UVSQ université PARIS-SACLAY | ISTY Institut des Sciences et Techniques des Yvelines
CAMPUS DE MANTES EN YVELINES
CAMPUS DE SAINT-QUENTIN-EN-YVELINES

Projet individuel - GPS

VIDAL Antoine

Encadrant : MANOUSSAKIS George
Référent : PILARD Laurence

IATIC5 2022-2023
ISTY Vélizy, 10-12 Avenue de l'Europe, 78140 Vélizy-Villacoublay

Table des matières

Liste des figures

Liste des tableaux

Liste des codes

Introduction

Contexte

Dans l'objectif d'élargir mes connaissances et de devenir plus familier avec l'environnement informatique, j'ai décidé d'orienter mon parcours vers la filière ingénieur informatique de l'*ISTY* : *IATIC* (Ingénierie des Architectures Technologiques de l'Information et de la Communication). De ce fait j'ai donc suivi l'ensemble de la formation et des modules proposés par cette filière. L'un de ces modules s'intitule "Projet Individuel". Il consiste à réaliser un projet au long du semestre sous la tutelle d'un des professeurs de l'université. Ce projet est à effectuer en autonomie. Je me suis donc mis à la recherche d'un sujet et d'un tuteur.

Choix et établissement du sujet

Ce projet se fait sans contrainte et le sujet est au choix de l'étudiant. J'ai donc voulu m'orienter vers un projet utilisant un langage très populaire et utilisé en ce moment : le langage Python. La formation ingénieur forme des ingénieurs polyvalents mais n'est pas spécialisé dans ce langage. Il semble donc judicieux de se familiariser et se former avec ce langage qui de plus offre de nombreuses librairies et fonctionnalités. Après avoir choisi un langage de programmation, il a fallu définir un sujet. Mon optique a été d'essayer d'implémenter des algorithmes et des notions abordées lors de la formation d'ingénieur que j'ai suivie. Le choix s'est finalement porté sur l'algorithme de Dijkstra. Cet algorithme est très connu et donc par conséquent semble intéressant à implémenter. Cet algorithme permet de calculer le plus court chemin dans un graphe comportant des poids positifs. L'application la plus concrète de cet algorithme est dans l'utilisation d'un GPS : chaque sommet du graphe représente une ville et l'utilisateur se trouve sur un sommet (ville) et souhaite se rendre dans une autre ville. Le chemin optimal sera déterminé avec l'algorithme de Dijkstra. Implémenter un GPS m'a donc semblé un choix intéressant. Cela me permet de plus de me familiariser avec une interface graphique. J'ai donc finalement décidé d'opter pour ce projet. Concernant le choix du tuteur, il s'agit du professeur m'ayant enseigné cet algorithme. Il m'a donc semblé le plus pertinent de le choisir comme tuteur. Après avoir obtenu son accord, j'ai pu commencer à travailler sur mon projet individuel.

1 Élaboration du projet

1.1 Cadre et organisation du projet

Le projet se réalise donc en autonomie. Dans mon cas, le mardi était un créneau réservé à ce projet. Ce créneau permet si besoin de prendre contact avec le tuteur pour faire un point ou poser des questions. Le week-end a aussi été utilisé pour permettre d'avancer sur le projet autant que possible. Comme énoncé précédemment, le mois de septembre a été consacré à l'établissement du sujet et du tuteur ce qui signifie que le projet a commencé à la fin du mois de septembre. De plus, le matériel nécessaire pour réaliser le projet consiste simplement en une machine ayant accès à Internet, ce qui est mon cas : il n'y a donc pas eu besoin de se fournir en matériel pour ce projet.

1.2 Choix des outils et de la méthode de développement

Avant de démarrer un projet, il est important de déterminer quels sont les outils et méthodes adéquates pour élaborer celui-ci. Comme déclaré antérieurement, le langage choisi est du Python. Cependant il a fallu faire d'autres choix, par exemple pour l'interface graphique : choisir du Python tout seul permet uniquement ne permet pas de réaliser des programmes avec interface graphique. Or dans notre cas nous devons réaliser un GPS : il est beaucoup plus simple de savoir la position de l'utilisateur et sa destination à l'aide d'une interface graphique qu'avec du texte uniquement. Après avoir recherché les différentes possibilités en Python, il semble que le choix optimal se porte sur *tkinter*. Cette librairie permet de créer des interfaces graphiques simples et est disponible sur la plupart des systèmes d'exploitation. Après avoir opté pour cette librairie, il a fallu réfléchir à l'implémentation de nos graphes. Le choix s'est porté initialement sur l'utilisation des fonctionnalités de dessin de *tkinter* toutefois après s'être renseigné sur les graphes en Python deux outils se sont démarqués : *NetworkX* et *igraph*. Ces deux librairies facilitent la réalisation de graphes et évite donc de devoir dessiner soi-même les sommets, arêtes... La différence entre ces deux outils est que *NetworkX* préfère des noeuds (sommets) nommés à partir de labels (de chaînes de caractères) tandis que *igraph* opte pour des noeuds associés à des nombres. Dans notre cas, nos noeuds représentent des villes et donc il n'y a de relation numérique entre chaque ville. Le choix s'est donc porté sur *NetworkX*.

Nous avons donc jusqu'ici les outils : Python avec les librairies *tkinter* et *Networkx*. Il convient par la suite de décider de la façon de développer et de la structure du code développé. Le motif d'architecture logicielle Modèle-Vue-Contrôleur a été choisi en raison de sa simplicité et de son efficacité. Le modèle représente les données que l'utilisateur ne voit pas et gère donc tout la partie invisible du programme : par exemple les algorithmes de calcul... La vue à l'inverse représente la partie visible du programme : il s'agit donc essentiellement de l'interface graphique. Le contrôleur sert de passerelle entre la vue et le modèle et donc transfère les données entre ces deux modules. Toutefois le modèle et la vue ne sont pas supposés communiquer entre eux, puisque c'est le rôle du contrôleur.

Enfin le développement du code a été réalisé avec la plateforme *Github* et le logiciel de gestion de versions *Git*. Celui-ci permet de développer confortablement son code et de revenir à des versions antérieures si l'utilisateur le souhaite.

