remise du « Livrable 2 ». Il s'agira d'exposer un prototype de notre logiciel. Pour atteindre les objectifs du Livrable 2, il suffit que ce prototype soit le squelette des développements futurs du logiciel.

Le 18 mai 2020 avec la remise du « Livrable 3 ». Le livrable 3 sera constitué des éléments suivants : le logiciel, les tests et le rapport.

Enfin, le 2 ou le 3 juin 2020, notre groupe réalisera la soutenance du projet.

## 2.Regroupement modulaire/Tâches à réaliser

Voici résumé le **regroupement modulaire des fonctionnalités** ainsi que le **flux des données à travers les modules** de notre projet (**en rouge** les points essentiels de notre projet, **en jaune** les options qui nous semblent importantes, **en vert** les tâches facultatives que nous réaliserons si nous avons le temps). Une tâche « haute » demande à ce que toutes les tâches plus basses soient réalisées.



Nous résumons maintenant plus en détail chaque tâche à effectuer dans le but de la réalisation de notre application avec toutes ses fonctionnalités mentionnées plus haut. Nous devons :

**Trouver le graphe des stations du métro parisien :**

Cela veut dire que nous devons trouver un fichier contenant l'ensemble des stations du réseau métropolitain (qui constitueront les sommets de notre graphe) ainsi que pour chaque station, toutes les stations voisines ainsi que la durée pour aller d'une station à une autre station voisine (afin de créer les arrêtes pondérés reliant les sommets voisins de notre graphe). Ce fichier devra aussi contenir les coordonnées géographiques des stations. Enfin, il est important de noter qu'une même station reliée à 2 lignes différentes sera représentée comme 2 stations différentes (par exemple la station Odéon, qui est à la fois une station de la ligne 4 et de la ligne 10 sera considérée comme 2 stations différentes : une station Odéon sur la ligne 4 et une autre station Odéon sur la ligne 10)

**Coder l'algorithme de Dijkstra :**

Cet algorithme a pour but de calculer le plus court chemin d'un sommet à un autre dans notre graphe. Une fois codé, notre application réalisera donc déjà sa fonction première qui est le calcul de l'itinéraire le plus court entre 2 stations de métro.

**Coder un algorithme similaire à celui de Dijkstra mais qui cette fois calcul l'itinéraire avec le moins de changements :**

Cette fois l'objectif n'est plus de calculer le plus court chemin mais celui avec le moins de changements de ligne (à noter qu'il faudra donc ajouter à notre base de données la ligne sur laquelle se trouve chaque station).

**Trouver une base de données avec les coordonnées géographiques de lieux publics (monuments, grands magasins, lycées, restaurants connus...) et la rentrer dans notre programme :**

Cela va permettre à l'utilisateur de saisir de nouvelles requêtes, si il souhaite par exemple aller du Louvre à la Tour Eiffel, au lieu d'indiquer directement qu'il souhaite aller de la station Louvre – Rivoli à la station Bir-Hakeim, il saisira simplement le fait qu'il veut aller du Louvre à la Tour Eiffel.

**Reprendre légèrement l'algorithme de Dijkstra dans le cas où l'utilisateur saisit un lieu d'intérêt au lieu d'une station de métro :**

Les modifications sont en fait mineures : l'algorithme doit sélectionner toutes les stations proches du lieu d'intérêt (à moins de 100 mètres par exemple), appliquer l'algorithme de Dijkstra précédent à toutes ces stations en ajoutant le temps qu'il faut pour se rendre des stations d'arrivée au lieu d'intérêt (ce n'est pas parce qu'il est plus rapide de s'arrêter à telle station qu'il s'agit du meilleur itinéraire, il faut aussi s'assurer que l'utilisateur ne marche pas trop de la station jusqu'au point d'intérêt).

**Compléter l'algorithme de Dijkstra pour qu'il prenne en compte des contraintes :**

interdiction de passer sur la ligne 7 par exemple ou éviter la station Havre Caumartin.

### **Enfin, s'occuper du support graphique de notre application :**

Mettre une carte de Paris en fond comme l'image présentée dans le cahier des charges

-Tracer un itinéraire sur cette carte

-Lorsque l'utilisateur commence à saisir le nom d'une station (ou d'un lieu public) proposer toute les stations (et lieu public) qui commence par ce nom :

Exemple : L'utilisateur entre Cha, une complétion automatique apparait avec : Château de Vincennes (Metro), Château d'eau (Métro), Château Landon (Métro), Champs-Élysées-Clémenceau (Métro), Champs de Mars (Lieu d'intérêt). Il faut donc créer une complétion automatique.

-créer un design à l'application pour la rendre plus esthétique

A ce jour (12/02/2020) :

Nous avons trouvé un graphe des stations de métro (à l'adresse

[https://perso.esiee.fr/~coupriem/Graphestp3/metro\\_complet.graph](https://perso.esiee.fr/~coupriem/Graphestp3/metro_complet.graph) , à ouvrir avec le bon

éditeur de texte pour voir comment le graphe a été conçue) contenant leur noms, leurs voisins, le temps qu'il faut pour parcourir chaque arrête et leurs coordonnées

géographiques. Nous avons en outre commencé à écrire l'algorithme de Dijkstra qui est déjà pratiquement opérationnel pour le calcul du plus court chemin.

