

# **Rendu final PPII**

**Mathis AULAGNIER**

**Matthias CABILLOT**

**Bastien DELHAYE**

**Ahtisham TOOR**

**Mai 2023**

# Table des matières

<b>Table des matières</b>	<b>1</b>
<b>Introduction générale</b>	<b>4</b>
Contexte . . . . .	4
Objectifs & Hypothèses . . . . .	4
Point de départ et délais . . . . .	5
Récupération des données . . . . .	5
<b>I Choix de l'algorithme principal</b>	<b>5</b>
<b>1 Quels sont nos possibilités ?</b>	<b>6</b>
1.1 Problème & solutions . . . . .	6
1.2 L'algorithme de Dijkstra . . . . .	6
1.3 L'algorithme Bellman-Ford . . . . .	7
1.4 L'algorithme A* . . . . .	7
1.5 L'algorithme naïf . . . . .	8
1.6 Conclusion . . . . .	8
<b>2 Preuve de M*</b>	<b>8</b>
2.1 Pseudo-code . . . . .	8
2.2 Illustration . . . . .	10
2.3 Complexité temporelle . . . . .	10
2.4 Terminaison . . . . .	11
2.5 Correction . . . . .	11
2.5.1 Cas de bases . . . . .	11
2.5.2 Induction . . . . .	11
2.5.3 Conclusion . . . . .	12
2.6 Ajout de paramètre dans M* . . . . .	12
2.6.1 Filtre disponible . . . . .	12
<b>3 Données nécessaires</b>	<b>13</b>
<b>II Choix &amp; implémentations des structures de données</b>	<b>14</b>
<b>4 Les voitures</b>	<b>14</b>
<b>5 Les localisations</b>	<b>15</b>
<b>6 Les stations</b>	<b>15</b>
<b>7 Les nœuds</b>	<b>16</b>
<b>8 Les graphes</b>	<b>16</b>

<b>9 Les bases de données courantes</b>	<b>17</b>
9.1 List_station . . . . .	17
9.2 Database . . . . .	17
9.3 Fonctions associées . . . . .	18
<b>III Implémentation des fonctions</b>	<b>20</b>
<b>10 Fonction du calcul de la distance à partir des coordonnées</b>	<b>20</b>
<b>11 Match</b>	<b>21</b>
<b>12 Find_cars_by_name</b>	<b>22</b>
<b>IV Simulation</b>	<b>23</b>
<b>13 Route</b>	<b>23</b>
<b>14 Simulation</b>	<b>24</b>
<b>V Visualisation</b>	<b>25</b>
<b>15 Carte de la France</b>	<b>25</b>
<b>16 Carte du monde</b>	<b>26</b>
<b>17 Le script python Mercator</b>	<b>26</b>
<b>18 Simulation en France</b>	<b>27</b>
<b>19 Simulation en France, cas extrême</b>	<b>28</b>
<b>VI Gestion de projet</b>	<b>29</b>
<b>20 SWOT</b>	<b>29</b>
20.1 Forces . . . . .	29
20.2 Faiblesses . . . . .	29
20.3 Opportunités . . . . .	29
20.4 Menaces . . . . .	29
20.5 Résumé . . . . .	30
<b>21 WBS</b>	<b>31</b>
<b>22 GANTT</b>	<b>31</b>

<b>23 Déroulement réel du projet</b>	<b>32</b>
<b>24 Conclusion du projet</b>	<b>33</b>
24.1 Différence avec l'ancien projet . . . . .	33
24.2 Leçons tirées . . . . .	33
24.3 Bilan . . . . .	33

# Introduction générale

## Contexte

Le prix de l'essence ne cesse d'augmenter et pour cause : c'est une ressource en voie d'épuisement et donc de plus en plus convoitée. En considérant son impact négatif sur l'environnement en plus, restreindre l'utilisation des moteurs thermiques est devenu une nécessité afin de nous nous préserver. Ainsi, la Commission Européenne a interdit la vente de nouvelles voitures à essence à partir de 2035.

Les routes étant très empruntées par les particuliers et les entreprises, trouver une alternative était obligatoire. L'une d'entre elles pourrait être les véhicules électriques.

De plus en plus performantes, les voitures électriques pourraient s'imposer comme le nouveau mode de transport privilégié. Et dans cette optique, de nombreux pays se sont positionnés en faisant construire des stations avec des bornes de recharge partout sur leur territoire, comme en France où l'on en dénombre 77 000 en fin d'année 2022 pour un objectif de 100 000. Nous pouvons aussi noter la présence de nombreuses aides qui vise à faciliter l'achat d'un tel véhicule. Les voitures électriques ont donc le vent en poupe et sont aujourd'hui au centre de l'attention.

Cependant, ce type de voiture présente un inconvénient : l'autonomie. En effet, une voiture électrique parcourt en moyenne 320 kilomètres avant de devoir aller en station alors que la moyenne des voitures thermique se situe plutôt autour de 500 kilomètres. Et même si l'écart entre les voitures électriques et thermiques se réduit avec le temps, le rechargeement reste un élément important à prendre en compte avant tout trajet conséquent. Ainsi, les applications de guidage doivent prendre en compte ce nouveau paramètre pour être aussi performantes qu'aujourd'hui pour les voitures thermiques, et c'est précisément le fil conducteur de ce projet.

Le but de ce projet est de fournir aux usagers (particuliers, professionnels et autorités de régulation) un ensemble de fonctions qui les aident dans le déploiement, le dimensionnement et la gestion d'un réseau de stations de recharge adapté aux besoins.

## Objectifs & Hypothèses

Le projet se subdivise en 2 étapes complémentaires :

La première étape consiste à produire une application codée en langage C qui fonctionne à partir de la ligne de commande (ou avec une interface plus poussées si le temps nous le permet) qui renvoie un chemin à suivre et qui prends comme arguments une station de départ, d'arrivée et la voiture qui sera utilisée pour le trajet. Pour aller plus loin, nous avons la possibilité d'enrichir

la fonction précédentes avec des paramètres pour pouvoir affiner la recherche de l'utilisateur selon des critères comme rester au dessus d'un certain pourcentage de batterie ou d'autres encore plus poussés.

La seconde étape vise à étendre la première. Le but ici est de créer une « simulation » destinée aux investisseurs et aux autorités. La conclusion de cette étape permettra de visualiser les déplacements et les éventuelles files d'attentes en temps réel.

Pour simplifier le problème, on considère qu'il existe un chemin entre chaque station. On considérera même que le trajet se fait à vol d'oiseau.

## Point de départ et délais

Nous allons utiliser deux bases de données. La première ([ici](#)) contient la liste des stations présentes sur le territoire français et à d'autres points d'intérêts autour du globe (en Algérie par exemple). La seconde ([ici](#)) recense des informations sur un grand nombre de véhicules électriques actuellement disponible sur le marché de l'automobile. Une étape préliminaire mais essentielle sera donc évidemment d'extraire ces informations et de les mettre en forme.

Notre groupe a été validé le **18 mars 2023**.

L'application ainsi que ce rapport sont dûs pour le **24 mars 2023**.

La soutenance finale aura lieu le **31 mai** et au cours de cette dernière nous devrons présenter l'intégralité du projet en deux parties. Il y aura une partie principale sur la structure du programme et les algorithmes utilisés et une partie secondaire orientée sur la gestion de projet.