1.3 Exemple d'utilisation

Nous allons ici illustrer le résultat obtenu et voir comment se servir de l'outil réalisé. Tout d'abord au démarrage nous avons la fenêtre en Figure 1 :

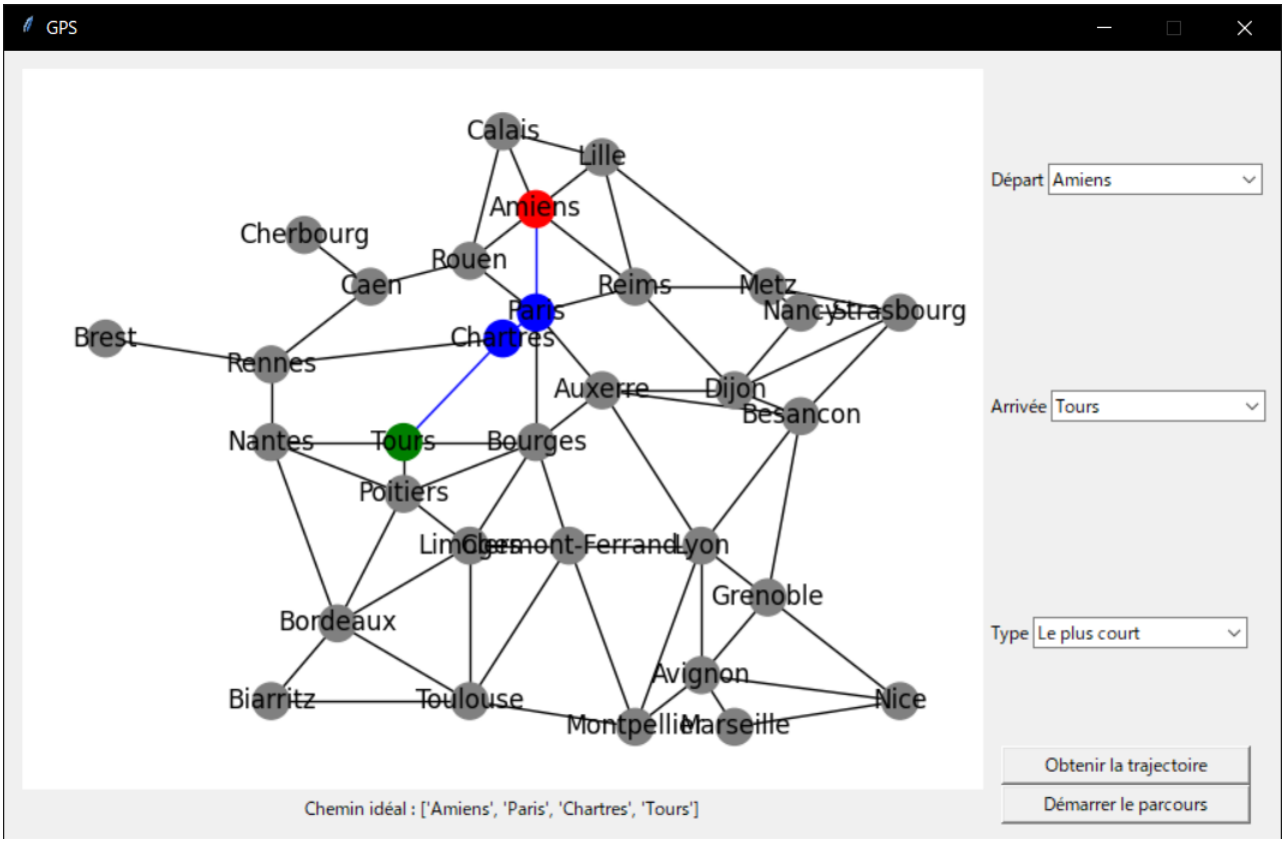

Figure 1: Exemple de choix de l'itinéraire

Nous avons ici deux parties principales : la partie de gauche qui comporte le graphe de la France ainsi que le chemin idéal sous forme de sommets colorés mais aussi sous forme de texte. La partie de droite permet de choisir la ville de départ et la ville d'arrivée ainsi que le type de trajet à déterminer. L'application en comporte deux : le chemin le plus court en fonction du poids de chaque arête et le chemin avec le moins de villes parcourues. La deuxième option peut être utile lorsque par exemple l'utilisateur ne souhaite pas prendre le chemin optimal mais le plus simple. Le bouton "Obtenir la trajectoire" permet d'obtenir le parcours supposé idéal sans démarrer le trajet. Ceci est le rôle du bouton "Démarrer le parcours". Cela permet de voir et vérifier que le parcours que l'utilisateur va effectuer est bien celui qu'il souhaite.

Une fois le chemin décidé, l'utilisateur peut commencer son trajet. Un exemple est visible en Figure 2. La ville de départ est en rouge, la ville d'arrivée en vert, la ville actuelle en jaune et en bleu nous trouvons les villes composant le chemin idéal. Ici le choix de l'utilisateur consiste en la ville de destination. Il doit choisir parmi les voisins de la ville actuelle, à quelle ville il souhaite se rendre. Il suffit ensuite de cliquer sur le bouton "Se déplacer", pour se rendre à la ville sélectionnée. Une fois la ville de destination atteinte, un message est indiqué à l'utilisateur lui indiquant son arrivée et l'application revient à l'écran précédent (permettant de choisir un départ et une destination avec le type de trajet à déterminer).

