



# Rapport de Projet Pluridisciplinaire d'Informatique Intégrative 2

Hélène Barbillon  
Aurélien Gindre  
Adrien Larousse  
Mathis Mangold

Mars 2023 - Mai 2023

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>État de l’art</b>	<b>5</b>
2.1	État de l’art . . . . .	5
2.2	Algorithmes de recherches d’itinéraires . . . . .	5
<b>3</b>	<b>Calcul d’itinéraire</b>	<b>6</b>
3.1	Objectif . . . . .	6
3.2	Genèse de l’algorithme . . . . .	6
3.3	Gestion des données . . . . .	6
3.3.1	Traitement des données des bornes . . . . .	6
3.3.2	Traitement des données des voitures . . . . .	6
3.4	Construction de l’algorithme principal . . . . .	6
3.4.1	Principe . . . . .	7
3.4.2	Implémentation de l’algorithme . . . . .	7
3.5	Paramètres et contraintes supplémentaires . . . . .	9
3.5.1	Traitement des données . . . . .	9
3.5.2	Implémentation . . . . .	9
3.6	Problématique du temps . . . . .	9
<b>4</b>	<b>Simulation</b>	<b>11</b>
4.1	Objectif . . . . .	11
4.2	Génération aléatoire de trajets . . . . .	11
4.2.1	Données . . . . .	11
4.2.2	Génération aléatoire . . . . .	11
4.3	Exécutions parallèles de l’algorithme . . . . .	12
4.3.1	Le multithreading . . . . .	12
4.3.2	Le fork . . . . .	12
4.4	Structures et fonctionnement . . . . .	13
4.4.1	Structures . . . . .	13
4.4.2	Fonctionnement . . . . .	13
4.4.3	Export . . . . .	16
<b>5</b>	<b>Interface graphique</b>	<b>17</b>
5.1	Objectifs . . . . .	17
5.2	Technologies utilisées . . . . .	17
5.3	Structure de l’application . . . . .	17
5.4	Fonctionnement de l’application . . . . .	18
5.4.1	Côté calcul d’itinéraire . . . . .	18
5.4.2	Côté simulation . . . . .	18
<b>6</b>	<b>Tests et performances des fonctions</b>	<b>19</b>
6.1	Complexité théorique des algorithmes . . . . .	19
6.1.1	Complexité de l’algorithme de recherche d’itinéraire . . . . .	19
6.1.2	Complexité de l’algorithme de génération aléatoire d’itinéraire . . . . .	19
6.1.3	Complexité de l’algorithme de simulation . . . . .	19
6.2	Tests unitaires et performance . . . . .	20
6.2.1	Cadre . . . . .	20
6.2.2	Performance de l’algorithme de recherche d’itinéraire . . . . .	20
6.2.3	Génération aléatoire d’itinéraire . . . . .	21

6.2.4	Multithreading et forks . . . . .	21
6.2.5	Performances de l'algorithme de génération multiple de trajets . . . . .	21
6.2.6	Tests unitaires . . . . .	23
<b>7</b>	<b>Gestion de projet</b>	<b>26</b>
7.1	Charte de projet . . . . .	26
7.1.1	1. Cadrage . . . . .	26
7.1.2	2. Déroulement . . . . .	27
7.2	Communication . . . . .	27
7.3	Analyse SWOT du projet . . . . .	28
7.4	WBS . . . . .	29
<b>8</b>	<b>Conclusion</b>	<b>30</b>
8.1	Retour sur les indicateurs de succès . . . . .	30
8.2	Pistes d'améliorations . . . . .	30
8.2.1	Optimisation de la recherche de trajet . . . . .	30
8.2.2	Génération de multiples trajets . . . . .	30
8.3	Bilan global . . . . .	30
<b>9</b>	<b>Bilans personnels</b>	<b>31</b>
9.1	Aurélien GINDRE . . . . .	31
9.2	Adrien LAROUSSE . . . . .	31
9.3	Hélène BARBILLON . . . . .	32
9.4	Mathis MANGOLD . . . . .	32
<b>10</b>	<b>Annexes</b>	<b>33</b>

# Table des figures

3.1	Fonctionnement de l'algorithme . . . . .	7
3.2	Nouvelle méthode de sélection de borne pour privilégier le temps de parcours . . . . .	10
4.1	Structure d'un passage . . . . .	13
4.2	Exemple de l'état initial lors d'un ajout à la liste . . . . .	14
4.3	Exemple d'un ajout à la liste . . . . .	14
4.4	Exemple d'une voiture devant être en attente à son arrivée (le rouge correspond aux éléments considérés et le vert à ceux ajoutés et qui n'existaient pas) . . . . .	14
4.5	Exemple d'une voiture ultérieure devant passer en attente . . . . .	15
4.6	Exemple d'une sortie devant être ajoutée . . . . .	15
4.7	Suppression d'un ancien élément et mise à jour du nouveau . . . . .	16
5.1	Structure de l'application . . . . .	17
5.2	Formulaire utilisateur . . . . .	18
5.3	Affichage de la simulation . . . . .	18
6.1	Performances de l'algorithme de recherche d'itinéraire . . . . .	20
6.2	Performances de l'algorithme de génération aléatoire d'itinéraire . . . . .	21
6.3	Évolution du temps d'exécution en fonction du nombre de trajets générés (pour 10 forks) . . . . .	22
6.4	Évolution du temps d'exécution en fonction du nombre de trajets générés (pour 20 forks) . . . . .	22
6.5	Évolution du temps d'exécution en fonction du nombre de trajets générés (pour 50 forks) . . . . .	23
6.6	Évolution du temps d'exécution en fonction du nombre de trajets générés (pour 75 forks) . . . . .	23
6.7	Tests unitaires du module random points . . . . .	24
6.8	Tests unitaires du module vehicules . . . . .	24
6.9	Tests unitaires du module bornes . . . . .	25
7.1	Critères et indicateurs de succès . . . . .	26
7.2	Matrice SWOT . . . . .	28
7.3	Décomposition des tâches du projet . . . . .	29
7.4	Ordre de priorité des tâches du projet . . . . .	29

# Chapitre 1

## Introduction

Ce rapport présente les recherches et les résultats du Projet Pluridisciplinaire d'Informatique Intégrative (PP2I) réalisé dans le cadre du second semestre de la première année de notre scolarité à Telecom Nancy.

Le projet a pour but de fournir aux usagers un ensemble de fonctions qui les aident dans le déploiement et le dimensionnement d'un réseau de recharge de véhicules adapté à leurs besoins. Le but réel de ce projet est de nous faire travailler en groupe et d'affiner nos connaissances et notre expérience de la programmation en C. Il est en effet également imposé que le développement de l'algorithme principal se fasse en C. Le sujet imposé traite, quant à lui, de la problématique des déplacements en voiture électrique, avec notamment les contraintes liées à la recharge des batteries.

Ce projet se décompose en deux parties complémentaires. La première est la réalisation d'un programme permettant à un usager souhaitant se rendre d'un point A à un point B un parcours de charge de son véhicule électrique. De plus, certaines fonctionnalités supplémentaires sont attendues afin d'anticiper des besoins plus spécifiques de l'utilisateur.

La deuxième partie consiste en un mode de simulation. Celui-ci a pour objectif, comme son nom l'indique, de simuler les trajets d'un certain nombre d'usagers se déplaçant simultanément. L'objectif est ici de visualiser l'occupation des bornes de recharge du territoire et de modéliser la charge que ces usagers représentent sur le réseau.

# Chapitre 2

## État de l'art

Ce chapitre présente les recherches que nous avons faites en amont du projet, à propos des solutions existantes répondant à notre problématique.

### 2.1 État de l'art

Ces dernières années, les véhicules électriques ont connu une croissance exponentielle et se sont imposés comme une alternative plus durable et écologique aux véhicules à combustion interne. Le nombre de bornes de recharge pour véhicules électriques a explosé en France, passant de 32 000 bornes en 2020, à plus de 82 000 en 2022. Le palier des 100 000 bornes en France a ainsi été atteint au début du mois de mai 2023. Des solutions ont ainsi vu le jour pour répondre aux problématiques de recharges de voitures électriques, l'une d'elles est Chargemap.

Chargemap est une application qui, pour 20 €, planifie vos trajets en voiture électrique en fonction du modèle de voiture et du temps de pause souhaité à chaque borne. Chargemap permet également de s'arrêter dans des lieux touristiques pour se recharger (afin de pouvoir profiter du temps d'attente pour visiter), et possède de nombreuses autres fonctionnalités avancées.

Dans le cadre du projet, nous allons également réaliser un outil de création d'itinéraires intelligent pour les propriétaires de voitures électriques, avec quelques fonctionnalités supplémentaires similaires (tel que le temps maximal de charge). La différence notable entre notre projet et Chargemap est la partie estimation de la surcharge sur le réseau. Une autre application, Nextcharge, montre quant à elle bel et bien à l'utilisateur si une borne est libre, occupée, ou en maintenance. Mais dans cette application, les informations sont en temps réel, sans estimation dans le temps, et le système d'itinéraire reste limité.

Notre projet a donc pour but de regrouper création d'itinéraire (avec quelques fonctionnalités adjacentes) et estimation de la charge sur le réseau. Ces deux axes principaux sont interdépendants, et c'est ainsi que l'on pourra suggérer un trajet différent à un utilisateur en fonction de la disponibilité de certaines bornes de recharge.

### 2.2 Algorithmes de recherches d'itinéraires

Nos recherches nous ont menées à deux algorithmes de recherches de plus courts chemins assez connus : l'algorithme de Dijkstra vu en cours, et l'algorithme A\* (A-star ou A-étoile).

L'algorithme de Dijkstra effectue une recherche en largeur en explorant tous les chemins possibles du noeud de départ à chaque autre noeud, sans prendre en compte un objectif spécifique à atteindre. Tandis que le A\* est plus optimisé et va prendre en compte la destination pour éviter d'explorer des chemins moins prometteurs.

Dans ce projet, nous connaissons la destination finale et la distance entre chaque borne. L'algorithme A\* est donc le plus pertinent dans ce cas.