## Récupération des données

La base de données contenant l'ensemble des informations sur les voitures électriques se trouve donc sur Internet sans pouvoir être simplement téléchargée. Elle n'est accessible que par requêtes HTML et ce faisant, son accès est limité. Le script que nous avons utilisé était efficace puisqu'il récupérait toutes les données en peu de requêtes et il a été mis à disposition de l'ensemble des équipes projets à la demande de M.Festor puisque certaines avaient eu leur accès au site restreint suite à l'envoi d'un nombre conséquent de requêtes.

# Première partie

## Choix de l'algorithme principal

### 1 Quels sont nos possibilités ?

#### 1.1 Problème & solutions

Avant d'extraire les informations des bases de données nous avons besoin de savoir quoi extraire. Et pour savoir cela, nous avons besoin de savoir ce que nous allons utiliser lors de l'exécution de notre programme pour être sûr de ne pas stocker d'informations inutiles ou de manquer d'informations. Ainsi

L'ensemble des stations de recharge représente un graphe non orienté complet puisqu'on considère qu'il existe un chemin entre chaque station. De plus, chaque arête est pondérée par la distance réelle entre les deux stations concernées. Notre objectif étant de trouver un chemin convenable pour l'utilisateur, nous avons le choix entre plusieurs algorithmes existant concernant la recherche de chemin. Les algorithmes les plus connus permettent même d'assurer que le chemin trouvé est le chemin le plus court. Nous nous intéresserons donc aux algorithmes suivants :

- Dijkstra
- Bellman-Ford
- A\*
- Algorithme naïf

#### 1.2 L'algorithme de Dijkstra

Cet algorithme a été publié pour la première fois par Edsger Dijkstra en 1959. Le fonctionnement de cet algorithme est récursif. Il nécessite une liste de sommets non visités L ainsi qu'un tableau T donnant pour chaque sommet S la distance minimale permettant de se rendre de A vers S.[1]

L'initialisation se fait en mettant tous les sommets dans L. Les distances dans T seront initialisées telles que la distance de A vers A soit 0 et que les autres soient infinies.

A chaque étape, on s'intéresse au sommet  $S_k$  encore présent dans L dont la distance depuis A est minimale. Pour chaque sommet  $S_a$  adjacent à  $S_k$ , on compare la distance de A à  $S_a$  dans T à la somme de la distance de A à  $S_k$  et la distance de  $S_k$  à  $S_a$ . On remplace la valeur de la distance de A à  $S_a$  dans T par le minimum des deux valeurs. Une fois tous les sommets adjacents visités, on retire  $S_k$  de L et on recommence.

L'algorithme termine lorsque l'algorithme visite Z. En effet, soit la distance de A vers Z est finie et donc (comme aucun des chemins n'a un poids négatif) on ne trouvera pas plus court. Soit cette distance est infinie et cela signifie qu'il n'y a pas de chemin de A vers Z. On se retrouve donc avec le chemin le plus court entre A et Z. En notant A le nombre d'arcs du graphe et S le nombre de sommets on trouve le plus court chemin avec une complexité allant de  $O((A + S) * \log(S))$  à  $O(A + S * \log(S))$  suivant la méthode choisie pour implémenter la liste de priorité L.[4]

### 1.3 L'algorithme Bellman-Ford

Cet algorithme a été publié à partir de 1956 par les chercheurs éponymes. Contrairement à l'algorithme de Dijkstra, cet algorithme autorise certains arcs à avoir une pondération négative et par conséquent, autorise l'existence de circuits absorbants.[3] Cet algorithme retrouve de manière itérative la distance minimale pour aller du sommet de départ A à un sommet S en franchissant un maximum de k arcs. Le nombre k d'arc varie de 0 au nombre de sommet moins 1 ce qui permet d'envisager tous les trajets possibles. En notant n le nombre de sommets du graphe, on peut remplir une matrice carrée de dimension n. Pour faciliter la compréhension on numérote les colonnes de 0 à n-1.

Chaque ligne correspondra à un sommet. Chaque case  $(i,j)$  de la matrice correspondra à la distance minimale pour aller du sommet 1 au sommet i en un maximum de j franchissements d'arc. La première colonne est initialisée par 0 pour le sommet initial et l'infini pour les autres. Il est ensuite possible de remplir les colonnes les unes après les autres. Pour chaque sommet D on regarde les sommets sources S permettant de s'y rendre. On note p le poids de l'arc  $(S,D)$  et on remplit la case  $(D, k)$  par le minimum entre  $(D, k-1)$  et la somme de  $(S, k-1)$  et p. Une fois tous les sommets sources explorés, on obtient bien la distance minimale pour aller du sommet initial au sommet D en un maximum de k franchissements d'arc. L'algorithme s'arrête lorsque la matrice est remplie. On peut alors lire la distance minimum dans la case  $(Z, n-1)$ .  
En notant S le nombre de sommets et A le nombre d'arêtes, la complexité de cet algorithme est  $O(S * A)$ .

### 1.4 L'algorithme A\*

Présenté pour la première fois en 1968 par trois chercheurs anglais, cet algorithme est une extension de l'algorithme de Dijkstra. Il a été conçu pour trouver un résultat le plus rapidement possible. Ainsi, la solution trouvée est une des meilleures mais n'est pas forcément la plus optimale.

Cet algorithme part du principe que pour se rendre du point A au point B, il faut se diriger vers B. Ainsi, le premier déplacement se fera dans cette direction

et ce jusqu'à ce que le chemin soit entravé ou qu'il relie bien le départ jusqu'à la destination. Les noeuds non choisis lors du premier passage car n'étant pas dans la bonne direction sont sauvegardés dans une file qui sera utilisée si le chemin direct n'aboutit pas. L'algorithme se termine si le but est atteint auquel cas il reconstitue le chemin nécessaire pour y arriver ou bien si la file de sommets disponibles est vide auquel cas il n'existe pas de chemin de A vers Z. Cet algorithme permet de trouver une solution correcte en un temps relativement court.

## 1.5 L'algorithme naïf

Dans le cas présent l'algorithme naïf est celui où à chaque itération on prend la station la plus proche de la station d'arrivée dans la liste des stations atteignables pour la voiture sélectionnée. Si le programme dépasse un certain nombre N d'itérations (fixé arbitrairement) alors le programme s'arrête sans avoir trouvé de chemin.

L'algorithme naïf est dans ce cas très intéressant. En effet, les algorithmes précédents "reculent" tous lorsqu'ils arrivent sur une station qui ne peut atteindre la destination. Avec cet algorithme et dans notre situation, il y aura toujours un chemin entre l'arrivée et le départ (tant qu'elles sont sur le même continent, puisque notre base de données contient des stations éparpillées sur tout le globe) puisque la distance entre chaque station est petite comparée à la distance qu'une voiture peut parcourir en moyenne avant de tomber en panne.

## 1.6 Conclusion

Chaque algorithme a ses avantages et ses inconvénients. Bien que les trois premiers évoqués soient fiables, démontrés et qu'ils trouvent le chemin le plus court, ils impliquent des implémentations de structures supplémentaires pouvant ralentir le programme et le développement du projet. Ainsi pour une implémentation plus claire, une meilleure lisibilité du programme, une complexité équivalente (preuve ci-après) et une plus grande flexibilité au niveau des paramètres à implémenter nous utiliserons l'algorithme naïf que nous nommerons M\*.