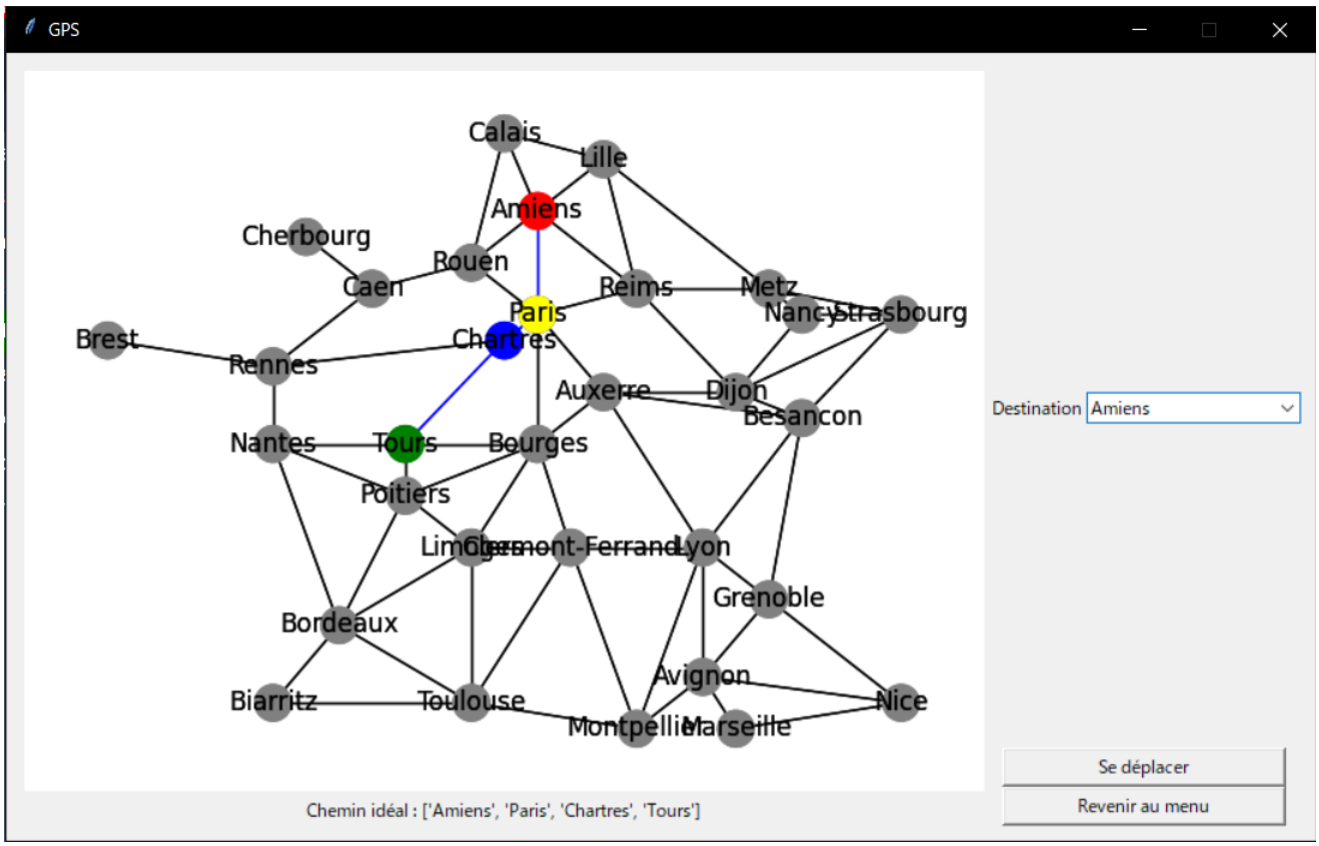

Figure 2: Exemple de déplacement avec le GPS

Si l'utilisateur se trompe et prend une destination ne se trouvant pas dans le chemin idéal, l'application lui informe du problème et détermine un nouvel itinéraire à partir de la ville actuelle mais en ne changeant pas la ville de destination initiale.

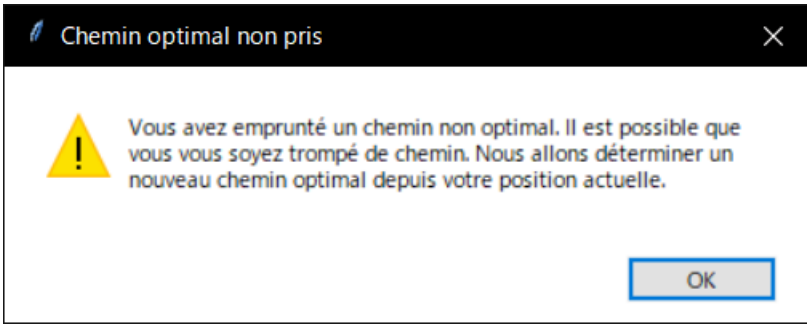

Figure 3: Message d'avertissement lorsque le chemin optimal n'est pas emprunté

2 Implémentation des fichiers hors Python

Avant de voir et d'étudier comment notre programme a été organisée, il convient de découvrir comment le programme Python interagit avec des données en dehors de celui-ci. En effet, mettre l'intégralité des données dans le code Python rend le programme moins modulable et plus compliqué à appréhender pour un utilisateur n'ayant pas de connaissance en Python.

2.1 Carte de la France

Il a tout d'abord fallu obtenir notre carte de la France. En effet, même si l'objectif est de faire un GPS permettant de se déplacer sur la France entière, il ne sera pas possible d'implémenter toutes les villes de la France sous peine d'avoir une carte surchargée. Nous avons décidé de nous limiter à une carte de la France avec les villes principales et les plus peuplées de celles-ci. Après avoir effectué plusieurs recherches, il semble que la carte qui représente le mieux mes objectifs soit trouvable sur ce lien.

Toutefois il reste des modifications à effectuer : il reste toujours des villes à retirer. Le choix sur les villes à retirer a été arbitraire, s'agissant principalement de villes adjacentes à de plus grandes villes. Une fois les villes obtenues et listées, il a fallu réfléchir à la manière de les afficher sur notre application. En effet, sans option de positionnement, les villes se retrouvent dans une position quelconque et donc ne refléteront pas la géographie de la France. L'avantage de la carte utilisée est qu'elle présente un quadrillage, ceci nous permet donc d'établir des relations de distance entre les différentes villes. L'origine du repère se situant en bas à gauche de l'image, il est possible de définir une position pour chaque ville. Enfin il ne nous reste plus qu'à déterminer les différentes arêtes reliant l'ensemble des villes. Celles-ci ont été aussi choisies arbitrairement : leur poids correspond globalement au nombre de carreaux reliant deux villes et de plus toutes les villes ne sont pas reliées entre elles pour éviter un graphe surchargé visuellement. Pour avoir un aperçu des modifications effectuées, nous avons inséré en Figure 4 et 5 la carte initiale et la carte sur laquelle notre application se basera.