## Chapitre 3

# Calcul d'itinéraire

### 3.1 Objectif

La première partie du projet consiste en un programme permettant de calculer un itinéraire d'un point A à un point B en voiture électrique. La contrainte majeure est ici l'autonomie du véhicule. De plus, l'algorithme devra être modulable pour répondre à différentes contraintes : temps de recharge minimal et seuil de décharge minimal.

### 3.2 Genèse de l'algorithme

Pour répondre au mieux à la problématique, nous avons fait de nombreuses recherches dont les résultats ont été présentés dans la partie "État de l'art". Nos observations nous ont menées à l'algorithme de recherche A\*. En effet, l'algorithme A\* combine pertinence des résultats avec rapidité d'exécution. Dans le pire des cas, lorsque chaque noeud doit être visité pour trouver la solution, sa complexité est exponentielle. Mais, dans la majorité des configurations, la complexité du A\* sera très convenable, et peut même tendre à être linéaire dans le meilleur des cas. C'est pour cela que nous nous sommes tournés vers cet algorithme pour notre projet.

### 3.3 Gestion des données

#### 3.3.1 Traitement des données des bornes

Nous avons fait le choix d'importer la base de données gouvernementale dans une base de données SQL, après avoir échoué à stocker les données dans une matrice d'adjacence. Cela nous permet de faciliter par la suite l'accès aux données grâce à des requêtes précises. Nous avons également fait 2 changements majeurs sur la structure de la base de données.

La base de données gouvernementale contenait beaucoup de doublons. En effet, s'il y a 10 bornes de recharge à un même endroit, il y avait 10 fois la même ligne dans la base de données. Nous avons fait le choix d'éliminer les doublons et d'ajouter une colonne `nbr_pt_recharge` qui dénombre les bornes présentes à un même endroit. La base de données est ainsi passée de 55320 à 17319 valeurs.

Ensuite, nous avons ajouté une colonne `primary_key` nommée `id_unique` (contenant des entiers de 1 à 17319). Nous pouvons ainsi identifier un point de recharge par cet entier unique, ce qui facilite l'accès aux données d'une borne précise.

#### 3.3.2 Traitement des données des voitures

Dans la même base de données SQL, nous avons créé une table `vehicules` qui comporte l'ensemble des véhicules du site donné dans le sujet, avec leur nom, leur autonomie en km, et la puissance de leur batterie en Wh. Les véhicules sont également identifiés par un entier unique `id`. L'importation des données depuis le site a été faite avec un script python qui récupère et traite l'HTML de la page.

### 3.4 Construction de l'algorithme principal

En nous basant donc sur l'algorithme A\*, nous avons implémenté un algorithme qui cherche à chaque étape le point le plus proche de l'arrivée, en tenant compte de la contrainte d'autonomie du véhicule.

### 3.4.1 Principe

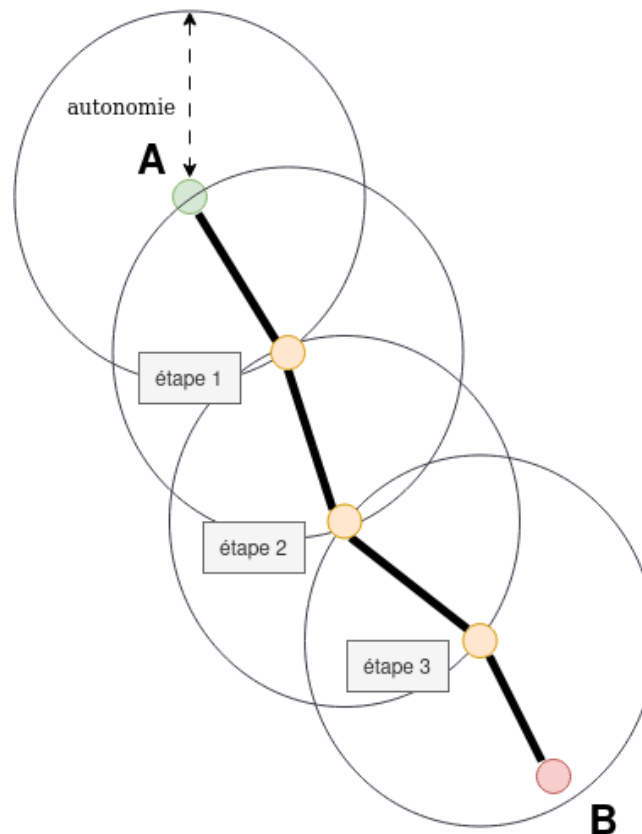


FIGURE 3.1 – Fonctionnement de l'algorithme

Comme on peut le voir sur le schéma ci-dessus, l'idée pour aller d'un point A à un point B est, à partir du point A, de trouver le point le plus proche de B dans le cercle de centre A et de rayon égal à l'autonomie du véhicule. Ce point constitue notre premier arrêt pour aller à B. On considère maintenant ce point comme le nouveau départ, et on applique la même logique : on cherche le point situé le plus proche de B dans le cercle ayant comme centre cette étape et comme rayon l'autonomie du véhicule. On se rapproche de l'arrivée et on construit ainsi la liste des points qui permettent d'atteindre B. L'algorithme s'arrête quand le point B est dans le cercle correspondant au dernier point.

### 3.4.2 Implémentation de l'algorithme

Maintenant que nous avons vu le principe de l'algorithme, il nous faut l'implémenter.

#### Structures de données utilisées

Nous avons besoin de 6 structures de données pour implémenter cette algorithme :

- **coord\_pt** déclarée dans le fichier d'en-tête **coordinates.h**. Cette structure permet de stocker un point en fonction de sa latitude (**long double**) et sa longitude (**long double**).
- **borne** déclarée dans le fichier d'en-tête **borne.h**. Cette structure permet de stocker les informations relatives à une borne :
  - ses coordonnées : **coord\_pt**
  - son nom : **char\***
  - son identifiant : **int**
  - sa puissance nominale : **int**
- **borne\_and\_distance** déclarée dans le fichier d'en-tête **borne.h**. Cette structure permet de connaître la distance entre un point A et la borne, et la distance entre la borne et un point B. Elle contient les informations suivantes :
  - les informations de la borne : **borne**
  - son tick de recharge : **int**, c'est une information qui sert pour la simulation
  - sa distance avec un point A : **double**



- sa distance avec un point B : `double`
- `position_point` déclarée dans le fichier d'en-tête `itinerary.h` qui se base sur `borne_and_distance` mais au lieu de stocker toutes les informations de la borne (`borne`), on stocke uniquement l'identifiant associé (`int`)
- `list_position` déclarée dans le fichier d'en-tête `itinerary.h` qui permet de créer une liste chaînée de type `position_point`.
- `etape` déclarée dans le fichier d'en-tête `etape.h`. Elle permet de stocker la liste des étapes pour aller d'un point de départ à un point d'arrivée. Elle contient la liste des étapes qui est un pointeur de `borne_and_distance` et entier `size` qui représente le nombre d'étapes.

## Manipulation des coordonnées

Pour pouvoir manipuler les coordonnées dans notre projet nous avons implémenté le module `coordinates.h`. En plus du type `coord_pt` vu précédemment, il ajoute une fonction `distance`. Elle permet de calculer la distance entre 2 points, on utilise la méthode des sinus trouvée sur le site : [Distance entre deux points géographiques](#). La formule qui permet de calculer la distance entre deux points est :

$$d = 6371 \times \arccos(\sin(\phi_A)\sin(\phi_B) + \cos(\phi_A)\cos(\phi_B)\cos(\lambda_B - \lambda_A))$$

avec  $\phi$  la latitude et  $\lambda$  la longitude.

Enfin, la précision pour calculer des distances étant importante, les coordonnées GPS sont stockées dans des `long double` et non des `double`. Les `long double` utilisent 80 bits pour être stockés contre 64 bits pour les `double`.

## Fonctions de l'algorithme

Maintenant que nous avons un cadre et des structures de données à utiliser, nous pouvons regarder l'écriture de l'algorithme. On trouve l'algorithme dans le fichier `itinerary.c`. L'algorithme est décomposé en 3 fonctions principales :

- `getBorneFromDistance(depart, arrivee)` : Elle permet de calculer pour toutes les bornes : la distance entre le départ et la borne, la distance entre la borne et l'arrivée. Les résultats sont stockés dans une liste chaînée de type `list_position`
- `plus_proche(list_position, liste_bornes_visitee, autonomie)` : Elle reçoit en paramètre une liste chaînée de type `list_position`, une autonomie, et une liste des bornes déjà visitées (liste de bornes à exclure du résultat). Elle permet de trouver la borne la plus proche de l'arrivée. Elle stocke le résultat dans `borne_and_distance` afin d'avoir toutes les informations sur la borne.
- `get_liste_etape_itineaire_type_distance(depart, arrivee, voiture)` : Elle permet de renvoyer une liste de type `etape*`. Elle combine l'utilisation des deux fonctions précédentes. C'est elle qui contient l'algorithme.

## Écriture de l'algorithme

Voici la description de l'algorithme tel qu'il est implémenté dans `itinerary.c` dans la fonction `get_liste_etape_itineaire_type_distance(depart, arrivee, voiture)`.

Initialisation des paramètres :

```
borne_and_distance depart = depart;
borne_and_distance arrivee = arrivee;
liste_etape = etape_create();
```

Condition d'arrêt : Distance entre l'arrivée et le départ inférieure à l'autonomie.

Cas de base : La fonction `getBorneFromDistance(depart, arrivee)` est vide.

Itération :

```
distance_etape_i = getBorneFromDistance(depart, arrivee);
borne_selectionner = plus_proche(distance_etape_i, autonomie);
depart = borne_selectionner;
etape_add(liste_etape, depart);
```

Retour de la fonction : `liste_etape`.

## 3.5 Paramètres et contraintes supplémentaires

Nous avons implémenté 3 paramètres d'entrées supplémentaires. Tout d'abord, l'utilisateur peut indiquer l'autonomie initiale de sa voiture. Ensuite, il lui est possible de spécifier un pourcentage de charge minimal qu'il souhaite garder tout le temps en réserve. Pour finir, l'utilisateur peut choisir de recharger sa voiture pendant une durée qu'il définit.