# 2 Preuve de M\*

## 2.1 Pseudo-code

---

**Algorithme 1 : Algorithme M\***

---

**Entrées :** un graphe G,  
une station de départ From,  
une station d'arrivée To,  
une voiture V.

**Sorties :** Un chemin entre From et To passant par des sommets de G

initialisation ;  
path = []  
N = 100 //Arbitraire  
Ajouter From à path  
count = 0  
range = V.range  
bestH = dist(From,To)  
currentStation = From

**tant que** bestH supérieure ou égale à range **faire**

**si** count == N **alors**  
| retourner path  
**fin**  
best = NULL

**pour** i allant de 0 à N **faire**

| Gi = station numéro i de G  
| **si** Gi est différent de From **alors**  
| | d = dist(currentStation,Gi)  
| | **si** d strictement inférieur à range **alors**  
| | | h = dist(Gi, To)  
| | | **si** h strictement inférieur à bestH **alors**  
| | | | best = Gi  
| | | | bestH = h  
| | **fin**  
| **fin**  
**fin**

**si** best == NULL **alors**  
| renvoyer path  
**fin**  
currentStation = best  
Ajouter currentStation à path  
Incrémenter de 1 count

**fin**  
Ajouter To à path  
Renvoyer path

---

## 2.2 Illustration

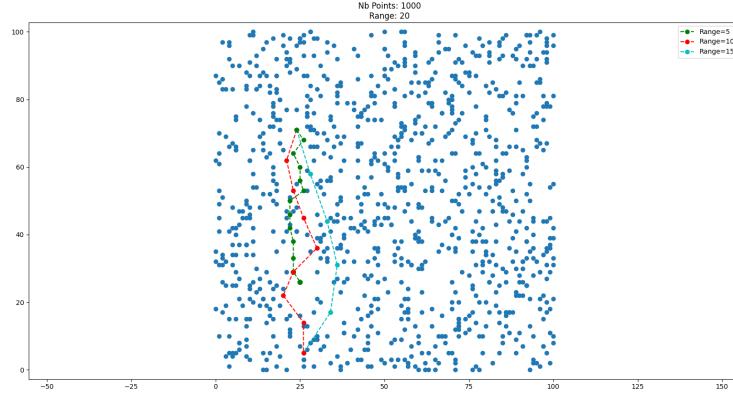


FIGURE 1 – Illustration de l'éventuel problème de  $M^*$

Cette simulation est indépendante du problème, elle permet simplement de mettre en évidence le problème évoqué dans la partie précédente.

Dans cette simulation, il y a 1000 points générés aléatoirement dans un certain périmètre. On a testé 3 voitures qui ont des portées différentes en les faisant partir du même point et en ayant comme objectif le même point.

On observe clairement que les voitures de portée 15 et 10 arrivent à destination, mais surtout que la voiture de portée 5 n'arrive pas à destination car elle arrive à un point qui est trop éloigné des autres points pour pouvoir avancer.

## 2.3 Complexité temporelle

L'ensemble des stations forme un graphe de  $S$  sommets et de  $A=S^{(S-1)}$  arrêtes.

L'algorithme  $M^*$  consiste à toujours se rapprocher au maximum de la station finale avec un nombre de stations intermédiaires maximal prédefini, nommons  $N$ . Cependant pour chaque stations intermédiaires, l'algorithme regarde toutes les stations existantes pour savoir si elles sont atteignables pour la voiture, et ce en fonction des paramètres données, en effet même si un chemin physique existe, rien ne nous assure que la voiture peut parcourir la distance entre les deux points. Or, savoir si une station est atteignable ou non est en temps constant puisqu'il suffit de calculer la distance entre la station actuelle de la voiture et la station que l'on observe.

Ainsi l'algorithme  $M^*$  est d'une complexité en  $O(S^*N)$  car il y a au maximum  $N$  itérations, chacune de complexité en  $O(S)$  pour devoir regarder chacun des sommets du graphe.

## 2.4 Terminaison

Il y a au maximum  $N$  itérations, ainsi la terminaison est assurée.

## 2.5 Correction

Nous allons faire une preuve par induction. Le but étant de prouver que s'il y a une solution, l'algorithme en donne une et que si il n'y en a pas, l'algorithme renvoie un chemin qui ne termine pas à l'arrivée prévue. Les hypothèses suivantes sur les paramètres sont admises :

- La station de départ est différente de celle d'arrivée
- Elles sont sur le même continent. (Exemple : nous n'essayerons pas de chercher un trajet en voiture entre la Guyane et la Réunion)

### 2.5.1 Cas de bases

La station de départ est suffisamment proche de la station d'arrivée pour que la voiture fasse le trajet sans détour. Ce cas est validé, en effet on a bien `From` dans `path` grâce à l'initialisation mais aussi `To`. Et puisque on ne rentre pas dans la boucle « Tant que » car la distance parcourable par la voiture dépasse la distance entre le départ et l'arrivée, on est assuré qu'il n'y aura pas d'autres stations dans `path`.

### 2.5.2 Induction

On considère qu'il y a déjà eu  $k$  stations ajoutées dans `path`. Sachant que la  $k$ -eme station n'est pas la station d'arrivée. On cherche donc à savoir quelle sera la  $k+1$ -eme station à ajouter.

- Si  $k$  est égal à  $N$  :
  - Le programme s'arrête et renvoie `path` tel quel. Ainsi la  $k$ -eme station n'est pas la station d'arrivée.
- Si  $k$  est plus petit que  $N$  :
  - Parmi les stations qui peuvent être atteintes, on prend une station  $G_i$  du graphe qui n'est pas `From` et l'on regarde si la distance entre la station  $k$  et  $G_i$  est inférieure à la distance parcourable par la voiture, dans ce cas on la compare à la meilleure station disponible parmi les

stations considérées par l'algorithme lors de la boucle « Pour », et si  $G_i$  est plus proche de To alors on considère que la meilleure station disponible est  $G_i$ . A la fin de la boucle « Pour » la meilleure station disponible est bien la plus proche de la station d'arrivée parmi les stations accessibles par la voiture.

- On ajoute alors cette meilleure station à path et elle représente donc la  $k + 1^{eme}$  station.
- Evidemment, on peut remarquer que si la station d'arrivée est atteignable alors la boucle « Tant que » s'arrête et on ajoute alors la station d'arrivée à path avant de renvoyer ce dernier.

### 2.5.3 Conclusion

Par induction structurée nous venons de prouver que l'algorithme  $M^*$  nous renvoie bien ce qu'on attend de lui.