## **3.Possibles ajouts et tests**

Si nous avons tout effectuer en temps et en heure, nous envisagerons des possibles ajouts à notre application. D'abord nous pourrions essayer d'ajouter les stations de RER, qui ne sont pas présentes dans le graphe original. Ensuite, nous pourrions afficher à l'utilisateur le prix que lui couterait son déplacement. On pourra aussi essayer de prendre en compte certaines particularités du métro parisien (comme le système de fourche par exemple).

De plus, nous réaliserons plusieurs tests lors du développement de notre application. Nous testerons d'abord évidemment nos différents algorithme (Algorithme de Dijkstra, algorithme qui calcule l'itinéraire avec le moins de changement...) à chaque fois que nous réaliserons un nouveau programme. Nous essayerons aussi de déterminer quels sont « les pires cas » (ce qui sont les plus longs à calculer pour l'algorithme). Lorsque notre application sera entièrement réalisée nous testerons si notre programme affiche des résultats cohérents en comparant les résultats obtenus avec ceux d'autres applications similaires. Enfin nous vérifierons la pertinence de l'affichage graphique.

# Rapport pour le livrable 3

Nous ajoutons à notre rapport pour le livrable 1, le rapport suivant qui constitue le rapport du livrable 3.

## 1. Analyse du problème et spécification fonctionnelle

Notre programme manipule des fichiers textes qui fournissent les informations nécessaires pour créer le graphe modélisant le réseau de métro parisien, puis faire les calculs grâce à l'algorithme du plus court chemin. Trois fichiers sont nécessaires au total :

- Un fichier « Stations1.txt » qui donne le nom de toutes les stations dans le réseau ainsi que leur numéro de ligne : ainsi si une station possède  $n$  lignes de métro, son nom apparaîtra  $n$  fois dans le fichier.
- Un fichier « coordonnées1.txt » qui donne les coordonnées (cartésiennes) de chaque station : la  $n$ ème ligne de ce fichier correspond aux coordonnées de la station de la  $n$ ème ligne de Stations1.txt.
- Un fichier « Arrêtes.txt » qui renseigne toutes les arrêtes existantes dans le graphe. Une arrête est renseignée de la manière suivante : «  $n\ m\ p$  » :  $n$  le numéro de la station de départ ( $n$  correspond à la ligne  $n+1$  des fichiers précédents),  $m$  celui de la station d'arrivée, et  $p$  le poids de l'arrête, tous sont des entiers.

Ces fichiers sont la base qui permet de créer les objets sommet, arrête, puis graphe qui sont pris en entrée pour calculer le plus court chemin, ou le chemin avec le moins de changement de station. Notre graphe est dynamique, puisqu'il prends en compte en direct les contraintes imposées par l'utilisateur, telles que ajouter ou supprimer les lignes ou les stations interdites. Si aucun chemin n'est possible avec les contraintes imposées par l'utilisateur, le programme l'indique.

L'interaction avec l'utilisateur se fait via une interface graphique, ou il peut sélectionner toutes les contraintes et lancer l'algorithme du plus court chemin. L'utilisateur peut, s'il le souhaite, visualiser le chemin sur un graphe. Le programme est modulable et fonctionne, en théorie pour n'importe quel réseau du moment qu'on peut fournir les 3 fichiers textes cités précédemment. Certains tests d'intégration sont à prévoir : vérifier qu'il y'a bien un message d'erreur lorsque les fichiers textes sont erronés notamment...



## 2. Conception préliminaire

Notre application se décompose en 6 modules :

- Main : fonction principale, point d'entrée de l'application : affichage de l'interface graphique qui permet d'imposer les contraintes voulues (JavaFX).
- Sommet : Création des sommets pour le graphe
- Arrête : Création des arrêtes pour le graphe
- Graphe : Création du graphe
- PlusCourtChemin : Objet qui sera pris en entrée pour faire les calculs (plus court chemin et moins de changements).
- Afficher : Permet d'afficher le chemin sur un graphe illustré.

- Arrête :

- Paramètres : int origine, int destination, int poids (une arrête part d'un sommet pour arriver sur un autre et possède un « poids », c'est-à-dire le temps, en seconde, qu'il faut pour aller du sommet numéroté « origine » au sommet numéroté « destination »)

- Fonctions : type **get** uniquement. Ces fonctions n'ont pas un mode de fonctionnement très complexe. Elles permettent de retourner les différents paramètres d'une arrête. Par exemple, la fonction **getpoids** retourne le poids de l'arrête sur laquelle on applique la fonction.

- Sommet :

- Paramètres : String nom, int ligne, int distanceànous, boolean visité, ArrayList<Arrête> arrêtes, int prédécesseur, float x, float y.

Le paramètre nom correspond au nom du sommet (par exemple « Châtelet »), ligne correspond à la ligne sur laquelle se trouve le sommet, distanceànous est un paramètre utilisé lors du calcul du plus court chemin, il est initialisé avec une valeur infinie, le booléen visité est aussi utilisé dans le calcul du plus court chemin, arrêtes est la liste des arrêtes qui partent ou arrivent du sommet en question, prédécesseur est aussi utile pour le calcul du plus court chemin. Enfin x et y sont les coordonnées du sommet pour le placer lorsqu'on dessine l'itinéraire.