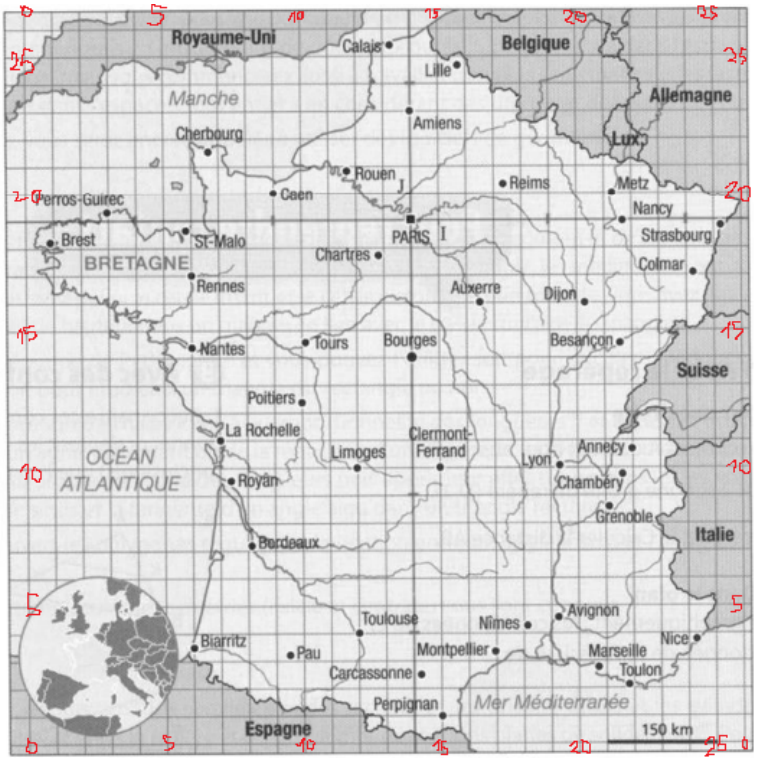

Figure 4: Carte originale de la France sur laquelle nous nous sommes basés

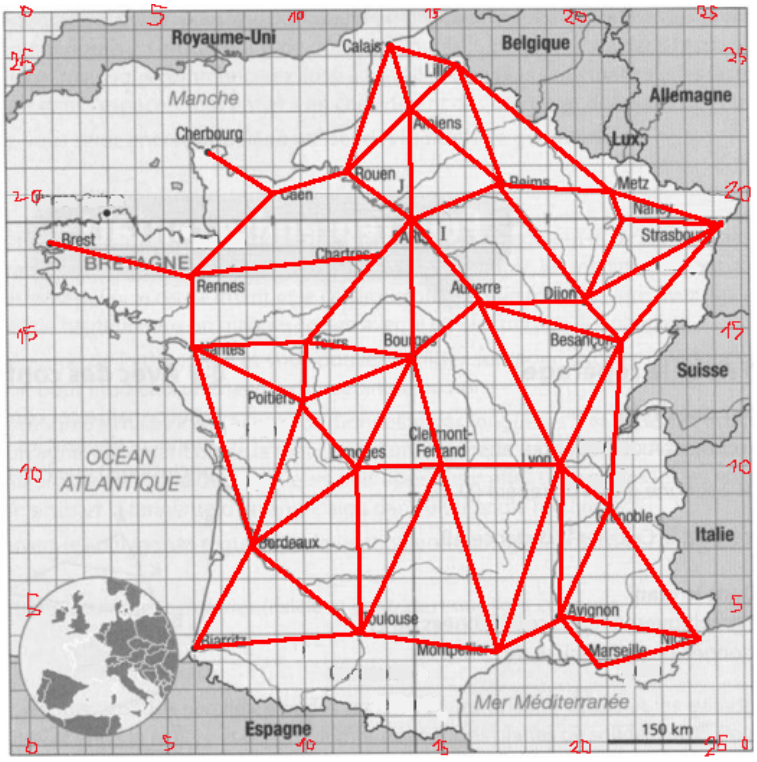

Figure 5: Carte modifiée de la France servant de base pour notre application

2.2 Format des fichiers

Nous avons donc notre image représentant notre graphe. Toutefois nous devons implémenter cette image dans un format que Python peut comprendre et utiliser. C'est pourquoi nous allons expliquer les différentes structures et formats utilisés pour accomplir cette tâche.

2.2.1 Position des villes

En ce qui concerne la position des villes, nous avons énoncé précédemment que celles-ci étaient définies par un couple de coordonnées (x, y). Utiliser un tableau semble donc judicieux. Cependant, stocker le tableau dans le programme Python pertinente : cela rend notre tableau moins modulable et oblige l'utilisateur à ouvrir un fichier Python pour modifier à sa guise les villes. Par conséquent il a été décidé de stocker ce tableau dans un fichier texte. Il ne s'agit en revanche pas d'un fichier texte classique mais d'un fichier csv. Un fichier .csv représente un fichier texte où les valeurs sont séparées par des virgules. Ceci est donc équivalent à un tableau puisque qu'une colonne de notre tableau est séparée entre deux virgules. Une nouvelle ligne dans notre fichier .csv représente en revanche une nouvelle rangée dans notre tableau. Nos colonnes sont ici au nombre de trois : le nom de la ville, l'abscisse et l'ordonnée. Une rangée représente ici une ville avec chaque ville qui est listée par ordre alphabétique. Nous obtenons ici un tableau semblable à la Figure 1 (nous représentons ici un tableau mais le fichier .csv contient bien des virgules, le tableau permet de mieux visualiser).

2.2.2 Matrice d'adjacence pour les arêtes

Nous avons implémenté nos sommets. Par la suite nous devons ajouter les arêtes qui relient les différentes villes. Dans notre application les arêtes représentent la jonction entre deux villes : des poids sont attribués à chaque arête avec comme poids le nombre de carreaux présent entre deux villes. Pour stocker les arêtes, nous allons procéder comme pour les positions des villes et les stocker dans un fichier texte. Deux options ont ensuite été considérées : la première a été d'utiliser des listes d'adjacence avec chaque ligne représentant une ville et ses voisins (ainsi que le poids permettant de se rendre à la ville). Toutefois cette option a semblé être moins modulable et plus compliqué à implémenter que la seconde proposition. Elle consiste à réaliser une matrice d'adjacence représentant donc le graphe mais sous forme de tableau. Cette matrice est symétrique puisque notre graphe n'est pas orienté (il est possible de faire le trajet Paris-Bourges mais aussi Bourges-Paris). De manière analogue à la position des villes, nous avons également une ligne représentant une ville (celles-ci étant également triées par ordre alphabétique). Cependant c'est aussi le cas pour les colonnes : une colonne représente une ville (toujours triées par ordre alphabétique). Une arête se représente donc par la case du tableau se trouvant à la ligne et la colonne des villes correspondantes. Les règles suivantes ont été appliquées : une valeur de 0 représente une absence d'arête tandis qu'une valeur positive entière représente le poids de l'arête entre deux villes. Ainsi par exemple si à la ligne "Paris" et à la colonne "Bourges" nous avons une valeur de 2, cela signifie qu'entre Paris et Bourges nous avons une arête de poids 2. Puisque la matrice d'adjacence est symétrique, c'est aussi le cas pour la colonne "Paris" et la ligne "Bourges". Il ne reste donc plus qu'à remplir les cases adéquates dans le tableau pour obtenir notre matrice d'adjacence. Un exemple de matrice d'adjacence non représentatif de la matrice final est visible en Figure 2 (la matrice complète étant visible en Annexe).

Ville	x (abscisse)	y (ordonnée)
Amiens	14	23
Auxerre	16	16
Avignon	19	5
Besancon	22	15
Biarritz	6	4
Bordeaux	8	7
Bourges	14	14
Brest	1	18
Caen	9	20
Calais	13	26
Chartres	13	18
Cherbourg	7	22
Clermont-Ferrand	15	10
Dijon	20	16
Grenoble	21	8
Lille	16	25
Limoges	12	10
Lyon	19	10
Marseille	20	3
Metz	21	20
Montpellier	17	3
Nancy	22	19
Nantes	6	14
Nice	25	4
Paris	14	19
Poitiers	10	12
Reims	17	20
Rennes	6	17
Rouen	12	21
Strasbourg	25	19
Toulouse	12	4
Tours	10	14

Table 1: Fichier .csv comportant les positions des villes

Tableau	Ville 1	Ville 2	Ville 3	Ville 4	Ville 5	Ville 6	Ville 7	Ville 8
Ville 1	0	1	0	0	0	0	2	0
Ville 2	1	0	0	2	0	0	0	0
Ville 3	0	0	0	0	0	4	4	0
Ville 4	0	2	0	0	5	0	0	3
Ville 5	0	0	0	5	0	0	0	0
Ville 6	0	0	4	0	0	0	0	0
Ville 7	2	0	4	0	0	0	0	0
Ville 8	0	0	0	3	0	0	0	0

Table 2: Exemple de fichier .csv comportant les arêtes entre les villes

Par conséquent, après avoir élaboré nos fichiers, il ne nous reste plus qu'à lire nos fichiers pour pouvoir tracer notre graphe sur Python. Cette partie sera expliquée ultérieurement mais nous obtenons le résultat visible en Figure 6 une fois les instructions effectuées.