## 2.6 Ajout de paramètre dans $M^*$

Pour pouvoir répondre au cahier des charges, et ainsi pouvoir ajouter des paramètres nous modifions le prototype de la fonction  $M^*$  pour que la fonction puisse filtrer les stations.

```

1 List_Station mstar(Graph* G, station_t* from, station_t* to,
2 voiture* car, bool (*filter)(Node*, Node*, voiture*));

```

### 2.6.1 Filtre disponible

Ne pas passer en dessous de 20% de batterie  
 Ne pas passer en dessous de 40% de batterie  
 Ne pas passer en dessous de 60% de batterie  
 Ne pas passer en dessous de 80% de batterie  
 //Code en C

```

1 bool keep20percentfuel(Node* current_node, Node* node, voiture
2 * car)
3 {
4     float d = dist_on_earth(current_node->station->location,
5     node->station->location);
6     return (float)car->range - d > 0.2*(float)car->range;
7 }

```

### 3 Données nécessaires

Ainsi, comme nous avons l'algorithme en pseudo-code nous pouvons déterminer quelles données seront utiles pour l'implémentation :

Pour chaque voiture :

- Nom
- Portée (ou "range") en km
- Efficacité de la batterie en Wh/km#
- Batterie pleine (=Portée\*Efficacité)##

Pour chaque station :

- Nom
- Puissance disponible en kW
- Localisation (Coordonnées GPS)
- Accès gratuit #
- Accès libre #

# marque les données qui deviennent nécessaires selon les paramètres choisis

# Deuxième partie

## Choix & implémentations des structures de données

### 4 Les voitures

Nous avons commencé par nous intéresser à l'implémentation des voitures. On récupère les informations demandées par l'algorithme qu'on utilise, ainsi que la marque pour pouvoir l'utiliser lors de l'exécution du programme dans des fonctions annexes. On a ajouté une structure *voiture\_array* qui est donc une liste contigüe de voitures, puisque nous l'utilisons que dans deux cas particuliers nous avons décidé de ne pas implémenter complètement la structure de données (avec une fonction d'ajout, de suppression d'élément ...).

Pour chaque voiture :

- Nom
- Marque
- Portée (=range) en km
- Efficacité batterie en Wh/km
- Batterie pleine (= *Portée* × *Efficacité*)

//Code en C

```
1 struct voiture{
2     char* brand;
3     char* name;
4     int range; //km
5     int efficiency; //Wh/km
6     int full_battery; // Wh
7 };
8 typedef struct voiture voiture;
9
10 struct voiture_array{
11     voiture* cars;
12     int size;
13 };
14 typedef struct voiture_array voiture_array;
```

## 5 Les localisations

Les localisations serviront à situer les bornes et uniquement les bornes. Ainsi elles comportent une adresse, une longitude et une latitude.

//Code en C

```
1 struct location{
2     char* addr;
3     float longitude; // Degres
4     float latitude;
5 };
6 typedef struct location location_t;
```

## 6 Les stations

Afin de permettre à l'utilisateur d'avoir un maximum d'informations sur les stations par lesquelles il peut passer, nous avons décidé de récupérer à la fois les informations nécessaires pour M\* mais aussi les restrictions, l'opérateur, le numéro de téléphone de ce dernier ainsi que des informations supplémentaires partagées par le propriétaire/constructeur.

De plus, nous avons remarqué lors de la récupération des données que les bornes de charges étaient toutes différencierées, même les bornes qui sont dans les mêmes stations. Ainsi nous avons décidé de les regrouper ensemble et d'ajouter dans la structure d'une station, une donnée avec le nombre de bornes de la station.

//Code en C

```
1 struct station{
2     char* name;
3     char* restrictions;
4     char* operator;
5     char* more_info;
6     char* tel_ope;
7     int power; //en kW
8     float tarif;
9     bool free; // gratuit
10    bool free_access; // acces libre
11    bool pmr;
12    location_t* location;
13    int nbr;
14 };
15 typedef struct station station_t;
```

## 7 Les nœuds

Un nœud est composé d'un pointeur vers une station et du nombre de voiture dans la station ou en attente.

//Code en C

```
1 typedef struct Node
2 {
3     station_t* station; // Nœud
4     int nbr_voiture; // Nombre de voiture dans ou en attente à
5     la station
6 } Node;
7 typedef struct station station_t;
```

## 8 Les graphes

Un graphe est composé d'un entier donnant sa taille, un entier donnant le nombre de stations ainsi qu'un tableau de pointeur vers les nœuds du graphe. On diffère la taille et le nombre de stations pour ne pas devoir réallouer la mémoire à chaque ajout d'élément.

//Code en C

```
1 typedef struct graphcomplet
2 {
3     int size; // La Taille totale du graphe
4     int nbr\station; // Le nombre d'elements présent dans le
5     graphe
6     Node* nodes; // Tableau de pointeur pointant vers les
7     noeuds du graphe
8 } GraphComplet;
9
10 typedef GraphComplet Graph;
```

## 9 Les bases de données courantes

### 9.1 List\_station

Liste chaînée de station. Cette fois-ci l'implémentation est complète.  
//Code en C

```
1 struct elt{
2     station_t* elt;
3     struct elt* next;
4 };
5 struct List_Station{
6     struct elt* first;
7     int size;
8 };
9 typedef struct List_Station List_Station;
```

### 9.2 Database

Contient un pointeur vers une List\_station, un pointeur vers un graphe ainsi qu'un entier pour sa taille et un autre pour le nombre d'éléments. On diffère la taille et le nombre d'éléments pour ne pas devoir réallouer la mémoire à chaque ajout d'élément.

//Code en C

```
1 struct Database{
2     List_Station* lists;
3     int size;
4     int nbr_elt;
5     Graph* G;
6 };
```

Pour répondre aux besoins de stockage et d'accès efficace aux données, nous avons décidé d'implémenter une table de hachage ou *hashmap*. Ainsi, à chaque nouvelle entrée dans database ou accès à un élément de la structure database, nous faisions appel à notre fonction de hachage. Nous avons délibérément choisi une fonction très simple :

```
1 int hash(const char* str)
2 {
3     int hash = 0;
4     int i=0;
```

```

5   while( str [ i ]!= '\0' )
6   {
7       hash += str [ i ];
8       i++;
9   }
10  return hash ;
11 }
```

### 9.3 Fonctions associées

Pour compléter l'implémentation de ces structures de données, nous avons fait toute une gamme de fonctions classiques pour créer une instance, ajouter des éléments, supprimer des éléments et etc. Voici une liste résumant les plus importantes d'entre elles :

La fonction *create\_list* permet de créer une nouvelle liste vide et de renvoyer une instance de *List\_Station*.

La fonction *list\_append* permet d'ajouter une station à la fin de la liste. Si la liste est vide, un nouvel élément est alloué et initialisé avec la station. Sinon, on parcourt la liste jusqu'à la fin et on ajoute un nouvel élément contenant la station.

La fonction *list\_get* renvoie la station à l'index spécifié dans la liste. On parcourt la liste jusqu'à l'index souhaité et on renvoie la station correspondante.

La fonction *list\_find\_by\_name* permet de rechercher une station dans la liste en utilisant son nom. On parcourt la liste et on compare les noms des stations avec la chaîne de recherche.

Les fonctions *print\_list*, *println\_list* et *free\_list* permettent respectivement d'afficher, d'afficher avec un saut de ligne et de libérer la mémoire utilisée par la liste.

La structure Database est utilisée pour regrouper plusieurs listes de stations dans une structure de données plus large, comme expliqué plus haut. Elle contient un tableau de *List\_Station* (*lists*), la taille du tableau (*size*), le nombre total d'éléments (*nbr\_elt*), ainsi que d'autres attributs liés à la gestion de la base de données.

La fonction *create\_db* permet de créer une nouvelle base de données avec une taille spécifiée et de renvoyer une instance de Database. Cette fonction alloue la mémoire pour le tableau de listes et initialise la base de données.