- Fonctions : On retrouve les fonctions **get**, mais s'y ajoute les fonctions de type **set**. Celle s'y permet de modifier les paramètres du sommet. Cela est utile lorsque nous initialisons notre graphe.

-Graphe :

-Paramètres : Sommet [] sommets, int nombreSommets, ArrayList<Arrête> arrêtes, int nombreArrêtes, int sat, int eat, ArrayList<Sommet> chemin.

Le paramètre sommets est un tableau contenant tous les sommets du graphe, nombreSommets est le nombre de sommets que possède le graphe (il s'agit simplement de `sommets.length`), arrêtes est un tableau contenant toutes les arrêtes du graphe, nombreArrêtes est le nombre d'arrêtes que possèdent le graphe (il s'agit simplement de `arrêtes.length`). Les paramètres sat et eat sont utilisés pour le calcul du plus court chemin. Enfin, chemin permet de stocker l'itinéraire calculé.

-Fonctions : La classe graphe possède tout d'abord un constructeur qui prend en paramètre une ArrayList d'arrêtes. Cela permet d'initialiser le paramètre arrêtes, mais aussi sommets : si on a toutes les arrêtes, on sait combien de sommets on a car une arrête est définie par les numéros de sommets qu'elle joint. Le nombre de sommets d'un graphe à partir de ses arrêtes est d'ailleurs calculé à l'aide de la fonction **calculnombreSommets**. Les fonctions **donnenom**, **donnecord** et **attribueligne** permettent de remplir les champs des sommets du graphe. La fonction **stringtoint** est importante même si son fonctionnement est simple : elle permet, à partir d'une chaîne de caractères (« République » par exemple), de savoir à quel sommet elle correspond : notre algorithme du plus court chemin travaille avec des objets de type sommet mais l'utilisateur saisit des chaînes de caractères. La conversion d'une chaîne de caractères en un sommet est donc capitale. La fonction **getpluscourtedistance** est utilisée pour choisir quel sommet sera visité dans notre algorithme du plus court chemin. La fonction **calculpluscourtedistance** est le cœur du calcul du plus court chemin. Elle prend en paramètre deux string (le nom de la station de départ et le nom de la station d'arrivée) mais aussi deux entiers. En effet, lorsque l'on indique par exemple que l'on souhaite aller de République à Bastille, il y a une légère ambiguïté : souhaite on partir de République ligne 9 ? ou de République ligne 8 ? Souhaite on arriver à Bastille sur la ligne 8 ? ou à Bastille ligne 1 ? En réalité, l'utilisateur se moque de cette distinction (pour lui, le plus important est de partir de République et d'arriver à Bastille) mais pour le programme la différence est capitale car République ligne 8 n'est tout simplement pas le même sommet que République ligne 9. Ainsi les 2 paramètres représentent le nombre de lignes sur lesquelles se trouvent les stations de départ et d'arrivée. Si une station A se trouve sur la ligne 1 et sur la ligne 2, on testera l'algorithme du plus court chemin en partant de la station A sur ligne 1 puis de la station A sur ligne 2, et on gardera le chemin dont la durée est la plus faible.

La fonction **afficherrésultat** est une fonction d'affichage. Elle permet d'afficher le temps d'un itinéraire : si par exemple, le poids total du chemin est de 340 (secondes) elle va retourner « l'itinéraire dure 5 minutes et 40 secondes ». Elle s'occupe donc de convertir le temps en format heures, minutes, secondes. Elle prend en compte l'orthographe (si le trajet dure 2 heures 1 minute et 6 secondes, elle écrira heures avec un s, minute sans le s et seconde avec un s).

La fonction **nombredeligne** donne le nombre de lignes sur lesquelles se trouve une station. La fonction **transformearrête** est utilisée dans le cadre de l'itinéraire avec le moins de changements. Celle-ci vient modifier le poids des arrêtes du graphe (chaque arrête qui correspond à un changement de ligne se voit affecter un poids excessivement grand, ce qui permet d'éviter les changements de lignes et de les prendre uniquement en dernier recours). Il n'y a alors plus qu'à appliquer au graphe nos fonctions de calcul du plus court chemin. Les fonctions **retirerligne** et **retirerstation** permettent de gérer le cas des lignes interdites et des stations interdites en venant mettre un poids immense (bien plus grand encore que pour l'itinéraire avec le moins de changements pour ne pas confondre les arrêtes « interdites » et celles « à privilégier en dernier ») sur les arrêtes concernées. La fonction **transformation** permet de passer d'un itinéraire en sommet à un itinéraire en String (l'utilisateur ne veut pas avoir comme itinéraire [Sommet 654, Sommet 255, ...] mais [République, Oberkampf,...]). Les autres fonctions sont des fonctions qui permettent de retourner certains paramètres du graphe.

-PlusCourtChemin :

-Paramètres : boolean b, String départ, String arrivée, ArrayList<Integer> LignesInterdites, ArrayList<String> StationsInterdites  
Les chaines de caractères départ et arrivée indiquent la station de laquelle on part et celle où l'on arrive. Le booléen b indique quel type d'itinéraire l'on souhaite (si b=False, on calcul l'itinéraire le plus court et sinon on calcul l'itinéraire avec le moins de changements). Enfin, les paramètres LignesInterdites et StationsInterdites indiquent l'ensemble des lignes et des stations interdites.