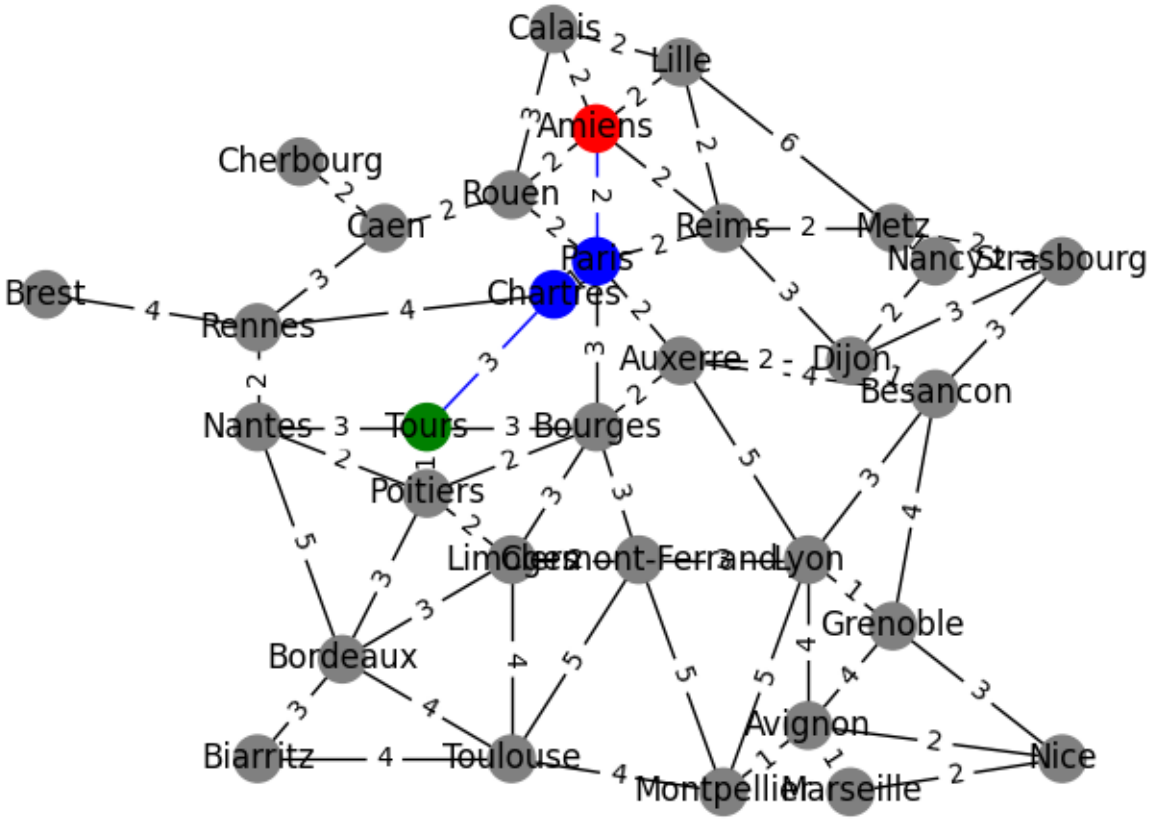

Figure 6: Graphe de la France utilisée par notre application. Ici les poids des arêtes sont indiquées à titre indicatif et ne sont pas visibles lors d'une utilisation normale du GPS.

3 Structures des codes Python

Nous avons donc tous les outils pour créer notre GPS. Il ne reste plus qu'à programmer les différentes fonctionnalités en se basant sur le travail de la section précédente. Comme abordée auparavant, notre application se base sur l'architecture "Modèle-Vue-Contrôleur". Nous allons donc analyser chacun de ces modules pour voir leurs relations et interactions.

3.1 Main

Le fichier main comme dans beaucoup de langages de programmation représente le fichier principal de notre application. C'est celui-ci qui instance les trois modules "Modèle-Vue-Contrôleur". Mis à part ces initialisations, il y a eu une particularité à ajouter. Le bouton "Fermer la fenêtre" de l'application. En effet, de base l'interface graphique utilisé (*tkinter*), ferme la fenêtre lorsque l'on appuie sur le bouton en forme de croix, néanmoins cela ne termine pas le processus qui tourne (représentant notre application). Il a donc fallu ajouter cette fonctionnalité. Celle-ci est visible en Listing 1. Lorsque l'utilisateur clique sur le bouton, une fenêtre d'avertissement apparaît pour demander si l'utilisateur souhaite bien quitter l'application. Si la réponse est oui, nous utilisons la méthode `destroy()` de *tkinter* pour mettre fin à l'application.

```
1 def quitter():
2     if messagebox.askyesno("Quitter", "Voulez-vous quitter?"):
3         app.destroy()
4
5 app.protocol("WM_DELETE_WINDOW", quitter)
```

Listing 1: Fonction permettant de mettre fin à l'application

3.2 Modèle

Le modèle représente la partie non visible par l'utilisateur. Dans notre application, cela représente par exemple les matrices d'adjacences. Or nous avons vu précédemment comment celles-ci étaient implémentées. Il ne reste plus qu'à lire les fichiers .csv pour obtenir notre graphe. Les algorithmes réalisant ces tâches sont visibles en Listing 2 et 3.

```
1 df_towns_coords = pd.read_csv("towns_coords.csv")
2 for row in df_towns_coords.itertuples():
3     controller.Controller.G.add_node(row.town, pos=(row.x, row.y))
4 controller.Controller.pos=nx.get_node_attributes(controller.Controller.G,"pos")
5 controller.Controller.list_towns_text = list(controller.Controller.G.nodes)
```

Listing 2: Algorithme de lecture des coordonnées des villes

Après avoir lu le fichier, il suffit d'itérer la lecture ligne par ligne. Pour rappel, une ligne se compose du nom de la ville, de son abscisse et de son ordonnée. Nous devons donc simplement lire ces trois données et les ajouter au graphe à chaque fois (à l'aide de la méthode `add_node()` de *Networkx*. Enfin nous stockons une liste des villes sous forme de texte pour pouvoir y avoir accès facilement (par exemple pour la Vue).

```
df_towns_edges = pd.read_csv("towns_edges_2d_array.csv")
df_towns_edges = df_towns_edges.set_index("Town")
for rowIndex, row in df_towns_edges.iterrows():
    for columnIndex, value in row.items():
        if value > 0:
            controller.Controller.G.add_edge(rowIndex, columnIndex, weight=value)
```

Listing 3: Algorithme de lecture des arêtes reliant les villes

Le principe est ici similaire. Nous parcourons le fichier ligne par ligne avec pour entête et index la liste des villes du fichier. Ici la condition pour ajouter une arête consiste à lire un chiffre strictement positif. En effet, nous avons établi précédemment qu'une valeur de 0 signifiait une absence d'arête entre deux villes. Si la valeur est strictement positive, alors nous ajoutons une arête entre les villes correspondant aux lignes et colonnes de cette cellule. La matrice d'adjacence est donc ainsi construite.