La fonction *db\_add* est utilisée pour ajouter une station à la base de données. Elle calcule d'abord l'indice de hachage en utilisant la fonction de hachage *hash* et ajoute ensuite la station à la liste correspondante dans la *hashmap*. Si une collision se produit, la fonction *list\_append* est utilisée pour gérer le chaînage.

Les fonctions *db\_find* et *db\_get* permettent respectivement de rechercher une station par son nom ou d'obtenir une station à un index donné dans la base de données.

La fonction *load\_db\_from\_csv* permet de charger les données de la base de données

à partir du fichier CSV source. Elle lit le fichier ligne par ligne, extrait les informations de chaque station et les ajoute à la base de données en utilisant la fonction *db\_add*.

Enfin, la fonction *free\_db* libère la mémoire occupée par la base de données en appelant la fonction *free\_list* pour chaque liste de stations et en libérant également la mémoire utilisée par le graphe.

## Troisième partie

# Implémentation des fonctions

### 10 Fonction du calcul de la distance à partir des coordonnées

On utilise la fonction Haversine pour calculer la distance entre deux points dont les coordonnées sont des longitudes et latitudes. La France est suffisamment plate pour pouvoir négliger les quelques kilomètres d'erreurs à cause de la courbure de la Terre, mais puisque dans les deux cas c'est en temps constant nous avons opté pour la précision.

$$d = 2r \arcsin \left( \sqrt{\sin^2 \left( \frac{\varphi_2 - \varphi_1}{2} \right) + \cos(\varphi_1) \cos(\varphi_2) \sin^2 \left( \frac{\lambda_2 - \lambda_1}{2} \right)} \right)$$

FIGURE 2 – Formule de la distance entre deux point à partir de la formule Haversine

$$d = 2r \operatorname{Arcsin} \left( \sqrt{\sin^2 \left( \frac{\phi_2 - \phi_1}{2} \right) + \cos \phi_1 \cos \phi_2} \right)$$

Où r est le rayon de la Terre en km,  $\varphi$  une latitude en radians,  $\lambda$  une longitude en radians et d la distance recherchée en km.

## 11 Match

C'est une fonction qui prend en entrée deux chaînes de caractères, s et tok et qui va vérifier si tok est dans s.

//Code en C

```
1  bool match( const char* s, const char* tok)
2  {
3      int tok_len = strlen(tok);
4      int s_len = strlen(s);
5      for( int i=0; i<s_len-tok_len+1; i++)
6      {
7          for( int j=0; j<tok_len; j++)
8          {
9              if(lower_a(s[i+j])!=lower_a(tok[j])) //lower_a(d)
rend d minuscule
10             break;
11             if(j==tok_len-1)
12                 return true;
13         }
14     return false;}
```

## 12 Find\_cars\_by\_name

C'est une fonction qui renvoie un *voiture\_array* contenant toutes les voitures qui contient le nom recherché. Ainsi lors de l'utilisation du programme, l'utilisateur n'est pas obligé de donner le nom complet du modèle de sa voiture, la marque peut suffire.

//Code en C

```
1  voiture_array find_cars_by_name(voiture_array db, const char*
2   name)
3   {
4     voiture_array rtn;
5     rtn.cars = calloc(db.size, sizeof(voiture));
6     rtn.size = 0;
7     for(int i=0; i<db.size; i++)
8     {
9       if(match(db.cars[i].name, name))
10      {
11        copy_car(rtn.cars + rtn.size, db.cars + i);
12        rtn.size++;
13      }
14    }
15    rtn.cars = reallocarray(rtn.cars, rtn.size, sizeof(voiture));
16  }
17 }
```

## Quatrième partie

# Simulation

Dans le cadre de notre projet, nous avons adopté deux approches distinctes pour atteindre les objectifs. La première approche, nommée *Route*, permet de donner l'itinéraire pour une seule voiture, tandis que la seconde offre la possibilité de simuler la demande d'un grand nombre de voitures.

### 13 Route

Une fois que le programme *Route* est compilé et prêt à être lancé, l'utilisateur a la possibilité de fournir en argument le modèle exact de la voiture utilisée ou simplement son nom. Dans ce deuxième cas, le programme se charge de trouver la première voiture correspondante au nom spécifié. Si aucun argument n'est fourni, le programme affiche la liste des voitures prises en charge pour permettre à l'utilisateur de sélectionner la voiture souhaitée en entrant son identifiant affiché à côté. Enfin, l'utilisateur est invité à entrer la longitude et la latitude de son point de départ, et dans un deuxième temps, les coordonnées de la destination.

Nous avons ajouté une fonctionnalité à *Route* qui permet lors de la compilation du programme de prendre en charge des filtres. Ces filtres sont des fonctions booléennes qui prennent trois arguments : la station où se trouve la voiture, la potentielle station où la voiture pourrait se rendre et le modèle de la voiture. Ces fonctions sont appelées et insérées dans le programme à l'aide de pointeurs de fonction.

Exemple de pointeur en C :

```
1 bool (*keep20percentfuel)(Node* current\_node , Node* node ,  
voiture* car)
```

Les pointeurs offrent la possibilité de référencer et d'appeler des fonctions de manière dynamique, permettant ainsi à ((M\*)) d'adapter son comportement en fonction des filtres spécifiés. Les filtres peuvent être utilisés de différentes manières, par exemple pour définir des limites ou des conditions spécifiques, ajoutant ainsi une dimension supplémentaire à la fonctionnalité de "Route".

## 14 Simulation

La fonction *Simulation* prend deux paramètres, une entrée et une sortie. Celles-ci peuvent être des fichiers textuels mais aussi l'entrée ou la sortie standard (*stdin* ou *stdout*). Par rapport à *Route*, *Simulation* peut récupérer ses données d'entrée depuis un fichier ou *stdin* selon la forme suivante :

```
1 Voiture ; lat_depart ; long_depart ; lat_arrivee ; long_arrivee
2 Aiways U6 ;46.31971;7.78046908;45.76011619906451;2.2906
3 Mazda MX-30 ;47.789103;4.8863;49.1296;2.465482
4 Abarth 500e Scorpio;47.2054; -1.0956;48.7286;3.314407
5 Mercedes EQA 250+;45.774634;0.670505;45.759568; -3.975574
```

Pour réaliser nos tests nous avons écrit le script python *create\_input\_station.py* qui génère de manière aléatoire un fichier de ce type. Le programme va ensuite charger les différentes stations de recharge dans un graphe. Puis, il va calculer les différents chemins pour chaque voiture en discrétilisant sous forme d'étape représentées par l'arrêt à une station le cheminement des voitures. On remarquera que les voitures ne sont pas contraintes de partir et d'arriver à une station existante. Le programme écrit ensuite les chemins trouvés dans un fichier passé en argument ou dans *stdout*. Cela permet de rediriger la sortie vers un script Python qui affichera les chemins sur une carte. Cela sera développé dans la partie Visualisation.

# Cinquième partie

## Visualisation

### 15 Carte de la France

Grâce à un script en Python, nous avons pu générer une carte sur laquelle il est possible de voir chacune des stations en France. Ce qui nous confirme que notre hypothèse selon laquelle à partir d'une station A, une voiture aura toujours un chemin de stations intermédiaires pour aller à une station B est bien cohérente.