-Fonctions : La classe possède un constructeur qui permet d'initialiser tous les champs de l'objet PlusCourtChemin. La fonction **afficheChemin** gère tout l'affichage du résultat. Elle s'occupe de retourner la liste des stations qui sont parcourues durant le trajet et d'afficher le plan. Elle permet aussi d'afficher les lignes « spéciales » du métro parisien (3bis et 7bis qui ne sont pas des entiers).

-Main :

- Fonctions : La fonction **start** est la principale fonction de cette classe. Elle permet de créer et d'afficher la fenêtre graphique avec laquelle l'utilisateur va interagir. Cette fenêtre est générée grâce aux outils de JavaFX : des boutons, des listes déroulantes (combobox), et des listes simples (listview) permettent de choisir les paramètres pour calculer puis afficher le plus court chemin.

-Afficher :

- Fonctions : La fonction **handle** affiche le graphe à partir des résultats du plus court chemin.

### 3.Conception détaillée

Voici les pseudo codes de la principale fonction de ce programme, **afficheChemin** :

fonction **afficheChemin** (String : départ, String : arrivée, ArrayList<Integer> : LignesInterdites, ArrayList<String> : StationsInterdites, Boolean : a)

```
// Création du graphe et de l'image
```

```
COORD est une liste de float <— contenu de coordonnées1.txt
```

```
ARRÊTES est une liste de Arrête <— contenu de Arrêtes.txt
```

```
IMAGE est un BufferedImage <— Placement des points de COORD, avec les  
couleurs correspondant aux lignes
```

```
G est un Graphe <— Créé à partir de ARRÊTES
```

```
// On impose les contraintes
```

```
G <— retirerligne(LignesInterdites)
```

```
G <— retirerstations(StationsInterdites)
```

```
// Calcul du plus court chemin
```

H est un ArrayList(String)

SI a :

H ← plus court chemin de départ à arrivée  
// Algorithme de Dijkstra classique

SINON :

H ← chemin avec le moins de changement de départ à arrivée  
// Algorithme de Dijkstra modifié pour avoir le moins de changements

// Affichage du texte et du graphe pour l'utilisateur

TEXTE est un String

SI le chemin existe :

SI a :

TEXTE ← « Le plus court chemin entre départ et arrivée est » + H

SINON :

TEXTE ← « Le chemin avec le moins de changement entre départ et  
arrivée est » + H

IMAGE ← ajouter les arrêtes correspondant au chemin H

SINON :

TEXTE ← « Un tel chemin n'existe pas »

Voici quelques détails sur le fonctionnement de l'interface utilisateur de la classe Main :

- Scène 1

- 4 combobox : Station de départ (Station1.txt), Station d'arrivée (Station1.txt), Stations à interdire (Station1.txt), Choix du type d'itinéraire (« Plus court chemin » ou « Moins de changements »)
- Une ListView : Lignes à interdire
- 3 boutons : deux pour modifier la liste StationsInterdites ( ajouter ou supprimer la station choisie sur la combobox), et un qui exécute afficherChemin et qui passe de la scene 1 à la scène 2

- Scène 2

- Une zone de texte (label) qui affiche H(plus court chemin rédigé)
- Trois boutons : un pour changer de scène (nouveau itinéraire), deux pour afficher l'image du graphe (plus court chemin et moins de changements)

Le contenu des combobox départ et arrivée sont utilisés directement pour créer l'objet PlusCourtChemin qui sert à exécuter afficherChemin. Pour ListView, une conversion de ObservableList vers ArrayList est nécessaire. Le contenu de la combobox type d'itinéraire décide de la valeur du booléen b.

## 4.Codage

Vous trouverez le code complet à l'adresse suivante :  
<https://github.com/larapa/metroprojet/tree/master/src/application>

## 5.Tests

Pour vérifier le bon fonctionnement de chaque programme, une classe Test se joint à notre programme, ou l'on compare les itinéraires avec ceux proposés sur le site de la RATP, nous avons aussi vérifié la cohérence en terme de temps.

Avec le mode débog nous avons pu vérifier pas à pas la cohérence de chaque étape de la création du graphe, du calcul du plus court chemin...

Néanmoins, bien que le programme remplisse son rôle, nous constatons quelques problèmes mineurs :

- La classe plus court chemin n'est pas assez modularisée: nous n'avons malheureusement pas eu le temps de modulariser cette classe. C'est pour cela que nous avons une immense fonction affichechemin qui réalisent beaucoup trop de tâches à la fois.

- Avoir une carte de Paris en fond : Nous n'avons pas trouvé de carte satisfaisante à mettre en fond lorsque nous traçons le trajet.

- Améliorer nos tests : Nous n'avons pas eu le temps d'effectuer beaucoup de tests pour démontrer le bon fonctionnement de notre programme.

- Partie graphique incorporée trop tard sans le projet : cette partie reste un peu bancal, mais nous n'avons pas eu le temps de la perfectionner.