### 3.5.1 Traitement des données

Un dernier traitement sur la base de données a été nécessaire. En effet, les valeurs de puissance nominales des bornes se sont révélées être pour certaines en W et pour d'autres en kW. Les valeurs ont toutes été converties en W, afin de pouvoir faire des calculs avec les puissances des voitures en Wh.

Une autre problématique a été l'absence de certaines données. Une centaine de bornes s'avérait avoir une puissance nominale non spécifiée (nulle). Afin de ne pas se résoudre à complètement omettre ces bornes dans notre calcul d'itinéraire, nous avons fait le choix de définir arbitrairement leur puissance nominale à la moyenne des puissances des autres bornes.

### 3.5.2 Implémentation

D'un point de vue algorithmique, il a fallu créer puis enrichir la structure `voiture`. Voici cette structure :

```
1 typedef struct voiture
2 {
3     char* nom;
4     int id;
5     double autonomie;
6     double autonomie_actuelle; // autonomie que je peux utiliser pour voyager = ON NE PREND
    PAS EN COMPTE LA BATTERIE EN RESERVE
7     double reserve_equivalent_autonomie; // équivalent en km du pourcentage minimum de
    batterie voulu en réserve
8     int puissance;
9     int puissance_actuelle; // puissance TOTALE disponible = ON PREND EN COMPTE LA BATTERIE
    EN RESERVE
10    int temps_recharge_max_minutes; // 0 si pas de temps spécifié
11 } voiture;
```

On dispose alors d'informations propres à la voiture telles que les caractéristiques de puissance et d'autonomie. D'autres valeurs sont des constantes établies au début de l'algorithme en fonction des paramètres d'entrée : `reserve_equivalent_autonomie` et `temps_recharge_max_minutes`. Ainsi, on peut utiliser la valeur `autonomie_actuelle` dans l'algorithme, lorsqu'il s'agit de savoir si une borne est accessible en conservant la réserve de batterie définie par l'utilisateur.

D'un autre côté, pour la contrainte du temps de recharge, il a fallu adapter très légèrement le programme en appelant une fonction `recharge` lorsque l'algorithme sélectionne une borne optimale pour s'arrêter. Cette fonction permet de simuler le fait que la voiture se recharge à une certaine borne pendant un certain temps. Les variables `autonomie_actuelle` et `puissance_actuelle` sont donc mises à jour et l'algorithme peut entamer la boucle suivante avec ces nouvelles valeurs.

## 3.6 Problématique du temps

Notre algorithme dérivé du A\* permet actuellement à l'utilisateur de récupérer un itinéraire optimisé sur la distance. Or, en voiture électrique, le facteur du temps de recharge est un facteur clé dans la rapidité de déplacement sur de grandes distances. En effet, une charge complète d'un véhicule électrique peut prendre jusqu'à une demi-journée. De plus, il est maintenant possible de fixer les temps de recharge dans les paramètres d'entrée du programme, et se recharger 10 minutes sur une borne de 3'000 W n'est pas équivalent à se recharger 10 minutes sur une borne de 400'000 W. Nous avons donc décidé de travailler sur une version de l'algorithme qui propose des itinéraires optimisés en temps.

L'algorithme optimisé en temps fonctionne globalement de la même manière que le premier algorithme dérivé du A\*. Le seul changement réside dans la sélection de la borne. Pour chaque borne, on simule quelle serait l'autonomie de notre voiture si on choisissait de s'y recharger. Puis on sélectionne la borne qui permettrait de

se rapprocher le plus de l'arrivée après s'y être rechargé (la plus petite valeur  $\text{distance\_borne\_arrivée} - \text{autonomie\_simulée\_après\_recharge}$ ).

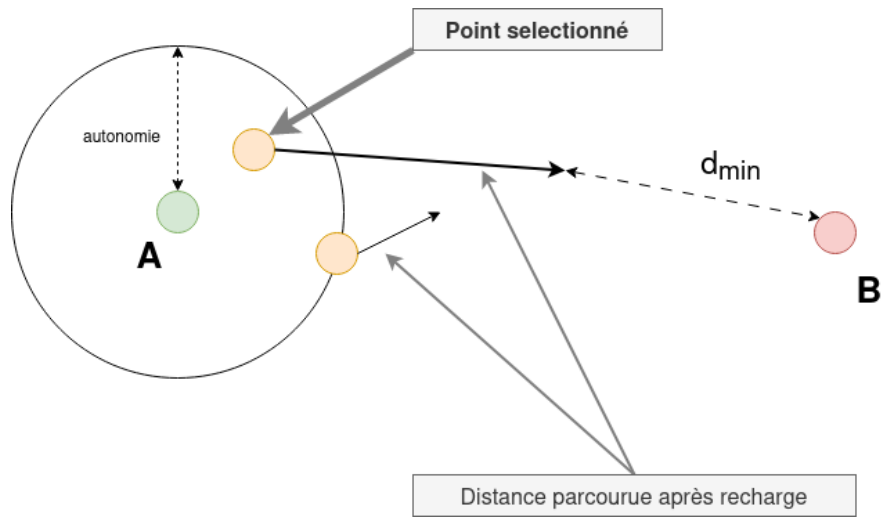


FIGURE 3.2 – Nouvelle méthode de sélection de borne pour privilégier le temps de parcours

# Chapitre 4

## Simulation

### 4.1 Objectif

La seconde partie du projet consiste en la simulation d'un grand nombre de trajets, afin d'analyser l'impact et la surcharge sur le réseau. Le temps est discrétisé par ticks de 10 minutes, et les voitures sont supposées comme se déplaçant en ligne droite entre les bornes, à une vitesse moyenne de 100km/h.

### 4.2 Génération aléatoire de trajets

Pour pouvoir lancer des simulations, nous avons implémenté une fonction pour générer des trajets aléatoirement en France métropolitaine.

#### 4.2.1 Données

Nous avons utilisé une base de données du gouvernement regroupant toutes les communes de France avec les coordonnées correspondantes (`app/data/bd_villes_france.csv`). Chaque ligne contient :

- Le code commune INSEE
- Le nom de la commune
- Le code postal
- La ligne 5 de l'adresse (précision de l'ancienne commune ou du lieu-dit)
- Le libellé d'acheminement
- Les coordonnées géographiques (latitude, longitude)

Nous avons supprimé toutes les lignes qui concernaient des départements d'Outre-mer à l'aide d'une expression régulière. Un autre problème rencontré avec cette base de données a été l'absence de certaines données. Certains champs étaient tout simplement vides, ce qui posait problème lors de la récupération des coordonnées par la commande `sscanf`. Cependant ce problème ne concernait jamais les coordonnées (la donnée qui nous intéressait dans ce cas), nous avons donc simplement rempli ces champs vides arbitrairement par `abc` afin de contourner le problème.

#### 4.2.2 Génération aléatoire

Nous choisissons le nombre de trajets que nous voulons générer à chaque fois, ils sont stockés dans cette structure :

```
1 typedef struct trajets_aleatoires{
2     int taille; //nb de trajets aléatoires qu'on veut générer
3     trajet* traj;
4 }trajets_aleatoires;
```

La fonction de génération de trajets aléatoire choisit deux lignes au hasard dans la base de données et vérifie si la distance entre les 2 villes choisies est bien supérieure à la distance minimale voulue. Pour ne pas avoir de

boucles infinies si la distance minimale choisie est trop élevée, nous avons limité le nombre de comparaisons entre les deux distances à 50. Elle stocke ensuite un certain nombre d'informations pour chaque trajets, certains paramètres de simulations étant eux-mêmes choisis au hasard.

```
1 typedef struct trajet{
2     coord_pt* depart;
3     coord_pt* arrivee;
4     long double distance_traj;
5     int id_voiture;
6     int pourcentage_mini_voulu;
7     int temps_max_attente_borne;
8     int type;
9     int pourcentage_autonomie_initiale;
10    int thread;
11 }trajet;
```

## 4.3 Exécutions parallèles de l'algorithme

Le premier problème pour la partie de simulation a été de trouver le moyen d'exécuter un grand nombre de fois notre fonction de création d'itinéraires sans que le temps d'exécution ne devienne trop élevé. Nous avons opté pour une solution très simple en apparence : lancer l'algorithme plusieurs fois en parallèle.

### 4.3.1 Le multithreading

Le multithreading permet d'exécuter une séquence d'instructions plusieurs fois dans le même processus. C'est la solution que nous avons choisie dans un premier temps pour notre projet, en exécutant jusqu'à 20 threads simultanément. Ainsi, générer 100 trajets est équivalent à 5 boucles de 20 threads qui gèrent chacun un trajet.

### 4.3.2 Le fork

Une seconde solution qui a été évoquée a été de créer des forks de l'algorithme d'itinéraire. Le fork consiste en la création d'un processus fils, créant une copie du processus appelant (code, données et état copiés). Ainsi c'est le même code qui s'exécute pour tous les forks. On peut identifier un processus fils grâce à son PID. Chaque processus s'exécute séparément et a donc ses propres ressources et son propre flux d'exécution. Une fois ce mécanisme de multitâche lancé, il suffit d'attendre dans le processus parent que les forks soient finis et l'on peut alors récupérer les valeurs de retour. Chaque processus ayant son propre flux et ses propres ressources, l'exécution devient plus rapide que si l'on devait lancer uns à uns les algorithmes.

Contrairement à un thread, un fork possède ses propres ressources et ses propres adressages mémoire. Cette différence rend l'utilisation des forks plus avantageuse. En effet, le but est de diviser l'exécution d'un algorithme assez gourmand en ressources. Utiliser des threads n'alloue pas des ressources pour chaque algorithme qui se lance, mais laisse tout s'exécuter dans un seul processus. C'est pour cela que ce n'est pas énormément rentable en termes de rapidité d'exécution, et que nous nous sommes tournés vers les forks.

Il a fallu également gérer les informations de retour des différents forks. Une première possibilité aurait été d'utiliser des pipes. Un pipe consiste en la définition de 2 file descriptors, un pour la lecture du pipe, et l'autre pour l'écriture. Ces deux clés abstraites vers un même fichier sont définies avant le lancement de la commande **fork**, afin que les 2 processus les aient en mémoire. Ensuite, chaque processus utilise ces descripteurs de fichiers et les ferme quand il n'en a plus l'usage. Par exemple, dans notre cas, le processus fils peut fermer dès le début le descripteur **READ**, exécuter l'algorithme d'itinéraire, rentrer les valeurs de retours dans le pipe, puis enfin fermer le **WRITE**. Et le processus parent peut, quant à lui, directement fermer son descripteur **WRITE**, puis attendre que le processus fils ferme son **WRITE** afin de récupérer toutes les informations de retour avec son propre descripteur **READ**.