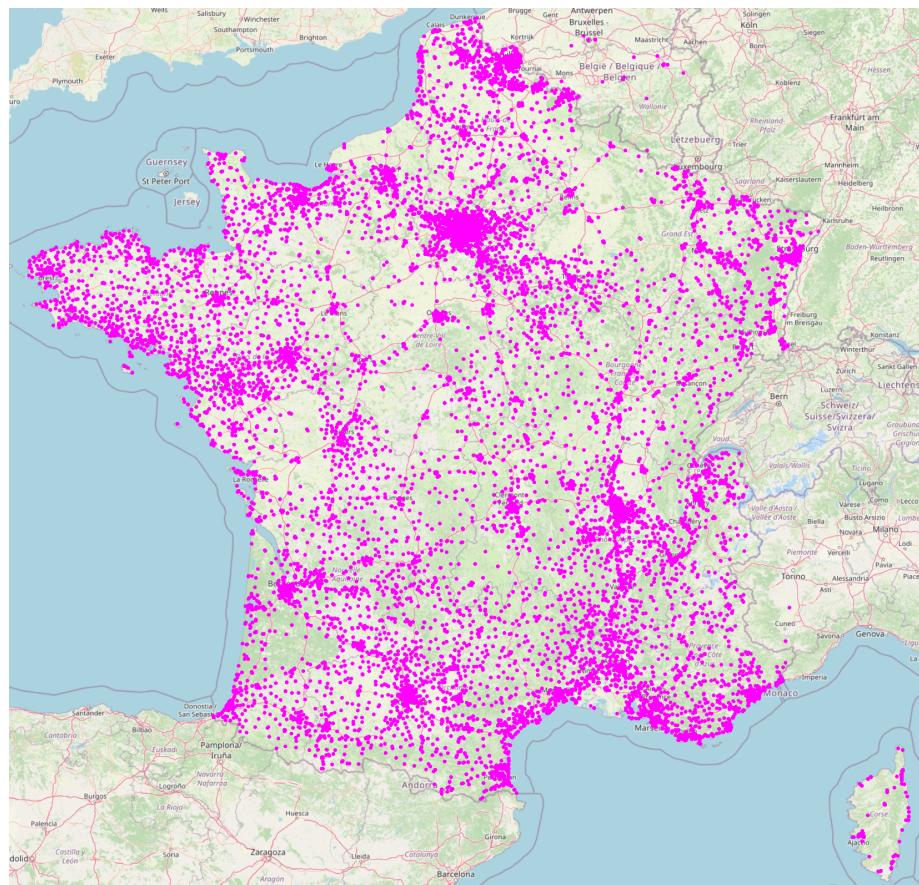


FIGURE 3 – Carte de la France avec les stations

## 16 Carte du monde

De la même manière nous avons décidé d'avoir une carte du monde avec les stations représentées dessus. C'est évidemment peu lisible mais c'est ainsi que nous avons pris conscience de la totalité des terres différentes sur lesquelles reposent les bornes.

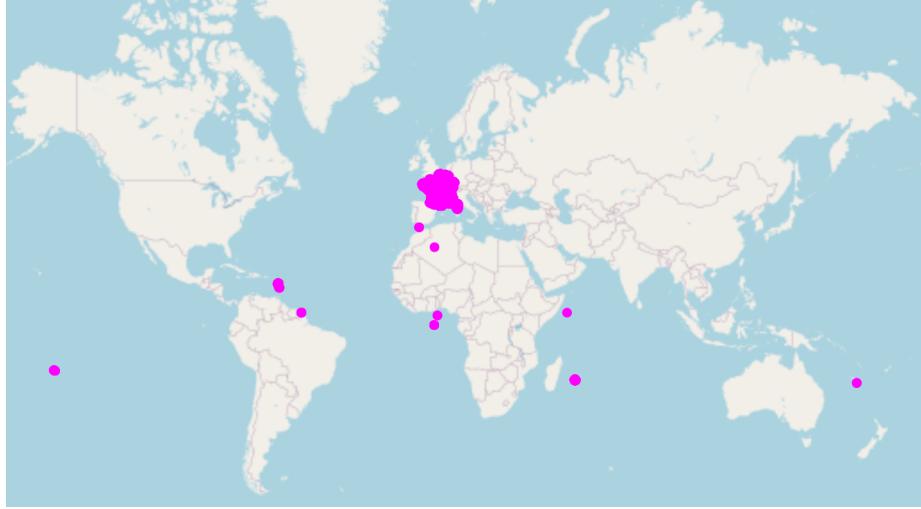


FIGURE 4 – Carte du monde avec les stations

## 17 Le script python Mercator

La formule de Mercator permet de faire le lien entre des coordonnées (longitude en degré :  $\lambda$ , latitude en radians :  $\phi$ ) d'un point sur Terre et un point de coordonnées (x en pixels ; y en pixels) sur l'image de référence de Mercator en dimension (h pixels \* w pixels).

$$x = w * (\lambda + 180) / 360$$

$$y = h / 2 - w / (2 * \pi) \ln(\tan(\pi / 4 + \phi / 2))$$

Grâce à ces formules et la librairie matplotlib nous sommes en capacités d'afficher les points sur une carte au travers du script python mapper.py. Ce dernier est capable de recevoir des informations redirigées depuis la sortie standard (stdout), qu'il pourra afficher les données comme sur les figures suivantes.

## 18 Simulation en France

Pour générer un exemple à simuler, nous nous sommes proposé de prendre des trajets aléatoires. Seulement, pour simplifier la génération des positions des voitures, nous avons décidé de simplement générer de manière aléatoire des coordonnées dans une zone rectangulaire englobant la France.

Ainsi il survient deux problèmes purement visuels :

- Certaines voitures sont en dehors de ce qu'appelle le programme "France" (le carré choisi dans le programme est plus restrictifs) et ne peuvent rejoindre une quelconque station car le programme pense que ces voitures ne sont pas sur le même territoire.
- Sachant que la France n'est pas parfaitement rectangulaire, des voitures commencent ou finissent dans l'eau.

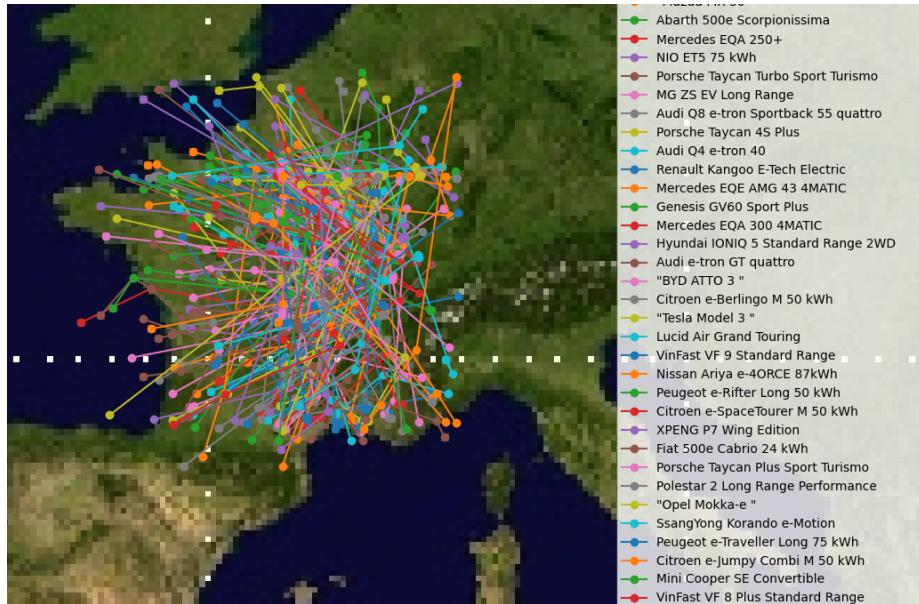


FIGURE 5 – Simulation en France

## 19 Simulation en France, cas extrême

Dans l'optique de tester les limites de notre programme, il fallait faire un essai avec un grand nombre de voitures qui partent en même temps. Ainsi, nous avons fait ce test avec 600 voitures qui devaient faire Marseille-Brest simultanément. Comme la figure suivante le montre, le programme s'est exécuté sans problème et chacune des voitures arrive à destination avec plus ou moins d'arrêts selon le modèle de la voiture et sa range.