Il reste une dernière partie faisant partie du modèle : le calcul du plus court chemin. En effet, l'utilisateur n'a nullement besoin de connaître comment ce chemin est calculé tant qu'à la fin l'application lui indique le chemin idéal. Le procédé utilise se trouve en Listing 4.

```
def compute_shortest_path(self, start, arrival, type_path):
    controller.Controller.start_town = start
    controller.Controller.arrival_town = arrival
    controller.Controller.shortest_path_type_value = type_path
    if controller.Controller.shortest_path_type_value ==
    ↪ controller.Controller.shortest_path_type_list[0]:
        controller.Controller.shortest_path = nx.dijkstra_path(controller.Controller.G, start,
        ↪ arrival)
    elif controller.Controller.shortest_path_type_value ==
    ↪ controller.Controller.shortest_path_type_list[1]:
        controller.Controller.shortest_path = nx.shortest_path(controller.Controller.G, start,
        ↪ arrival)
    controller.Controller.shortest_path_edges =
    ↪ [(controller.Controller.shortest_path[i],controller.Controller.shortest_path[i+1]) for
    ↪ i in range(len(controller.Controller.shortest_path)-1)]
```

Listing 4: Calcul du plus court chemin à partir de notre graphe de la France

Les paramètres ici sont la ville de départ, la ville d'arrivée et le type de chemin. Après avoir récupéré ces paramètres, il suffit de regarder la valeur de la variable `shortest_path_type_value`. Nous parcourons la liste des types de chemins disponibles et si cette variable est égale à la valeur d'un des éléments de la liste alors nous pouvons calculer notre plus court chemin. Enfin nous ajoutons la liste des sommets à parcourir dans une liste : cette liste représente le chemin idéal. Les fonctions de calculs de plus court chemin étant déjà implémentées dans *Networkx*, il suffit simplement de regarder comment ces méthodes fonctionnent pour les utiliser (par exemple `nx.dijkstra_path` prend en paramètre le graphe, une ville de départ et une ville d'arrivée). Ici nous n'avons que deux choix de plus courts chemins : "Le plus court en fonction des poids des arêtes" et "Le moins de villes parcourues".

3.3 Vue

La vue représente donc toute la partie visible par l'utilisateur. Comme énoncé antérieurement, notre fenêtre possède deux parties distinctes : la partie gauche représentant le graphe et le chemin idéal et la partie droite représentant les actions de l'utilisateur.

3.3.1 Partie droite de l'écran : actions de l'utilisateur

Dans cette partie droite de l'écran, nous avons deux principaux modules : celui du choix de l'itinéraire à parcourir et celui du déplacement vers les différentes villes.

Voyons d'abord comment fonctionne le choix de l'itinéraire. Celui-ci visuellement consiste en une liste de boîtes déroulantes où l'utilisateur doit choisir une valeur dans ces boîtes (le départ, l'arrivée et le type de chemin). Regardons comment une boîte déroulante est réalisée. Le procédé étant similaire pour chaque boîte seulement une seule boîte sera évoquée. Le Listing 5 permet d'effectuer cette tâche.

```
1 frm_town_start = Frame(master=self.frm_ui_choose_route)
2 lbl_town_start = Label(master=self.frm_town_start, text="Départ")
3 cmb_town_start = Combobox(master=self.frm_town_start, state="readonly",
  ↪ values=controller.Controller.list_towns_text)
4 cmb_town_start.set(controller.Controller.list_towns_text[0])
5
6 frm_town_start.pack(fill=X, expand=True)
7 lbl_town_start.grid(row=0, column=0)
8 cmb_town_start.grid(row=0, column=1)
```

Listing 5: Commandes permettant de construire une boîte déroulante avec *tkinter*

Nous devons tout d'abord choisir un parent, c'est-à-dire sur quel cadre la boîte sera affichée. Ici il s'agit de la partie droite de notre application. Après avoir créer un `Label` permettant d'indiquer à l'utilisateur le rôle de la boîte déroulante, nous pouvons nous attarder sur la partie intéressante : celle-ci se trouve au niveau des valeurs à indiquer à la boîte déroulante. Ici nous souhaitons que l'utilisateur puisse choisir entre la liste des villes disponibles. Il suffit donc de récupérer la variable `list_towns_text` que nous avons défini précédemment. Enfin un choix arbitraire a été effectué : la valeur par défaut de la boîte déroulante. Par exemple pour la boîte déroulante représentant la ville de départ il s'agit de la première valeur de la liste (puisque les villes sont triées par ordre alphabétique, il s'agit donc de la première ville). Enfin il ne reste plus qu'à afficher les modules (à l'aide des méthodes `pack() et grid()` sous *tkinter*). La boîte déroulante représentant la valeur d'arrivée est presque identique puisque seulement le `Label` diffère. Pour le type de chemin les valeurs de la boîte déroulante sont différentes et comporte uniquement les deux choix abordés auparavant.

Une fois les boîtes déroulantes construites, il faut que l'utilisateur puisse valider ses choix. Le moyen le plus approprié pour cela est de fabriquer des `Button`. Ce sont les deux boutons "Obtenir la trajectoire" et "Démarrer le parcours" qui occupent ces rôles. Leur fonctionnement est visible en Listing 6.

```
self.btn_confirm = Button(self.frm_ui_choose_route,text="Obtenir la trajectoire",
↪ command=lambda:controller.Controller.get_shortest_path(controller.Controller,
↪ self.cmb_town_start.get(), self.cmb_town_arrival.get(), self.cmb_shortest_path_type.get()))
self.btn_start_path = Button(self.frm_ui_choose_route,text="Démarrer le parcours",
↪ command=lambda:View.start_path(self))
```

Listing 6: Commandes permettant de construire les boutons permettant de démarrer une trajectoire choisie

Nous avons ici deux boutons. Le premier permet donc d'obtenir la trajectoire idéale. La fonction `get_shortest_path()` provient du Contrôleur et donc sera abordée ultérieurement. Toutefois, en exécutant cette fonction il devient donc possible d'obtenir les sommets représentants le chemin que l'utilisateur devrait emprunter. Le second bouton enclenche donc le deuxième module de cette partie droite de notre application : le démarrage du parcours et donc le déplacement vers une ville au choix.

L'utilisateur doit donc choisir une ville voisine à sa ville actuelle. Pour cela nous recréons une boîte déroulante qui fonctionne de manière similaire à l'exception des valeurs de celles-ci. Ici l'utilisateur ne peut uniquement se rendre à une ville voisine à sa ville actuelle (autrement dit une ville voisine avec laquelle une arête est présente). Il convient donc de créer une variable permettant d'avoir accès à ses voisins. *Networkx* possède justement une méthode à cet effet : `list = G.neighbors(current_town)`. Cette méthode permet à partir d'un graphe G et d'une ville (en paramètre) de stocker ses voisins dans une liste. C'est ce que nous avons donc effectué sur la variable `neighbors_current_town`. Cette variable contiendra à tout instant les sommets voisins de la ville actuelle. À chaque que l'utilisateur arrive à une ville, cette liste est mise à jour. Nous avons donc par conséquent une boîte déroulante avec dedans les voisins disponibles. Une fois que l'utilisateur a fait son choix, l'application doit donc le prendre en compte et se rendre à la ville sélectionnée. Nous répétons ainsi ce processus jusqu'à ce que l'utilisateur arrive à destination. Le Listing 7 illustre cette démarche.

```
if(controller.Controller.current_town == controller.Controller.arrival_town):
    messagebox.showinfo("Information", "Vous êtes arrivé.")
    controller.Controller.shortest_path.clear()
    View.lbl_best_path.configure(text="Chemin idéal :
    ↪ {}".format(controller.Controller.shortest_path))
    View.frm_ui_moving.pack_forget()
    View.frm_ui_choose_route.pack(fill=Y, side=RIGHT, expand=True)
```

Listing 7: Commandes exécutées lorsque l'utilisateur arrive à destination

Pour vérifier que l'utilisateur est bien arrivé à destination, il suffit de comparer la ville actuelle et la ville de destination. Si celles-ci sont égales, alors l'utilisateur est arrivé. Un boîte dialogue lui informe de ce fait et il convient donc de réinitialiser les différentes variables utilisées au long du parcours. Nous comptons parmi celles-ci le chemin optimal et toute la partie droite de notre application. En effet, dans un vrai GPS, lorsque l'utilisateur arrive à destination, l'application revient au choix de l'itinéraire idéal. C'est ce que nous effectuons ici également (avec la méthode `pack_forget()`).