## 6.Manuel Utilisateur

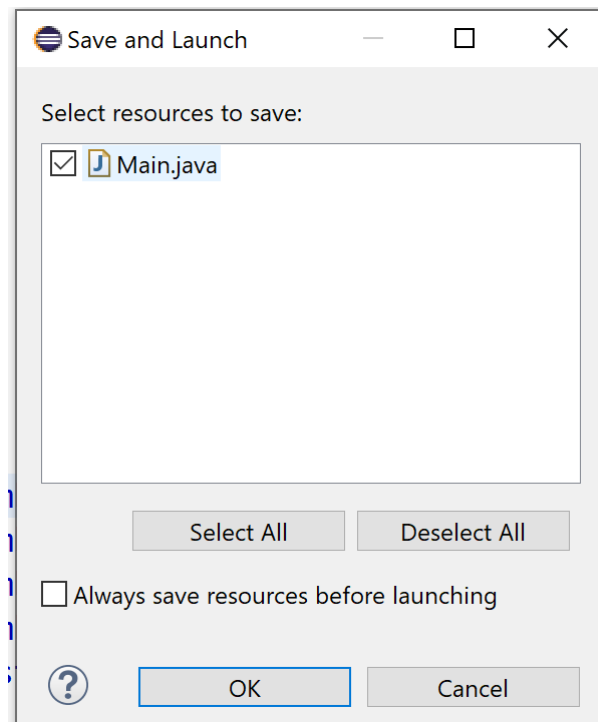
Nous allons, à travers ce tutoriel, guider l'utilisateur afin qu'il puisse faire fonctionner notre programme. Pour cela, l'utilisateur doit au préalable avoir installé Java sur son appareil ainsi qu'un logiciel permettant d'exécuter les projet java, nous vous conseillons pour cela le logiciel eclipse. C'est celui-ci que nous utiliserons pour la suite de notre tutoriel.

Une fois notre programme chargé dans eclipse, il convient, avant de le lancer, d'installer JavaFX (sur la plupart des ordinateurs, JavaFX n'est pas installé par défaut). Pour l'installation de JavaFX, vous trouverez de nombreux tutoriels sur internet.

Une fois ces prérequis effectués, notre programme est prêt à être lancé. Pour cela, cliquez sur la flèche verte en haut d'eclipse :



Si une fenêtre du même type que celle dans l'image ci-dessous apparaît, cliquer sur «ok».



Une fenêtre de cette forme (sans les numéros) devrait alors s'ouvrir :

The screenshot shows a window titled "Menu itinéraire métro" with standard window controls (minimize, maximize, close). The interface includes the following elements:

- Field 1:** A text input field at the top left.
- Field 2:** A dropdown menu labeled "Station d'arrivée" below the first field.
- Field 3:** A large list box containing the following items: 1, 2, 3, 3 bis, 4, 5, 6, 7, 7 bis. A vertical scrollbar is on the right side of the list.
- Field 4:** A dropdown menu labeled "Stations à interdire" below the list box.
- Buttons:** "Ajouter" and "Supprimer" buttons are located below the "Stations à interdire" dropdown.
- Field 5:** A dropdown menu labeled "Choix du type d'itinéraire" below the buttons.
- Field 6:** A "Calcul itinéraire" button at the bottom left.

Les champs 1 et 2 vont vous servir à indiquer respectivement de quelle station vous souhaitez partir et à quelle station vous souhaitez aller. Pour cela, inutile de taper le nom de la station en question, il suffit de la chercher dans le menu déroulant. Le champs 3 correspond aux lignes que vous souhaitez interdire (les lignes par lesquelles vous ne souhaitez pas passer lors de votre trajet). Si vous ne souhaitez pas interdire de lignes, vous n'avez rien à modifier dans ce champ. Par contre, si vous souhaitez supprimer une ou plusieurs lignes, il vous suffit de maintenir la touche ctrl appuyée et de cliquer sur les différentes lignes que vous souhaitez interdire. Si vous avez cliqué par mégarde sur une ligne que vous ne souhaitez pas interdire, il vous suffit de recliquer dessus pour la réautoriser de nouveau.

Le champ 4 correspond aux stations que vous voulez interdire (les stations par lesquelles vous ne souhaitez pas passer lors de votre trajet). De la même façon qu'avec le champ 3, si vous ne souhaitez pas interdire de station, vous n'avez rien à modifier dans ce champ et n'avez pas besoin d'appuyer sur les boutons « Ajouter » et « Supprimer » qui se trouve en dessous. Par contre, si vous voulez interdire une station A, procédez comme suit :

sélectionnez la station A dans le menu déroulant. Ensuite, appuyez une fois sur le bouton « Ajouter ». Voilà, la station est maintenant interdite ! Vous pouvez répéter l'opération avec les autres stations que vous souhaitez interdire. Si vous avez ajouté une station que vous ne vouliez pas interdire, vous pouvez la resélectionner dans le menu déroulant et appuyé sur supprimer. Votre itinéraire pourra de nouveau passer par cette station.

Le champ 5 vous permet de choisir quel type d'itinéraire vous souhaitez (le plus rapide, celui avec le moins de changements de lignes et les deux). Si vous ne sélectionnez rien, le programme vous affichera alors les deux itinéraires.

Il ne vous reste plus alors qu'à cliquer sur le bouton « Calcul itinéraire » du champ 6 pour afficher votre itinéraire.