Dans notre cas, cette solution reviendrait à faire traiter énormément de valeurs de retour à la fonction parent. En effet, il faudrait alors traiter chacun des 100, voire 1000 itinéraires un à un quand chaque fork a fini l'exécution de l'algorithme. Et la rapidité du programme serait fortement impactée.

Dans l'outil de simulation, le calcul des itinéraires est simplement la première étape afin d'avoir des entrées pour les fonctions suivantes. Il n'y a donc pas besoin de traiter particulièrement les valeurs de retour de l'algorithme dans chaque fork. C'est pourquoi nous avons choisi de faire gérer à chaque fork l'exportation de son propre itinéraire calculé. Et cette exportation se fait en étendant un seul et même fichier `app/data/forks.txt` d'une

ligne. Les données exportées et leur format d'export ont été choisis pour correspondre au mieux aux besoins des fonctions de simulation.

## 4.4 Structures et fonctionnement

La simulation consiste à faire simultanément plusieurs trajets et à les représenter sur une carte par tranches de 10 min afin de connaître l'état des différentes bornes de France.

Le plus grand problème de la simulation est la manière d'aborder la gestion de multiples trajets simultanément. Il est nécessaire de choisir des structures adaptées au problème afin d'avoir un outil de simulation intéressant. En effet, une bonne simulation repose sur des données pertinentes et nombreuses. Il est par ailleurs également important de se représenter, lors de la conception, la manière dont vont être utilisées les structures, afin de les moduler de sorte à ce qu'elles soient facilement manipulables après.

### 4.4.1 Structures

Nous avons appelé chaque tranche de 10 min un "tick", ainsi 1h correspond à 6 ticks. Nous avons remarqué que très peu de trajets durent plus de 13h environ et avons ainsi décidé de limiter la simulation à 24h, soit 144 ticks.

Pour la simulation, il est nécessaire de connaître l'état de toutes les bornes à chaque tick. Étant donné cette contrainte, il est plus intéressant d'utiliser un tableau pour représenter l'ensemble des bornes, permettant ainsi d'accéder à chacune d'entre elles à un coût constant.

Lors de la simulation, nous ne connaissons pas au préalable comment chaque borne sera utilisée, il est donc préférable d'utiliser une structure dynamique plutôt qu'une structure statique pour représenter l'évolution des bornes au fil des ticks. À chaque borne est attribuée une liste des "passages" des voitures. Chaque passage est composé de l'identifiant d'un véhicule, du nombre de places restantes à la borne après avoir considéré son passage, du tick du passage, d'un pointeur vers le prochain passage (étant donné que c'est une structure de type liste), et d'un statut de passage : 0 -> le véhicule quitte la borne, 1 -> le véhicule arrive à la borne, 2 -> le véhicule est en attente à la borne.

id véhicule	statut passage	places restantes	tick	pointeur prochain passage
----------------	-------------------	---------------------	------	------------------------------

FIGURE 4.1 – Structure d'un passage

Ainsi, lorsqu'une voiture arrive à une borne, elle entraîne l'ajout d'au moins 2 "passages" à la liste de cette borne : celui d'arrivée et celui de sortie. Il arrive qu'un troisième passage soit ajouté, celui où le véhicule est en attente, mais une liste de passages ne comportera jamais plus de trois passages d'un même véhicule.

Pour éviter de parcourir l'entièreté de la liste des passages pour trouver un tick particulier, la liste doit être ordonnée (selon les ticks croissants). À chaque ajout, l'ordre doit être respecté.

Le nombre de places restantes dans un passage permet de connaître l'état de la borne après le passage considéré. Ainsi, il est possible de savoir si un véhicule arrivant au tick suivant est en attente, ou s'il peut directement se recharger.

### 4.4.2 Fonctionnement

L'outil de simulation va étudier les trajets un par un et modifier la liste des passages déjà établie pour ajouter les véhicules.

Pour chaque étape d'un trajet, nous connaissons la borne où le véhicule arrive, son tick d'arrivée, et le nombre de ticks pendant lesquels le véhicule se recharge (en fonction des critères du trajet).

On ajoute alors à la liste des passages de la borne notre véhicule au travers de la fonction `ajout_passage`.

La fonction `ajout_passage` comporte une file d'attente des éléments à ajouter. Ainsi, si la liste était amenée à changer, des éléments seraient ajoutés à la fin de la file d'attente.

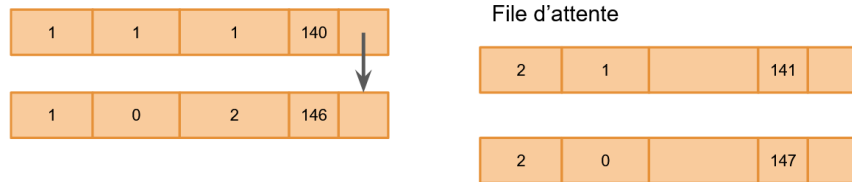


FIGURE 4.2 – Exemple de l'état initial lors d'un ajout à la liste

On étudie chaque élément pour savoir s'il peut être ajouté à la liste en respectant l'ordre, et si c'est le cas, on l'insère dans la liste.

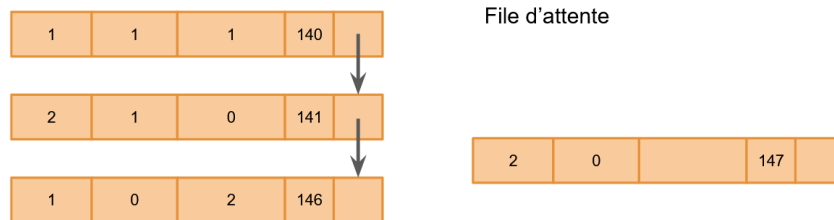


FIGURE 4.3 – Exemple d'un ajout à la liste

Il existe différents cas pouvant poser des problèmes. Le premier étant si un véhicule est censé arriver mais qu'il n'y a plus de places disponibles, dans ce cas il est en attente et attend qu'une place se libère. Le statut du passage devient alors "3" (utile seulement lors du traitement des passages), indiquant que ce passage doit être ajouté dès qu'une place se libère. Un passage indiquant que le véhicule est en attente est alors ajouté à son tick d'arrivée.

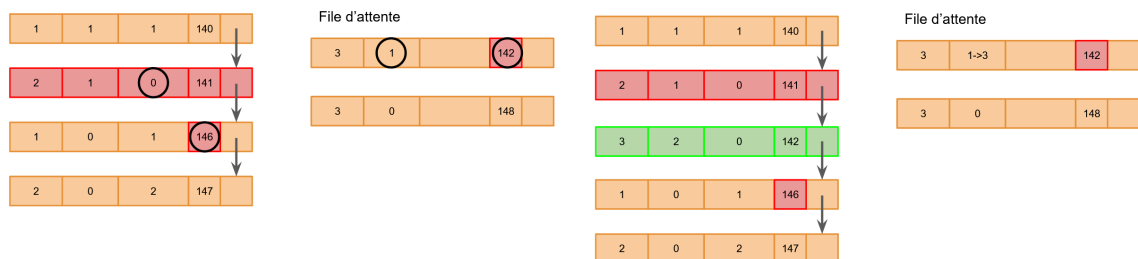


FIGURE 4.4 – Exemple d'une voiture devant être en attente à son arrivée (le rouge correspond aux éléments considérés et le vert à ceux ajoutés et qui n'existaient pas)

Un second cas serait lorsqu'un véhicule est ajouté à la liste des passages et qu'un véhicule qui arrive plus tard doit se mettre en attente. Dans ce cas un passage est ajouté à la file d'attente et le statut du passage devient 2 au lieu de 1.

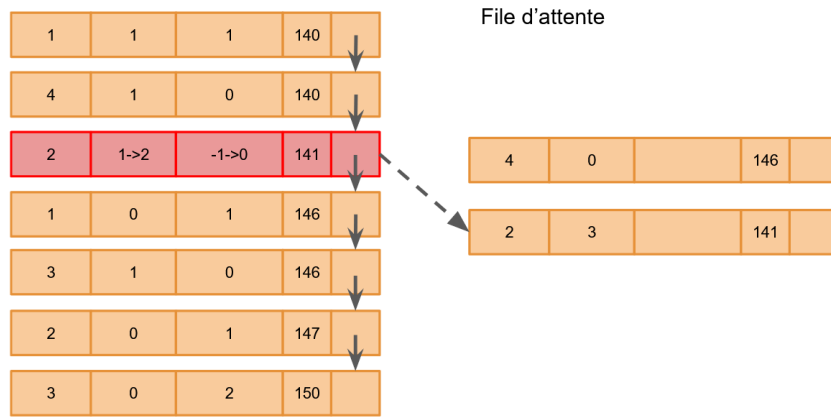


FIGURE 4.5 – Exemple d'une voiture ultérieure devant passer en attente

Le problème de ce second cas est que le "passage" de sortie du véhicule devant passer en attente n'est pas dans la file d'attente. Il doit donc être trouvé et mis à jour. Un autre problème est que le temps de charge du véhicule n'est plus connu, car il a été traité précédemment. On se sert alors du tick comme donnée de stockage. On stocke le décalage entre le tick d'arrivée initial et le nouveau tick d'arrivée, et on l'ajoute plus tard au tick de sortie.