Enfin il y a un dernier problème à résoudre lors du parcours à effectuer. L'utilisateur peut se tromper de ville et donc ne pas suivre le chemin optimal. Dans un vrai GPS, un nouvel itinéraire est calculé à partir de la nouvelle position actuelle mais en gardant la destination identique. Ceci se réalise comme indiqué sur le Listing 8. Un message d'avertissement prévient l'utilisateur du problème mais celui-ci est résolu en appelant la méthode `get_shortest_path()` (qui nous rappelons sera abordé en même temps que le Contrôleur) mais en changeant simplement le paramètre représentant la ville actuelle. Celle-ci était initialement la ville de départ de l'utilisateur mais devient donc la ville où celui-ci se trouve.

```
1 elif(controller.Controller.current_town not in controller.Controller.shortest_path):
2     messagebox.showwarning("Chemin optimal non pris", "Vous avez emprunté un chemin non
      ↪  optimal. Il est possible que vous vous soyez trompé de chemin. Nous allons déterminer
      ↪  un nouveau chemin optimal depuis votre position actuelle.")
3     controller.Controller.get_shortest_path(controller.Controller,
      ↪  controller.Controller.current_town, controller.Controller.arrival_town,
      ↪  controller.Controller.shortest_path_type_value)
```

Listing 8: Commandes exécutées lorsque l'utilisateur se trompe d'itinéraire

3.3.2 Partie gauche de l'écran : affichage du graphe et du chemin idéal

La partie gauche de l'écran représente donc le graphe de la France et consiste simplement à afficher la position de l'utilisateur et le chemin qu'il doit emprunter. Cette partie de l'application doit être mise à jour à chaque fois que l'utilisateur choisit un trajet mais également lorsqu'il se déplace vers une ville adjacente. Le Contrôleur se charge d'effectuer les modifications sur les variables permettant de mettre à jour l'image. La Vue se charge quant à elle de mettre à jour l'image. Cette image se situe à la racine du projet est consiste en une simple image au format .png. *Networkx* utilise *Matplotlib* pour l'affichage de ses graphes, nous utilisons donc les méthodes propres à cette librairie pour réaliser nos mises à jour. Celles-ci sont réalisées dans le Listing 9. À chaque fois que le Contrôleur modifie l'image, la Vue s'occupe de recharger à nouveau l'image et de la réafficher.

```
1 def update_image_France():
2     View.img_France = PIL.ImageTk.PhotoImage(PIL.Image.open("france_graphe.png"))
3     View.lbl_France.configure(image=View.img_France)
```

Listing 9: Commandes exécutées lorsque l'image est mise à jour (dans le module de la Vue)

3.4 Contrôleur

Le Contrôleur se charge donc de faire la liaison entre la Vue et le Modèle. Lorsque la Vue ou le Modèle doivent se transmettre des données, celles-ci font appel au Contrôleur pour y arriver. Notre Contrôleur est constitué principalement de deux fonctions : `get_shortest_path()` déjà abordée précédemment et `draw_graph_France()` qui s'occupe de modifier le graphe de façon adéquate aux actions de l'utilisateur pour ensuite laisser la vue afficher le graphe modifié. Cette première fonction est décrite dans le Listing 10.

```
def get_shortest_path(self, start, arrival, type_path):
    if(start == arrival):
        messagebox.showwarning("Attention", "Le départ et la destination sont identiques.")
    else:
        Controller.current_town = start
        model.Model.compute_shortest_path(model, start, arrival, type_path)
        Controller.draw_graph_France()
```

Listing 10: Commandes exécutées pour obtenir le chemin idéal

La seule exception à prendre en compte ici consiste à avoir un départ et une destination différente (puisque si ces deux variables sont égales le plus court chemin est déjà trouvé). Un message d'avertissement est envoyé si cela se produit. En revanche si le départ et la destination sont bien différents, il devient alors possible de déterminer un chemin idéal. Nous utilisons pour cela la méthode du Modèle `compute_shortest_path()` pour obtenir ce chemin. Il convient ensuite de transmettre à la vue ce chemin. Puisque la Vue et le Modèle ne peuvent pas communiquer entre eux, c'est au Contrôleur de le faire. Ceci se réalise avec le code présent en Listing 11.

```
def draw_graph_France():
    color_map = ["gray"]*len(Controller.G.nodes())
    for i, node in enumerate(Controller.G.nodes()):
        if node == Controller.start_town:
            color_map[i] = "red"
        elif node == Controller.arrival_town:
            color_map[i] = "green"
        elif node == Controller.current_town:
            color_map[i] = "yellow"
        elif node in Controller.shortest_path:
            color_map[i] = "blue"

    edge_color_list = ["black"]*len(Controller.G.edges())
    for i, edge in enumerate(Controller.G.edges()):
        if edge in Controller.shortest_path_edges or (edge[1],edge[0]) in
        ↪ Controller.shortest_path_edges:
            edge_color_list[i] = "blue"

    view.View.lbl_best_path.configure(text="Chemin idéal :
    ↪ {}".format(Controller.shortest_path))
    nx.draw(Controller.G, Controller.pos, with_labels=True, node_color=color_map,
    ↪ edge_color=edge_color_list)
    plt.savefig("france_graphe.png")
    view.View.update_image_France()
```

Listing 11: Commandes exécutées pour mettre à jour le graphe de la France

Le principe consiste à affecter une couleur pour chaque noeud de notre graphe. Comme nous avons établi précédemment, nous avons ici cinq couleurs : gris pour les villes normales, rouge pour la ville de

départ, vert pour la ville d'arrivée, jaune pour la ville actuelle, et bleu pour une ville appartenant au chemin idéal. Nous parcourons donc la liste des noeuds et affectons la couleur correspondante en fonction de son attribut. Pour les arêtes un procédé similaire est exécuté. Nous affectons la couleur noire pour chaque arête de base et si celle-ci fait partie du chemin à emprunter, alors la couleur de cette arête devient bleue. Après ces affectations, nous avons donc notre graphe mis à jour. Il ne reste plus qu'à informer la Vue de ces changements : pour cela nous modifions le `Label` "Chemin idéal" puis nous sauvegardons le graphe dans une image au format .png. La Vue n'a plus qu'à ouvrir la nouvelle image comme nous l'avons évoqué antérieurement.

Conclusion

Dans le cadre de mes études en *IATIC5*, j'ai donc dû effectuer un projet individuel sur le sujet de mon choix. Cette expérience a été très enrichissante. J'ai pu me familiariser avec un langage très populaire en ce moment, le langage Python. Maîtriser ce langage peut être un atout majeur dans le monde d'aujourd'hui. J'ai également pu découvrir de nouvelles libraires, principalement *tkinter* et *Networkx*. Se familiariser avec de nouveaux outils et toujours intéressant et instructif.