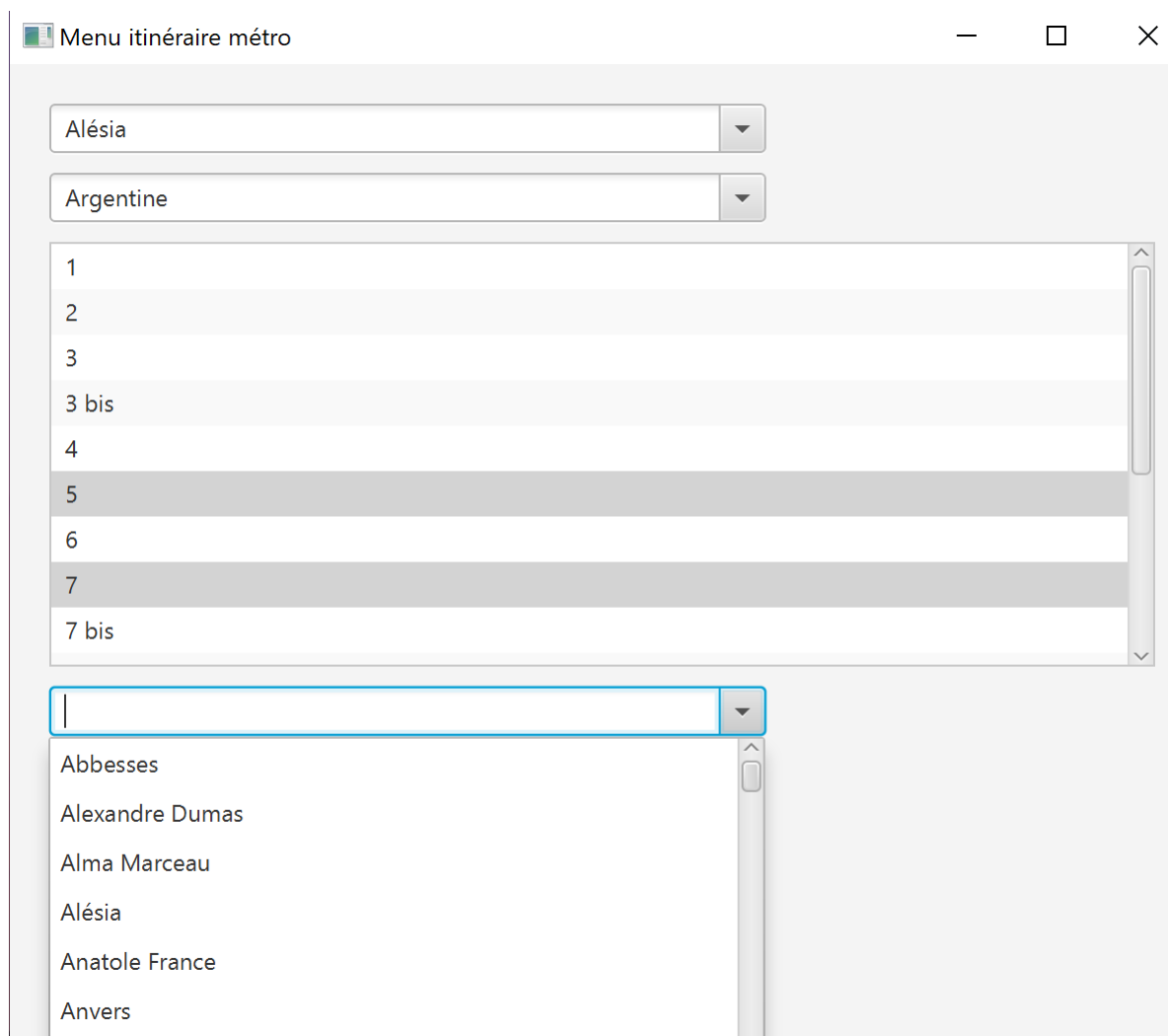
Si dessous un exemple si l'on souhaite aller, avec le moins de changement de la station Alésia à la station Argentine, en interdisant les lignes 5 et 7 :

The screenshot shows a window titled "Menu itinéraire métro". It contains the following elements:

- Two dropdown menus for origin and destination, with "Alésia" and "Argentine" selected respectively.
- A list box containing the following items: 1, 2, 3, 3 bis, 4, 5, 6, 7, 7 bis. Items 5 and 7 are highlighted in grey, indicating they are selected for prohibition.
- A dropdown menu labeled "Stations à interdire" (Stations to prohibit).
- Two buttons: "Ajouter" (Add) and "Supprimer" (Remove).
- A dropdown menu for route type, currently set to "Itinéraire avec le moins de changements" (Route with the fewest changes).
- A "Calcul itinéraire" (Calculate route) button.



Si l'on souhaite en plus interdire des stations, on clique sur le menu déroulant qui se nomme « Stations à interdire » et on sélectionne celle(s) que l'on souhaite :



The screenshot shows a window titled "Menu itinéraire métro" with standard window controls (minimize, maximize, close). Inside the window, there are two dropdown menus at the top. The first dropdown is set to "Alésia" and the second to "Argentine". Below these is a list box containing the following items: 1, 2, 3, 3 bis, 4, 5, 6, 7, and 7 bis. Items 5, 7, and 7 bis are highlighted with a grey background. At the bottom, there is a search input field with a blue border and a dropdown arrow. The dropdown menu is open, showing a list of station names: Abbesses, Alexandre Dumas, Alma Marceau, Alésia, Anatole France, and Anvers.