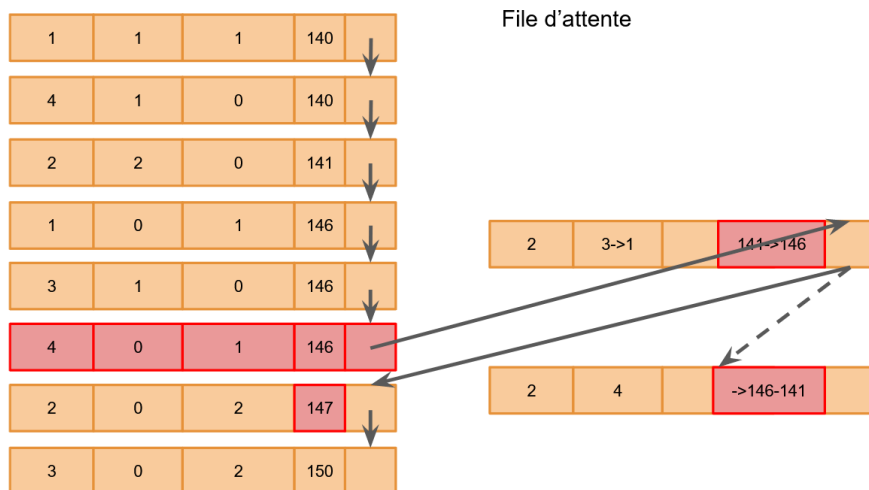


FIGURE 4.6 – Exemple d'une sortie devant être ajoutée



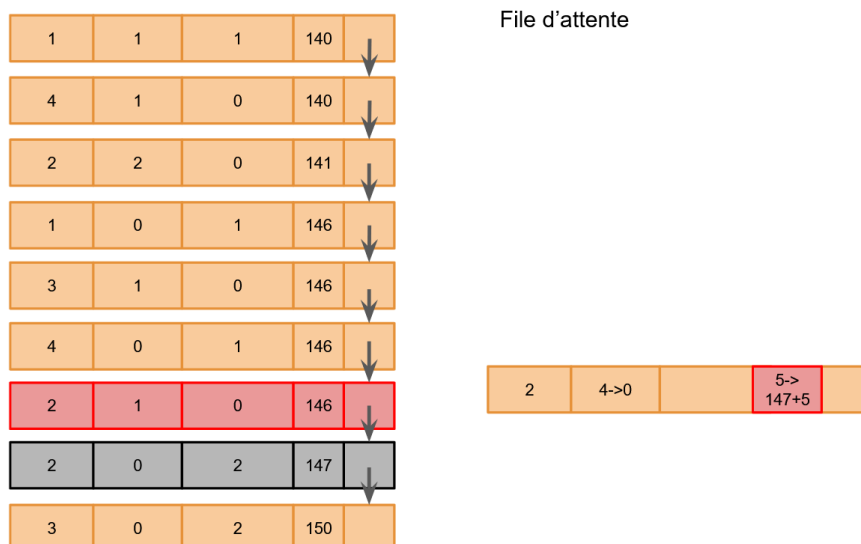


FIGURE 4.7 – Suppression d'un ancien élément et mise à jour du nouveau

### 4.4.3 Export

Une fois l'ensemble des trajets traités et l'ensemble des listes de passages obtenues, il faut les convertir en une autre forme plus adaptée à l'affichage. En effet, on souhaite regarder à chaque tick l'évolution des bornes, or chaque borne possède sa propre liste de passages, il faudrait donc à chaque tick parcourir chaque borne pour trouver les passages correspondants à ce tick.

Il est alors préférable de n'effectuer cette tâche de segmentation qu'une unique fois et de stocker le résultat. C'est à double tranchant : on devient plus efficace, mais on utilise davantage de stockage. On crée alors un tableau de tous les ticks (144), et on attribue à chaque tick une liste des bornes n'étant pas vides à ce tick.

# Chapitre 5

## Interface graphique

### 5.1 Objectifs

Les objectifs liés à la réalisation d'une interface graphique sont multiples. L'interface permet premièrement de visualiser les résultats du calcul d'itinéraire et de la simulation. L'interface graphique permet aussi de contrôler les entrées utilisateurs.

### 5.2 Technologies utilisées

L'interface graphique fonctionne grâce à plusieurs éléments :

- Un serveur Flask python qui permet la gestion des routes et le lien entre les entrées utilisateurs et le programme C.
- Des scripts Javascript qui permettent l'affichage et la manipulation d'une carte.
- Une carte OpenLayers qui permet l'affichage d'une carte de qualité avec des fonctionnalités pré-implémentées.
- Du code HTML et CSS qui permet d'assurer la partie visuelle de l'application Web.

### 5.3 Structure de l'application

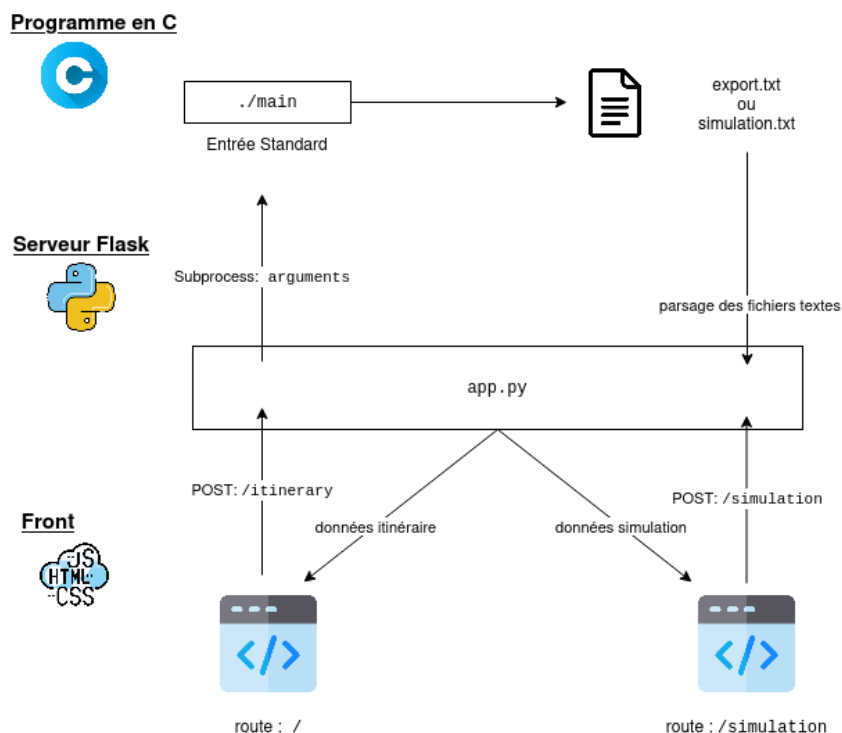
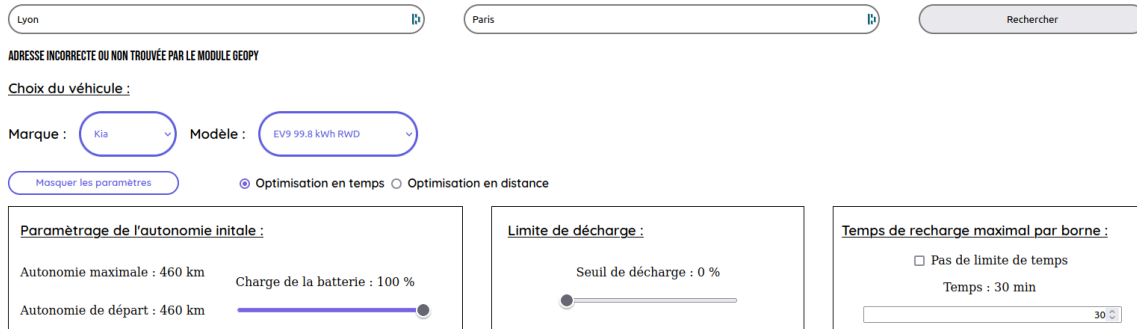


FIGURE 5.1 – Structure de l'application

## 5.4 Fonctionnement de l'application

### 5.4.1 Côté calcul d'itinéraire



The form is titled "Formulaire utilisateur". It has two input fields for addresses: "Lyon" and "Paris", each with a location icon. A "Rechercher" button is to the right. Below the addresses is a message: "ADRESSE INCORRECTE OU NON TROUVÉE PAR LE MODULE GEOPY". Under "Choix du véhicule :", there are dropdowns for "Marque" (Kia) and "Modèle" (EV9 99.8 kWh RWD). A "Masquer les paramètres" button is next. Below are three configuration panels:

- Paramétrage de l'autonomie initiale :** "Autonomie maximale : 460 km", "Charge de la batterie : 100 %", "Autonomie de départ : 460 km" with a slider.
- Limite de décharge :** "Seuil de décharge : 0 %" with a slider.
- Temps de recharge maximal par borne :** ☐ "Pas de limite de temps", "Temps : 30 min" with a slider.

FIGURE 5.2 – Formulaire utilisateur

Pour effectuer un calcul d'itinéraire, l'utilisateur doit rentrer une adresse postale (et non des coordonnées) pour le départ et l'arrivée. L'utilisateur peut choisir d'ajouter des options. Au moment d'appuyer sur le bouton **Rechercher**, une requête est envoyée au serveur Flask. De cette manière, tous les paramètres pour lancer le processus `main` sont récupérés. Grâce au module `Nominatim`, les adresses sont converties en coordonnées. La fonction `execute_app(args)` permet d'appeler le programme C qui écrit la liste des étapes dans un fichier texte `data/etape.txt`. Comme la fonction `subprocess.run` est bloquante, elle attend que le fichier texte soit édité. Enfin, une fois que le programme C est exécuté, on est redirigé sur la route / qui est là où s'affichent les résultats. La fonction associée envoie plusieurs informations pour générer la page résultat :

- le dictionnaire des voitures pour les futures recherches
- la liste des étapes pour afficher la recherche précédente

L'utilisateur peut alors consulter le résultat de sa recherche. Si elle s'est bien passée, on voit les étapes sur la carte. Sinon un message s'affiche : "Nous n'avons pas trouvé d'itinéraire".

### 5.4.2 Côté simulation



FIGURE 5.3 – Affichage de la simulation

Lorsque l'on charge la route `/simulation`, on va récupérer le fichier `data/simulation.txt` qui contient la liste de toutes les bornes occupées et les bornes avec file d'attente par ticks. Cela forme un tableau qui est ensuite stocké dans une variable `JavaScript` nommé `table_simulation`. Ainsi, pour afficher les bornes occupées à  $i^{eme}$  tick, il suffit de parcourir la liste des coordonnées dans `table_simulation[i]`.