Ce résultat, combiné avec le précédent et avec tous ceux qui ne sont pas présentés dans ce rapport, montrent que le programme fonctionne. La sortie obtenue est conforme à la demande que représentait l'entrée.

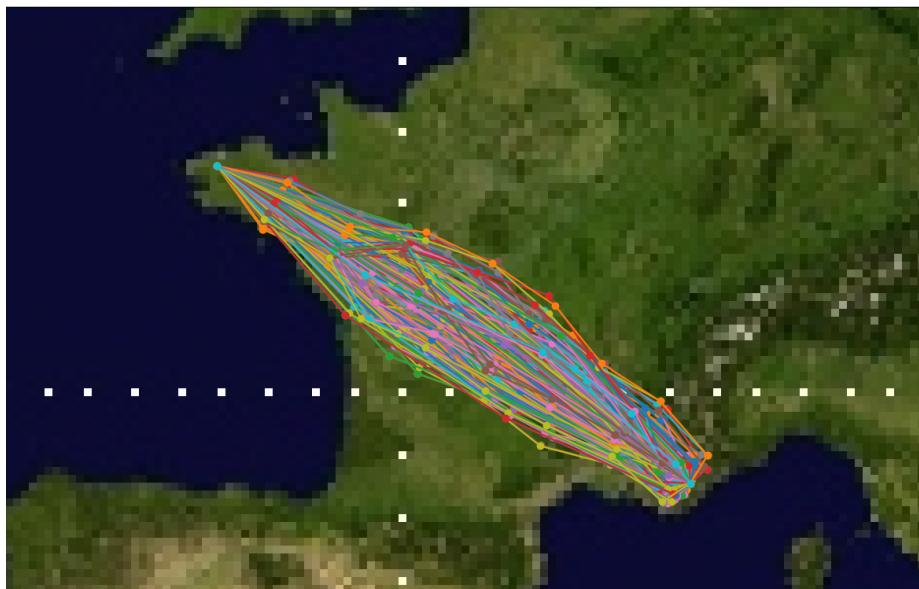


FIGURE 6 – Simulation en France

# **Sixième partie**

## **Gestion de projet**

### **20 SWOT**

#### **20.1 Forces**

L'équipe a travaillé ensemble par le passé, ce qui facilite sa communication interne et sa collaboration sur le projet.

Les membres de l'équipe savent programmer en C, ce qui est essentiel pour ce projet.

La problématique posée est complexe, concrète et intéressante, ce qui peut motiver l'équipe à travailler dur pour réussir.

#### **20.2 Faiblesses**

Il y a un écart de niveau entre les membres du groupe, ce qui peut rendre difficile la répartition équitable des tâches et le travail collaboratif.

Les membres du groupe ont seulement deux semaines de vacances, ce qui limite le temps disponible pour s'investir sur le projet.

#### **20.3 Opportunités**

Le projet peut aider l'équipe à acquérir de nouvelles compétences en programmation et en gestion de projet, ce qui peut être bénéfique pour leur avenir professionnel.

Si le projet est bien réussi, il peut être présenté comme un projet de référence pour les futurs employeurs ou projets universitaires.

La navigation en voiture est un souci quotidien, le projet est donc plus significatif et motivant puisque l'impact est visible à bout de bras.

#### **20.4 Menaces**

Le projet peut prendre plus de temps que prévu, ce qui peut entraîner une charge de travail supplémentaire pour les membres de l'équipe.

Les membres du groupe doivent également se préparer pour des examens de fin d'année, ce qui peut réduire le temps disponible pour le projet.

Il y a peut-être d'autres équipes travaillant sur des projets similaires, ce qui peut entraîner une concurrence pour les ressources et les compétences.

## 20.5 Résumé

L'équipe a des avantages considérables comme son expérience de travail ensemble et sa connaissance de la programmation en C. Et il y a aussi la motivation de réussir un projet complexe et intéressant. Cependant des défis sont présents, notamment l'écart de niveau entre les membres de l'équipe, la charge de travail supplémentaire liée aux examens de fin d'année et la limite de temps en raison des vacances de deux semaines.

En considérant les opportunités, il est possible pour l'équipe de gagner de nouvelles compétences en programmation et en gestion de projet, ce qui pourrait être bénéfique pour leur future carrière. Si le projet est bien réalisé, il pourrait embellir le CV des membres de l'équipe. Dans le meilleur des cas, le projet pourrait aboutir vers une vraie application de navigation automobile utilisée tous les jours par de vrais utilisateurs, ce qui rend le projet encore plus motivant.

Enfin, en considérant les menaces, il est important de noter que le projet peut prendre plus de temps que prévu et que des problèmes techniques pourraient survenir lors de la mise en œuvre de l'application, ce qui pourrait entraîner des retards et des coûts supplémentaires. Par ailleurs, la concurrence d'autres équipes travaillant sur des projets similaires pourrait entraîner des défis pour la gestion des ressources, des compétences et l'originalité du livrable.

En utilisant cette matrice SWOT, l'équipe peut évaluer les avantages et les risques associés à ce projet et ainsi élaborer une stratégie pour réussir. Par exemple, l'équipe peut décider de travailler ensemble pour combler l'écart de compétences entre les membres de l'équipe, ou de planifier soigneusement leur temps pour intégrer les examens de fin d'année et les vacances dans le projet. En identifiant les opportunités, l'équipe peut également décider de se concentrer sur la création d'une application de navigation automobile utile et conviviale qui pourrait avoir une grande valeur pour les utilisateurs.

## 21 WBS

Pour pallier aux quelques problèmes de communication et afin d'avoir une idée claire des tâches à accomplir pour le projet, nous avons établis un Work Breakdown Structure (WBS). Les tâches seront donc attribuées en fonction de ce document.