Menu itinéraire métro

Alésia

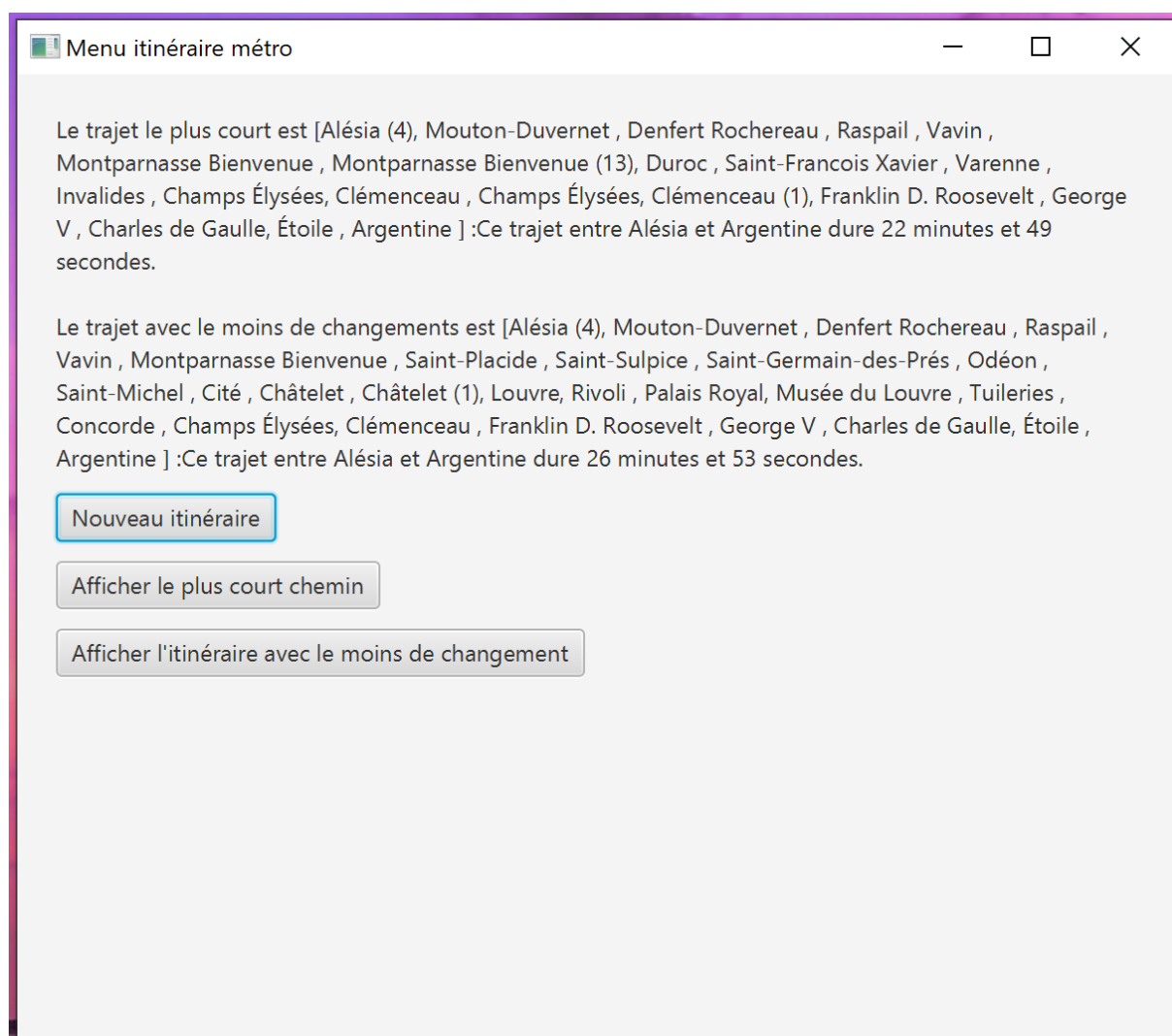
Argentine

1  
2  
3  
3 bis  
4  
5  
6  
7  
7 bis

|

Abbesses  
Alexandre Dumas  
Alma Marceau  
Alésia  
Anatole France  
Anvers

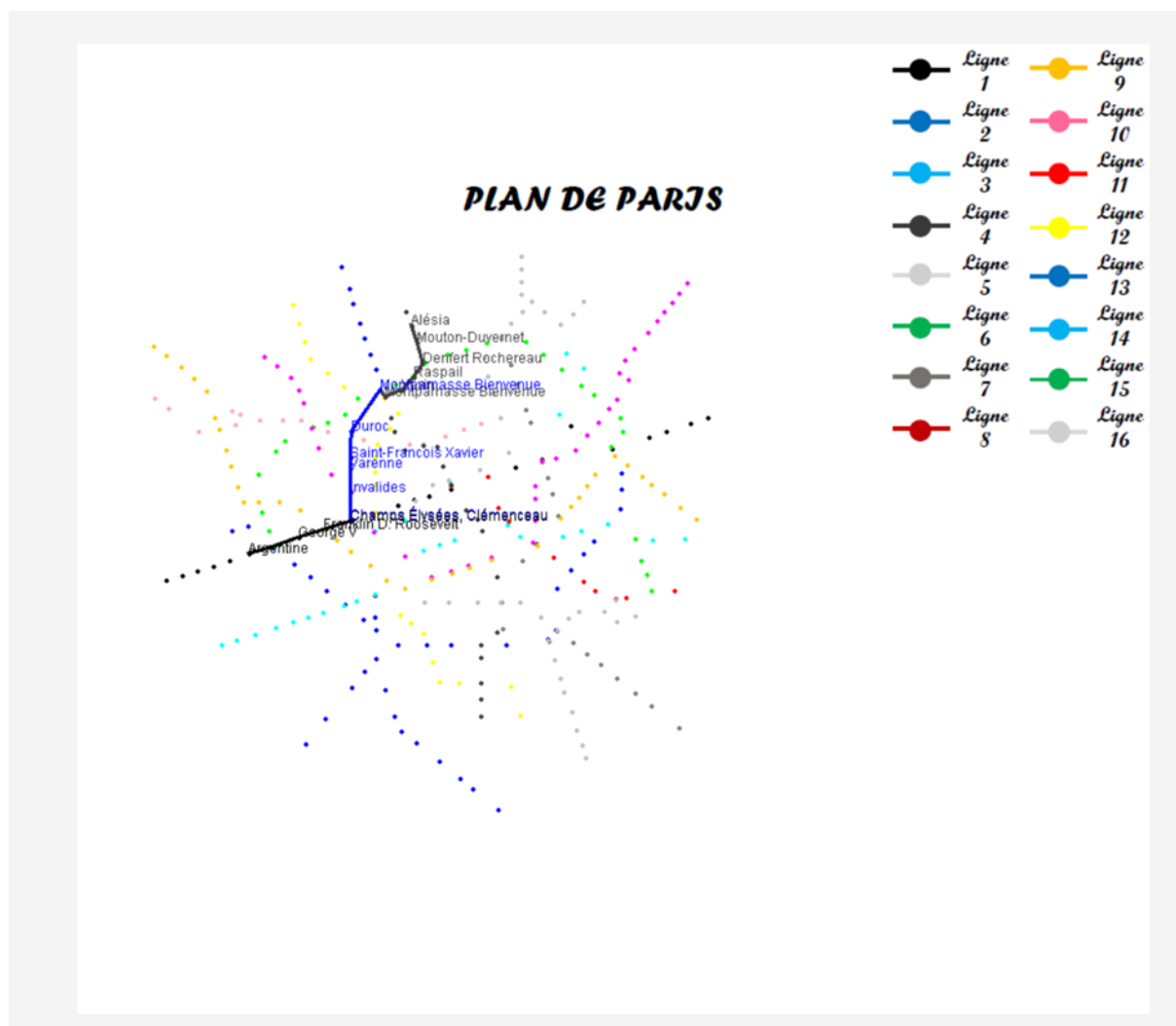
Une fenêtre de cette forme devrait alors s'afficher :



Comme vous pouvez le voir, votre (ou vos s'il y en a plusieurs) itinéraire sera affiché en haut. Le trajet sera affiché ainsi que le temps estimé de celui-ci.

3 boutons sont affichés à la suite. Le bouton nouvel itinéraire vous ramène à la fenêtre précédente et vous permet de chercher un nouvel itinéraire. Attention ! Si vous aviez interdit des stations, elles seront toujours interdites dans votre nouvel itinéraire. Vous pouvez les supprimer si vous le souhaitez en les sélectionnant dans le menu déroulant et en appuyant sur le bouton « supprimer »

Pour finir, vous pouvez afficher votre itinéraire sur une carte grâce aux deux derniers boutons :



## Conclusion :

Ce rapport décrit les principales étapes du développement d'un programme de calcul et d'affichage d'itinéraires dans le métro parisien et dans n'importe quel graphe.

La conception de ce programme a tourné autour de deux axes principaux : d'abord la partie calcul du plus court chemin, où nous avons appris les bases de la conception d'un graphe, de l'algorithme de Dijkstra; puis la partie graphique, où nous avons pu construire une interface claire et cohérente pour l'utilisateur grâce à JavaFX et l'image du chemin sur graphe rends l'utilisation plus ergonomique.

Bien sûr, ce programme est simplifié : nous ne prenons pas en compte les grèves, les retards... et les valeurs pour les temps de trajet et de changement de ligne sont des moyennes... Pourtant, malgré la simplicité apparente du programme, le développement de ce programme nous a permis de mieux la complexité et l'expérience que demande la programmation : allers retours constant entre les programmes, parfois la nécessité de réécrire entièrement des bouts de code, passer des heures à chercher une erreur dont la solution se trouve dans la modification d'une seule de code... Ainsi ce projet a clairement amélioré notre compréhension et nos compétences en programmation. C'est par la pratique qu'on progresse le plus vite. Le bilan global de ce projet est positif : nous éprouvons de la satisfaction à voir nos lignes de code « prendre vie » et nous aurions voulu aller plus loin, par exemple travailler sur l'auto-complétion des combobox, embellir l'interface utilisateur...