# Chapitre 6

## Tests et performances des fonctions

### 6.1 Complexité théorique des algorithmes

#### 6.1.1 Complexité de l'algorithme de recherche d'itinéraire

En posant les notations suivantes :

- $nb_{etapes}$  = nombre d'étapes pour aller du début à l'arrivée,
- $nb_{bornes}$  = nombre de bornes en France.

La fonction `getBorneFromDistance()` parcourt toutes les bornes et effectue un nombre fini d'opérations. Cette fonction est de complexité  $O(nb_{bornes})$ .

La fonction `plus_proche()` parcourt dans le pire des cas toutes les bornes, et pour chaque borne elle parcourt la liste des étapes déjà visitées. D'où une complexité en  $O(nb_{etapes} \times nb_{bornes})$ .

Enfin la fonction `get_liste_etape_type_distance()` et `get_liste_etape_type_temps()` appelle  $nb_{etapes}$  les fonctions `getBorneFromDistance` et `plus_proche()`. On arrive donc à une complexité en :

$$O(nb_{etapes} \times (nb_{etapes} \times nb_{bornes} + nb_{bornes}))$$

Soit approximativement :

$$O((nb_{etapes})^2 \times nb_{bornes})$$

#### 6.1.2 Complexité de l'algorithme de génération aléatoire d'itinéraire

On considère que la fonction `rand()` qui choisit un nombre aléatoire est une opération basique. On considère également que récupérer des données du fichier csv est une opération basique.

On note :  $nb_{trajets}$  = le nombre de trajets aléatoires que l'on veut générer.

La fonction `generate_x_random_itinerary` choisit dans un premier temps pour chaque trajet 2 nombres aléatoires qui constituent un trajet temporaire. On est donc en  $O(nb_{trajets})$ . Ensuite, pour chaque trajet, elle compare la distance du trajet temporaire avec la distance minimale choisie. Si le trajet temporaire n'est pas validé, on choisit un autre trajet aléatoirement et on teste à nouveau la distance. On répète cette opération jusqu'à trouver un trajet avec la bonne distance. Pour ne pas avoir de boucle infinie, nous avons ajouté une condition à cette comparaison pour limiter le nombre de tours de boucle à 50. La complexité de cette fonction dans le pire des cas est donc en  $O(50 \times nb_{trajets})$ . On obtient donc une complexité en :

$$O(nb_{trajets})$$

#### 6.1.3 Complexité de l'algorithme de simulation

Appellations :

- $n$  le nombre de trajets (ou le nombre de véhicules)
- $b$  le nombre total des bornes (17319)
- $t$  le nombre total de ticks (144 + 100 pour les décalages)

La récupération des données des bornes est à coût constant  $O(b)$ .

La génération de la liste des passages se fait trajet après trajet, et pour chaque trajet étape par étape. Un trajet dépasse rarement les 30 étapes, et considérer qu'un trajet passe par l'ensemble des bornes est trop extrême. Fixons le nombre d'étapes à 50 dans le pire cas. Il y a donc  $50 \times n$  tours de boucles appelant la fonction `ajout_passage`.

Complexité de la fonction `ajout_passage` :

De la même manière que précédemment, il est peu probable qu'une borne voit passer tous les véhicules, on fixe aussi le nombre maximum de véhicules passant à une borne à 50.

Avant ajout, une liste de passage comporte au pire  $49 \times 3$  passages (arrivée, attente, sortie). La pire configuration serait que chaque élément de la liste de passages soit amené à changer, la liste d'attente comporterait au pire  $50 \times 3$  passages. On parcourt alors l'ensemble des éléments de la liste des passages. Le parcours peut entraîner des modifications dans la liste d'attente, au pire 50 fois (une fois par passage en attente).

La complexité est alors de  $O(49 \times 3 \times (50 \times (3 + 1)))$  ou  $O(12n^2)$ .

Complexité de la fonction `export` :

La fonction parcourt une unique fois l'ensemble des liste des passages de l'ensemble des bornes. Mais pour chaque borne, le parcours se fait sur l'ensemble des ticks. A cela s'ajoute un ultime tour de boucle sur les bornes pour écrire dans le fichier texte d'export. La complexité est alors en  $O(2btn)$ .

Au final, la complexité de la fonction de simulation est :  $O(b) + O(600n^3) + O(2btn)$ .

## 6.2 Tests unitaires et performance

### 6.2.1 Cadre

Pour réaliser les tests unitaires et les tests de performances, nous allons utiliser la librairie `snow.h` disponible sur le [GitHub de mortie](#). L'objectif des tests est de vérifier la vitesse d'exécution des différents algorithmes et de vérifier la précision de certaines fonctions.

### 6.2.2 Performance de l'algorithme de recherche d'itinéraire

Impact de la distance sur les performances

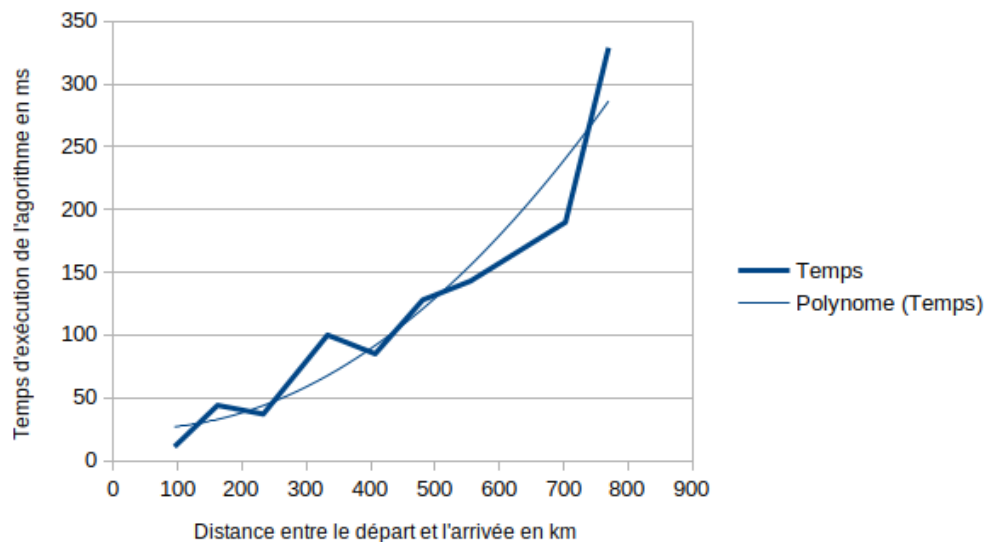


FIGURE 6.1 – Performances de l'algorithme de recherche d'itinéraire

On voit que les performances d'exécution sont directement liées à la distance entre le départ et l'arrivée. L'algorithme reste efficace sur tout le territoire avec un temps d'exécution qui est inférieur à 1 seconde. On retrouve la complexité en  $O((nb_{etapes})^2 \times nb_{bornes})$  en approchant les résultats par une courbe de tendance de type polynomiale de degré 2.

### 6.2.3 Génération aléatoire d'itinéraire

Dans cette section, on teste les performances de la fonction `generate_x_random_itinerary()` implémentée dans `random_points.c`

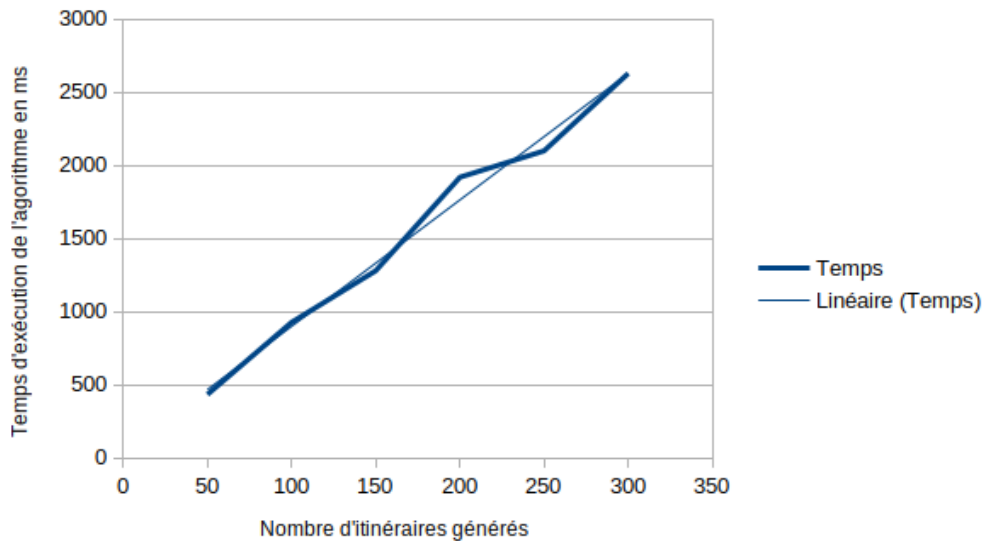


FIGURE 6.2 – Performances de l'algorithme de génération aléatoire d'itinéraire

On voit que le temps de génération est linéaire.

### 6.2.4 Multithreading et forks

Le multithreading a finalement été peu concluant. Sur toutes les batteries de tests que nous avons pu lancer, faire des threads s'est révélé être aussi voire moins rapide d'exécution que de ne pas en faire.

L'utilisation de forks a permis une meilleure fluidité d'exécution. Les premiers tests se sont avérés concluants, avec une exécution durant une petite trentaine de secondes pour 100 trajets générés aléatoirement. Étant donné que les trajets individuels prennent en général jusque 500ms à se générer, cela marquait déjà une amélioration. En travaillant sur le nombre de forks lancés simultanément, on se rend compte que plus on en lance, plus la moyenne de vitesse de traitement d'un trajet devient petite. On se retrouve néanmoins rapidement bloqué par les ressources de l'ordinateur. Un fork étant un processus à part entière, les ressources utilisées sont vite très hautes, avec notamment le processeur qui travaille à 100% sur ses 8 coeurs.

On observe alors que l'on a avec une rapidité de génération plutôt optimale à environ 50 forks simultanés. C'est cette valeur que nous avons choisi de garder pour la suite. Au final, sur un total de 1000 générations d'itinéraires, nous avons réussi à atteindre au mieux une moyenne de 150ms par trajet généré.