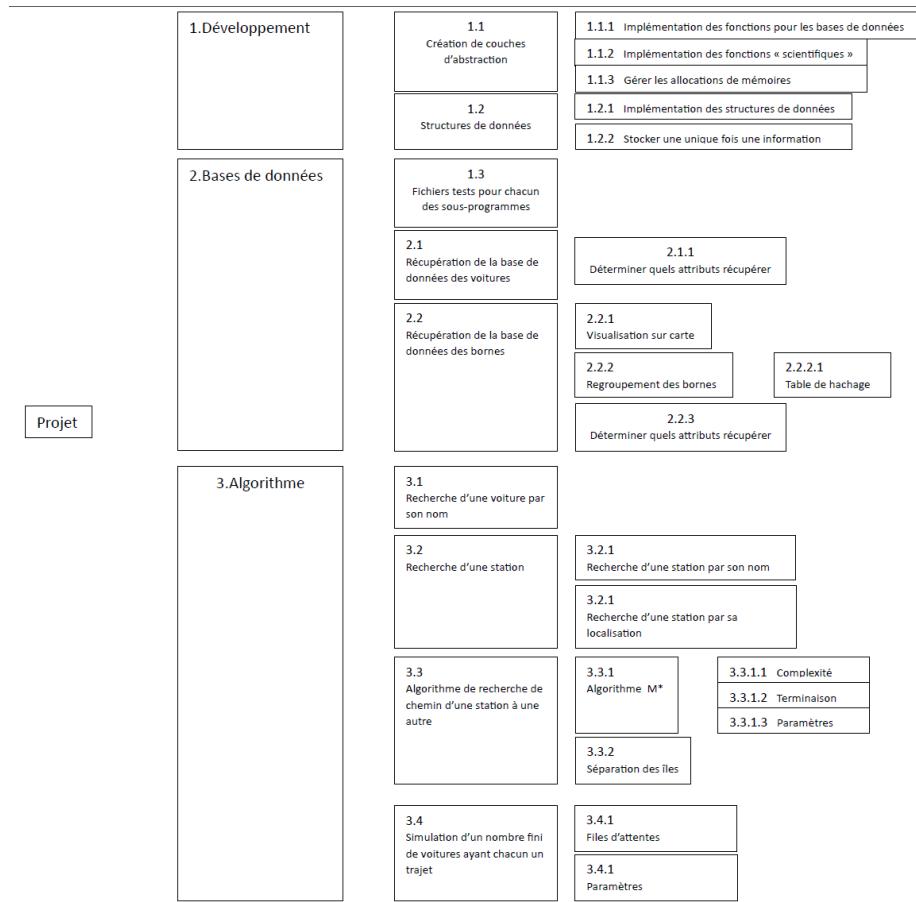


FIGURE 7 – Work Breakdown Structure

## 22 GANTT

Lors du projet précédent, nous avions opté pour les méthodes agiles qui nous demandaient de rebondir constamment sur l'avancement du projet. Cette fois-ci, ayant déjà fait l'expérience des examens pendant la remise du projet, nous

avons décidé d'utiliser les méthodes plus classiques, impliquant l'utilisation d'un document Gantt permettant ainsi à chacun d'avoir une idée sur l'avancement du projet et d'éviter l'effet tunnel. De plus cela nous permet de planifier des réunions en avance, octroyant la possibilité à tout le monde de s'organiser en fonction.

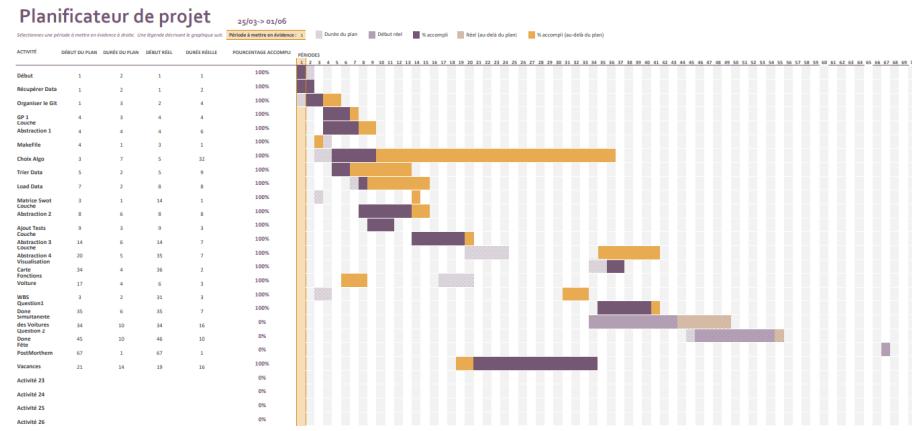


FIGURE 8 – GANTT

## 23 Déroulement réel du projet

Des réunions régulières ont été tenus, ces jalons nous ont permis d'exprimer nos différents problèmes rencontrés lors de nos tâches respectives. Mis à part quelques difficultés sur le langage C en lui-même, les délais ont tous été respectés. Les réunions permettaient aussi de suivre le projet dans sa totalité et de modifier les délais en fonction de l'avancement du projet. En effet alors que temps passait, nous avons refondé plusieurs fois notre système de structure pour répondre au mieux aux attentes du cahier des charges. Nous avons donc rapidement été obligé de revoir nos ambitions vis-à-vis du projet.

En effet nous voulions au départ avoir plusieurs paramètres différents, de natures différentes. Malheureusement nous avons du nous restreindre sur la nature des paramètres. Nous avons préférés nous focaliser sur la partie "Simulation" qui nous a pris plus de temps que prévu.

## 24 Conclusion du projet

### 24.1 Différence avec l'ancien projet

Utiliser des tableaux de bord de suivi de projet en ligne a permis une meilleure visibilité sur l'avancement du projet pour tous les membres de l'équipe. Tenir des réunions de stand-up régulièrement a contribué à une communication efficace au sein de l'équipe et à la résolution rapide des problèmes. Cette fois-ci nous avons établi des "conventions" de codage, ce qui nous a permis de faciliter la compréhension à chaque fois que du code était proposé.

### 24.2 Leçons tirées

Les méthodes classiques de gestion de projet avec le document GANTT sont peu flexibles et ajoutent une pression psychologique aux membres du groupe. Dans un projet avec autant d'inconnues comme celles que comporte le fonctionnement du langage C, ce type de méthode est certainement à éviter. En effet nous n'avions pas prévu autant de contre-temps, et c'est pour cela que tout n'est pas fini.

Malgré le fait que notre équipe ait déjà travaillé ensemble, nous avons eu des problèmes de communications, mais beaucoup moins que lors du dernier projet. De même, pour l'implication de chacun, il y a un manque d'initiatives de la plupart des membres, probablement dû à un problème de motivation vis-à-vis du sujet.

On peut noter toutefois que la précision lors des messages des commit sur le dépôt git est un atout majeur de l'équipe.

### 24.3 Bilan

Le répertoire GitLab du projet sera fermé le 24 mai à 20h.

Nous nous sommes assurés que le rendu soit conforme aux attentes du cahier des charges. Tout n'a pas pu être réalisé. Certaines informations demandées dans le cahier des charges comme le taux de charge des stations ne sont pas récupérables.

L'équipe s'est investie, organisée et a produit un livrable valide. Lorsque le programme est lancé, il n'y a pas de problème visible. Ce dernier a été rendu dans les délais et respecte le sujet.

Toutefois, et à l'avenir, une gestion plus rigoureuse du temps disponible et un investissement plus régulier seront nécessaires pour éviter d'être surpris par les quelconques problèmes.

Par ailleurs, on peut aussi noter l'utilisation adéquate des documents vus et introduits au cours du MOOC GdP comme la matrice SWOT ou le Work Breakdown Structure. Contrairement à l'ancien projet, les méthodes agiles n'ont pas été utilisées pour savoir ce qui convenait le mieux à notre équipe. Mais comme nous le pensions lors du dernier projet, les méthodes agiles sont ce qui est le plus cohérents au vu de notre environnement de travail et de notre équipe.