Concernant le projet, je suis satisfait du résultat obtenu. Il ressemble à l'idée de base que je m'étais faite lors de la conception de celui-ci. Le projet possède les fonctionnalités prévues et permet même de choisir le chemin parcourant le moins de villes possibles. Toutefois certaines améliorations restent possibles. Par exemple en plus d'avoir un poids sur une arête représentant le temps de trajet, nous pouvons également ajouter un autre poids comme par exemple le prix du trajet. Toutefois *Networkx* ne propose qu'un seul poids par graphe, ce qui signifie qu'il faudrait implémenter des graphes supplémentaires en plus du graphe de base. Nous pouvons également ajouter d'autres moyens de transports, le seul étant implémenté n'ayant pas de nom (nous pouvons supposer qu'il s'agisse d'une voiture), il est tout à fait possible d'ajouter d'autres moyens de locomotion. Mais cela pose le même problème et donc revient à se munir d'autres graphes.

C'est sur ce type de projets que je souhaite m'orienter. Développer des applications tout en apprenant de nouveaux outils. Je trouve cela pertinent et enrichissant de réaliser des projets tout en se familiarisant avec des langages ou libraires inédites. Ceci permet d'élargir ses connaissances et donc d'être plus polyvalent.

Annexes

Description initiale du projet

À partir d'une carte de la France, l'objectif est de construire un GPS : à partir d'une position intiale, l'utilisateur souhaite se rendre à une ville. Toutefois, le principe consiste à choisir le trajet qui convient le mieux à l'utilisateur : ici cela sera le plus court chemin. Si le temps le permet, d'autres options seront considérées : le nombre de villes parcourues ou le trajet le moins cher (en affectant à chaque arête un prix)

La France sera représenté ici sous forme de graphe / tableau contenu dans un fichier csv qui représentera la matrice d'adjacence (préalablement remplie). L'entièreté de la France ne sera pas représenté pour des contraintes de stockage, nous nous limiterons donc à quelques dizaines de villes.

L'algorithme appliqué pour trouver le plus court chemin sera l'algorithme de Dijkstra.

À chaque ville atteinte (dans notre graphe un sommet), il sera possible de choisir la prochaine destination. Si celle-ci ne correspond pas à celle indiquée par le GPS, le GPS devra recalculer le plus court chemin.

Le tout devra être implémenté d'une interface graphique pour rendre l'outil plus simple d'utilisation. Le langage de programmation n'a pas encore été choisi mais sera du C ou du Python.

Matrice d'adjacence complète de notre carte de France

Town	Amiens	Auxerre	Avignon	Besancon	Biarritz	Bordeaux	Bourges	Brest	Caen	Calais	Chartres	Cherbourg	Clermont-	Dijon	Grenoble	Lille	Limoges	Lyon	Marseille	Metz	Montpellier	Nancy	Nantes	Nice	Paris	Poitiers	Reims	Rennes	Rouen	Strasbourg	Toulouse	Tours
Amiens	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	2	0	0	0	0	0	0	0	0	2	0	2	0	2	0	0	0
Auxerre	0	0	0	4	0	0	2	0	0	0	0	0	0	2	0	0	0	5	0	0	0	0	0	0	2	0	0	0	0	0	0	0
Avignon	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0	0	4	1	0	1	0	0	0	2	0	0	0	0	0	0	0
Besancon	0	4	0	0	0	0	0	0	0	0	0	0	0	1	4	0	0	3	0	0	0	0	0	0	0	0	0	0	0	3	0	0
Biarritz	0	0	0	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0
Bordeaux	0	0	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0	3	0	0	0	0	5	0	3	0	0	0	0	0	4	0
Bourges	0	2	0	0	0	0	0	0	0	0	0	0	3	0	0	0	0	3	0	0	0	0	0	0	3	2	0	0	0	0	0	3
Brest	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0	0	0	0
Caen	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	2	0	0	0
Calais	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	3	0	0	0
Chartres	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	4	0	0	3
Cherbourg	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Clermont-Fe	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	3	0	0	5	0	0	0	0	0	0	0	0	5	0
Dijon	0	2	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	3	0	0	0	0
Grenoble	0	0	4	4	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	3	0	0	0	0	0	0	0	0
Lille	2	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0
Limoges	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	4	0
Lyon	0	5	4	3	0	0	0	0	0	0	0	3	0	0	0	1	0	0	0	0	5	0	0	0	0	0	0	0	0	0	0	0
Marseille	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0
Metz	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	2	0	0	2	0	0
Montpellier	0	0	1	0	0	0	0	0	0	0	0	0	5	0	0	0	0	5	0	0	0	0	0	0	0	0	0	0	0	0	4	0
Nancy	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	2	0	0
Nantes	0	0	0	0	0	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	2	0	0	0	3
Nice	0	0	2	0	0	0	0	0	0	0	0	0	0	0	3	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0
Paris	2	2	0	0	0	0	3	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	2	0	0	0
Poitiers	0	0	0	0	0	3	2	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	2	0	0	0	0	0	0	0	1
Reims	2	0	0	0	0	0	0	0	0	0	0	0	0	3	0	2	0	0	0	2	0	0	0	0	2	0	0	0	0	0	0	0
Rennes	0	0	0	0	0	0	0	4	3	0	4	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0
Rouen	2	0	0	0	0	0	0	0	2	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0
Strasbourg	0	0	0	3	0	0	0	0	0	0	0	0	0	3	0	0	0	0	0	2	0	2	0	0	0	0	0	0	0	0	0	0
Toulouse	0	0	0	0	4	4	0	0	0	0	0	0	5	0	0	0	4	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0
Tours	0	0	0	0	0	0	3	0	0	0	3	0	0	0	0	0	0	0	0	0	0	0	3	0	0	1	0	0	0	0	0	0

Figure 7: Matrice d'adjacence complète de notre carte de France

Bibliographie

Programmation

[1] *Bases de tkinter*. URL: https://realpython.com/python-gui-tkinter/.

[2] *Exemple d'architecture Modèle-Vue-Contrôleur sous tkinter*. URL: https://github.com/abdullahsumbal/mvc_template.

[3] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. "Exploring Network Structure, Dynamics, and Function using NetworkX". In: *Proceedings of the 7th Python in Science Conference*. Ed. by Gaël Varoquaux, Travis Vaught, and Jarrod Millman. Pasadena, CA USA, 2008, pp. 11–15.

[4] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[5] The pandas development team. *pandas-dev/pandas: Pandas*. Version latest. Feb. 2020. DOI: 10.5281/zenodo.3509134. URL: https://doi.org/10.5281/zenodo.3509134.

[6] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009. ISBN: 1441412697.

Connaissances scientifiques

[7] MANOUSSAKIS George et PILARD Laurence. *Cours de théorie des graphes de l'ISTY*.

Divers

[8] *Carte de la France*. URL: https://www.lewebpedagogique.com/mathasion/files/2013/09/2nde_TP1_repere-du-plan_20132014.pdf.