### 6.2.5 Performances de l'algorithme de génération multiple de trajets

La génération multiple de trajet se basant sur l'utilisation de forks, il est difficile d'évaluer sa complexité. La complexité d'un fork, indépendamment des autres, est cependant connue : c'est celle de l'algorithme de génération d'itinéraire. Afin de faire quand même une étude de cette génération d'itinéraires, nous avons lancé de nombreux tests afin de mesurer l'impact du nombre de forks sur la rapidité d'exécution, ainsi que l'influence du nombre de trajets générés. Pour 10, 20 puis 50 forks, nous avons exécuté des tests pour un nombre de trajets compris entre 100 et 1000. Pour chaque nombre de trajet, 10 simulations ont été lancées pour ensuite calculer une moyenne et lisser les imprécisions.

Les graphes suivants montrent le temps d'exécution en fonction du nombre de trajets voulus :

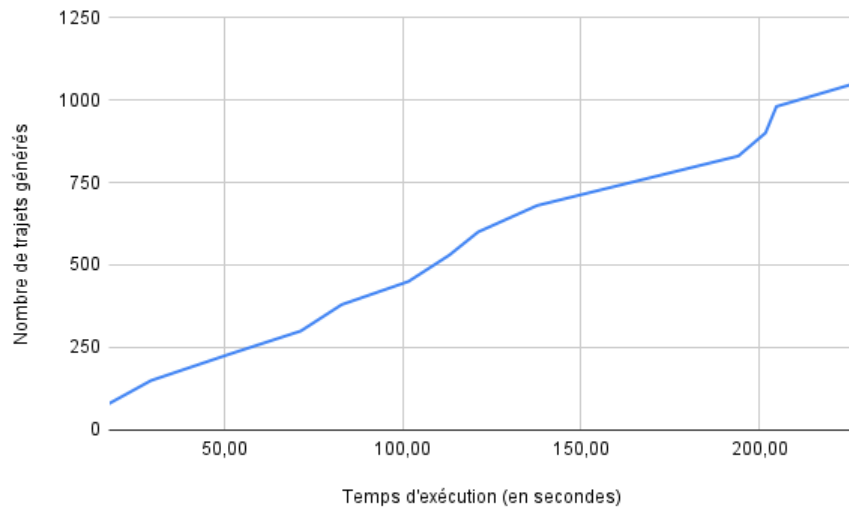


FIGURE 6.3 – Évolution du temps d'exécution en fonction du nombre de trajets générés (pour 10 forks)

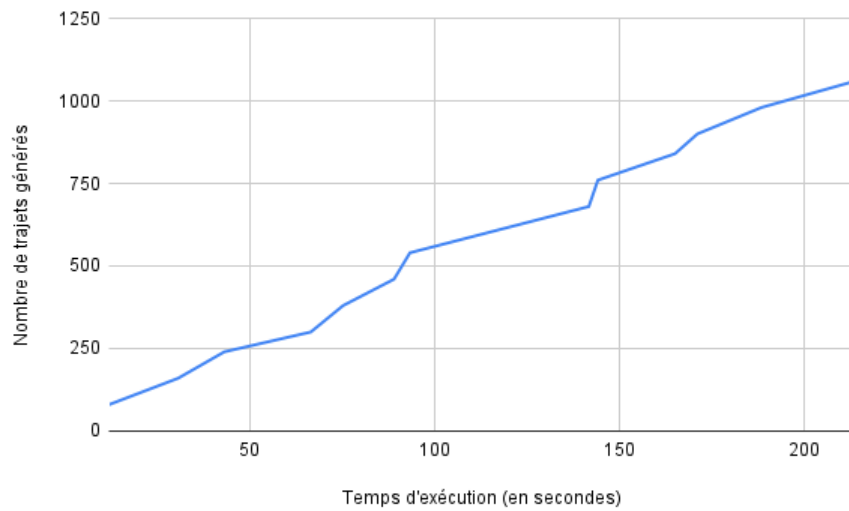


FIGURE 6.4 – Évolution du temps d'exécution en fonction du nombre de trajets générés (pour 20 forks)

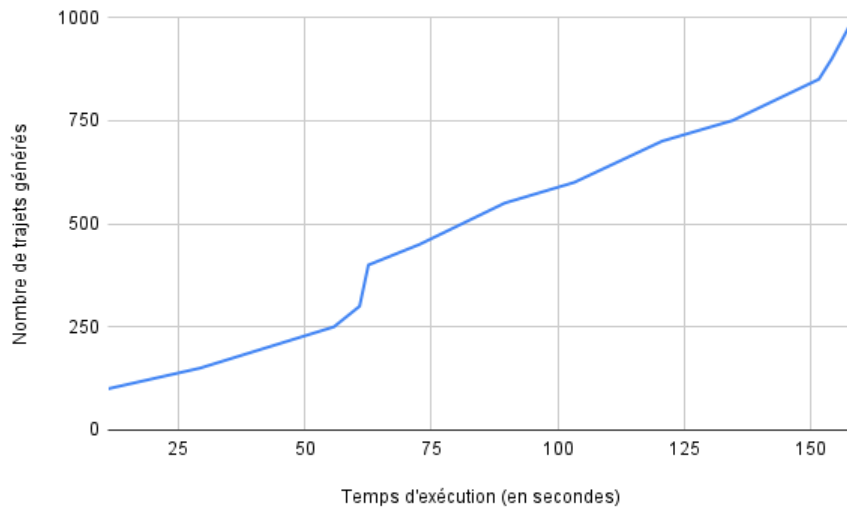


FIGURE 6.5 – Évolution du temps d'exécution en fonction du nombre de trajets générés (pour 50 forks)

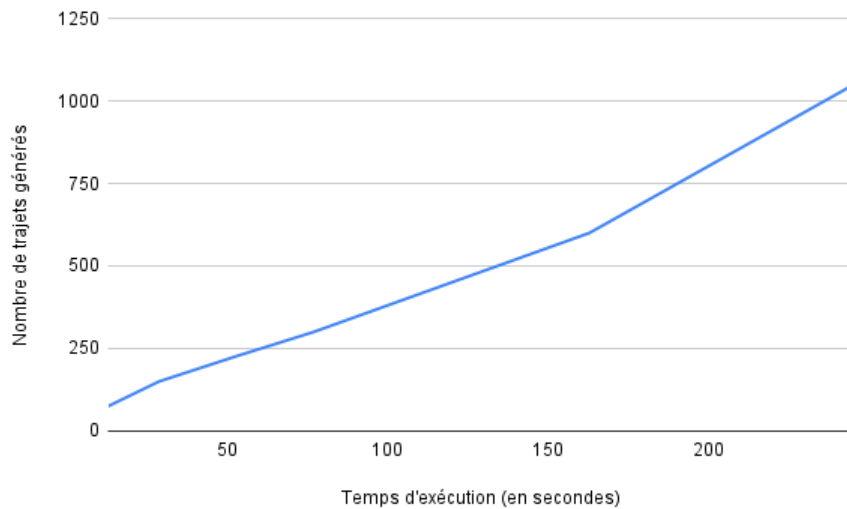


FIGURE 6.6 – Évolution du temps d'exécution en fonction du nombre de trajets générés (pour 75 forks)

Les données pour 75 forks sont moins précises car, par manque de temps, nous avons lancé bien moins de tests. On remarque cependant que 50 forks semble être une valeur optimale. En effet, la durée pour générer 1000 trajets avec 50 forks est d'environ 158 secondes (soit 2min 38s) contre plus de 200 pour 20 forks. Dans une configuration optimale, on peut donc générer en moyenne 1 trajet toutes les 158ms.

### 6.2.6 Tests unitaires

Grâce au module `snow.h` nous avons pu tester le résultat de certains modules. Nous avons testé les modules qui implémentent des structures de données utilisées pour les deux algorithmes (calcul d'itinéraire et simulation). Nous avons testé :

- `bornes.h`
- `random_points.h`
- `vehicules.h`

L'objectif de ces tests est de vérifier la cohérence et la conformité des retours de fonctions. Comme ces modules sont fondamentaux pour notre code, cela permet de nous assurer des bases saines et de limiter au maximum les possibilités d'erreurs.



Les tests sont disponibles sur le projet dans le dossier `app/tests` et appelables avec une commande `make`. Voici les résultats de ces tests :

```
Testing test_parametre_aleatoire:
✓ Success: Conformance test of id vehicule (15.14µs)
✓ Success: Conformance test of 'Seuil de décharge' (4.88µs)
✓ Success: Conformance test of recharge time (5.86µs)
✓ Success: Conformance test of initial autonomy (5.13µs)
test_parametre_aleatoire: Passed 4/4 tests. (218.02µs)

Testing test_recuperation_coord_ligne:
✓ Success: Conformance test (2.94ms)
test_recuperation_coord_ligne: Passed 1/1 tests. (2.97ms)

Testing test_generate_x_random_itinerary:
✓ Success: Conformance test : size (279.98ms)
✓ Success: Conformance test : distance trajet (170.85ms)
test_generate_x_random_itinerary: Passed 2/2 tests. (450.96ms)

Total: Passed 7/7 tests. (454.19ms)
```

FIGURE 6.7 – Tests unitaires du module random points

```
Testing test_create_voiture:
✓ Success: Conformance test (2.95ms)
test_create_voiture: Passed 1/1 tests. (3.45ms)

Testing test_update_charge:
✓ Success: Conformance test (1.14ms)
test_update_charge: Passed 1/1 tests. (1.31ms)

Testing test_update_puissance:
✓ Success: Conformance test : without reserve (1.12ms)
✓ Success: Conformance test : with reserve (1.08ms)
test_update_puissance: Passed 2/2 tests. (2.49ms)

Testing test_update_autonomie:
✓ Success: Conformance test (1.11ms)
test_update_autonomie: Passed 1/1 tests. (1.26ms)

Testing test_recharge:
✓ Success: Conformance test : no recharge time minimum (1.24ms)
✓ Success: Conformance test : recharge more than needed (1.15ms)
✓ Success: Conformance test : doesn't recharge to maximum (1.15ms)
test_recharge: Passed 3/3 tests. (3.99ms)

Testing test_set_autonomie:
✓ Success: Conformance test (1.17ms)
test_set_autonomie: Passed 1/1 tests. (1.34ms)

Testing test_simulation_recharge:
✓ Success: Conformance test : no recharge time minimum (1.15ms)
✓ Success: Conformance test : recharge more than needed (2.44ms)
✓ Success: Conformance test : doesn't recharge to maximum (2.12ms)
test_simulation_recharge: Passed 3/3 tests. (6.15ms)

Total: Passed 12/12 tests. (20.10ms)
```

FIGURE 6.8 – Tests unitaires du module vehicules

```
Testing test_list_bornes_visitees_create:
✓ Success: Conformance test (30.03µs)
test_list_bornes_visitees_create: Passed 1/1 tests. (150.15µs)

Testing test_list_bornes_visitees_append:
✓ Success: Conformance test (8.79µs)
test_list_bornes_visitees_append: Passed 1/1 tests. (39.06µs)

Testing test_borne_deja_visitee:
✓ Success: Conformance test (9.03µs)
test_borne_deja_visitee: Passed 1/1 tests. (37.11µs)

Total: Passed 3/3 tests. (239.01µs)
```

FIGURE 6.9 – Tests unitaires du module bornes

# Chapitre 7

## Gestion de projet

### 7.1 Charte de projet

#### 7.1.1 1. Cadrage

**Contexte :**

Le projet se déroule dans le cadre du 2ème semestre de première année d'étude à TELECOM NANCY. Le thème est : *Optimisation et simulation de trajets en voiture électrique*. Le projet doit :

- répondre au thème
- intégrer des éléments de Gestion de Projet
- mettre en oeuvre les connaissances du cours de C et de Structure de Données.

**Finalités et importances du projet :**

- Importance : Ce projet est *prioritaire* car il est nécessaire à la validation de la première année
- Contraintes et dépendances : Projet à réaliser en même temps que les cours
- Tous les membres maîtrisent les éléments du cours de C&SD et de Gestion de Projet

**Cahier des charges succinct :**

Le livrable :

- Créer un ensemble de fonctions qui détermine un trajet en voiture électrique répondant à des contraintes
- Créer un mode *simulation* pour évaluer l'état du réseau électrique en fonction d'un ensemble d'utilisateurs actifs.

Langage imposé :

- C

Critère et indicateur de succès :

	Gestion de projet	Le livrable	Compétences acquises
Faible	Réunions sans préparation et groupe sujet à l'effet tunnel	Fonctions qui permettent de retourner l'itinéraire et les points de recharge	Comprendre uniquement son code
Moyen	Réunions organisées	Le rendu comporte les fonctions d'itinéraire et le mode simulation	Comprendre son code et celui des autres
Fort	Gestion de projet suivie, utilisations récurrentes des outils de gestion de projet	Interface graphique qui permet de visualiser le calcul d'itinéraire et le mode simulation	Maîtrise complète du langage C, être capable d'optimiser son code et celui des autres

FIGURE 7.1 – Critères et indicateurs de succès

## 7.1.2 2. Déroulement

### Organisation :

- Mathis MANGOLD
- Aurélien GINDRE
- Hélène BARBILLON
- Adrien LAROUSSE

### Ressources mise en place :

- Communication : Serveur Discord
- Gestion des tâches : Comptes-Rendus
- Partage de documents : Google Drive, GitLab, Overleaf

### Evènements importants :

- Formation du groupe de projet : **24 mars 2023**
- Dépôt du projet : **24 mai 2023 à 20h00 CEST**
  - Gestion de projet : Compte-Rendus de réunion, éléments de planification, suivi analytique et analyse post-mortem
  - Rapport : décrivant les choix motivés des structures de données et des algorithmes (sur la base d'un état de l'art détaillé), les fonctions réalisées et leur analyse (complexité mémoire, temps), les fonctions réalisées et les tests mis en œuvre
  - Guide utilisateur : document décrivant l'installation du programme et les modalités de son utilisation
  - Code : le code doit être compilable par un appel à la commande `make`
- Soutenance : **31 mai 2023 à 8h30**

## 7.2 Communication

Pour communiquer efficacement, nous avons utilisé un serveur discord dédié au projet. Nous avons fait au moins une réunion par semaine, et avons adapté le format (réunion de chantier ou stand-up meeting) en fonction des nécessités du projet et de nos emplois du temps (*comptes-rendus des réunions*). Nous avons utilisé un dossier partagé google drive pour certains documents, l'éditeur de documents partagé LaTeX pour le compte-rendu, et bien entendu le dépôt Git du projet pour le code.

## 7.3 Analyse SWOT du projet

	POSITIF	NÉGATIF
INTERNE	<b>Forces</b> <ul style="list-style-type: none"> <li>• Expérience du premier projet P2I</li> <li>• Membres qui ont déjà travaillé ensemble</li> <li>• Module de C suivi par tous les membres</li> <li>• Algorithmes de recherche d'itinéraires qui existent déjà</li> </ul>	<b>Faiblesses</b> <ul style="list-style-type: none"> <li>• Délais courts: besoin d'être efficaces</li> <li>• Langage C complexe pas tout à fait maîtrisé</li> </ul>
EXTERNE	<b>Opportunités</b> <ul style="list-style-type: none"> <li>• École qui ferme tard pour y travailler le soir</li> <li>• Sujet d'actualité: problématique de transition énergétique</li> </ul>	<b>Menaces</b> <ul style="list-style-type: none"> <li>• Partiels, associatif (attention à l'organisation et aux événements imprévus)</li> </ul>

FIGURE 7.2 – Matrice SWOT

**Forces :** Le groupe est composé de 4 membres provenant de 2 équipes du 1er projet : Adrien et Aurélien ont donc déjà travaillé ensemble sur le 1er PPII, de même pour Mathis et Hélène. Il y a donc une certaine synergie déjà présente. Il y a une bonne cohésion et une bonne entente entre les 4 membres, ce qui nous permettra d'avoir une communication fluide et efficace et de limiter les risques d'effet tunnel.

Les membres du groupe se voient également quasiment tous les jours de la semaine, ce qui permettra une communication synchrone et sans intermédiaire.

L'expérience acquise lors du 1er projet PPII (réalisé au 1er semestre) va nous être précieuse : nous allons reprendre les éléments qui nous ont permis d'être efficaces et surtout éviter de répéter les erreurs commises. Nous avons mis en commun nos retours et comparé nos expériences respectives lors de la 1ère réunion (*comptes-rendus des réunions*).

Nous allons adopter une gestion de projet plus souple : nous n'allons pas faire de diagramme de Gantt que nous jugeons inapproprié pour ce genre de projet (car c'est difficile voire impossible de tout prévoir à l'avance), et nous allons mettre à jour régulièrement les lots de travail pour qu'ils collent à la réalité et aux avancements du projet. Nous avons repris des éléments d'organisation du 1er PPII comme l'architecture du discord, le drive et le déroulement des réunions.

De plus, il existe déjà des solutions et des algorithmes pour trouver des itinéraires en voiture électrique, nous n'aurons pas à tout inventer.

Enfin, tous les membres du groupe ont suivi le module de C enseigné à Telecom Nancy, ce qui constitue une bonne base pour démarrer.

**Faiblesses :** Nous avons environ 2 mois pour réaliser ce projet, les délais sont courts, il va donc falloir être efficace. C'est pourquoi nous misons sur une gestion de projet dynamique et sur l'investissement de chacun.

De plus, ce projet est en C, un langage de programmation complexe qui nécessite beaucoup d'expérience pour être parfaitement maîtrisé. Nous avons commencé à apprendre ce langage en début de semestre, nos connaissances sont donc encore assez faibles, mais ce projet va être l'occasion d'appliquer toute la théorie vue en cours et de nous améliorer.

**Opportunités :** Nous avons la chance d'être dans une école qui ferme tard le soir, nous pourrions donc y rester pour travailler et faire des sessions de groupe après les cours. Le thème de ce projet est un sujet d'actualité, les problématiques de transition énergétique sont "à la mode" et nous concernent directement.

**Menaces :** Ce projet est réalisé en parallèle des cours. Il faudra donc faire très attention à ne pas se laisser submerger par une charge de travail trop importante et anticiper les partiels qui arrivent en Mai. Les membres du groupe sont également tous très impliqués dans la vie associative de l'école. Il faudra donc savoir anticiper toute accumulation d'événements imprévus, et prendre en compte les impératifs de chacun.

## 7.4 WBS

Nous avons volontairement choisi de ne pas faire de diagramme de Gantt pour ce projet, puisqu'il nous a tous desservi lors du premier projet, mais nous avons décomposé les tâches du projet et les avons classées par ordre de priorité :

<b>Traitement des données</b>
<ul style="list-style-type: none"> <li>- Récupérer les données (par pertinence)</li> <li>- Stocker les données</li> <li>- Accéder aux données</li> <li>- Lire les entrées et sorties utilisateurs (<b>casse et sécurité</b>) <ul style="list-style-type: none"> <li>- Contrôler</li> <li>- Stocker</li> <li>- Traiter</li> </ul> </li> </ul>
<b>Makefile</b>
<ul style="list-style-type: none"> <li>- lancer le projet depuis une commande <b>make</b></li> </ul>
<b>Calcul de l'itinéraire</b>
<ul style="list-style-type: none"> <li>- Algorithme du plus court chemin</li> <li>- Intégration de contraintes <ul style="list-style-type: none"> <li>- Autonomie minimum</li> <li>- Temps de recharge</li> </ul> </li> <li>- Disponibilité des bornes sur le réseau → amélioration/adaptation d'itinéraire (partie 2)</li> </ul>
<b>Mode de simulation</b>
<ul style="list-style-type: none"> <li>- Stocker les itinéraires détaillés du jeu de données</li> <li>- Calculer taux de charge des bornes sur territoire (par tranches de 10 min)</li> <li>- Identifier les files d'attentes (par tranches de 10 min)</li> </ul>
<b>Tests</b>
<ul style="list-style-type: none"> <li>- Créer un jeu de données <ul style="list-style-type: none"> <li>- Mode simulation</li> <li>- Mode itinéraire (pas plus que 20 min de charge, 30% mini, etc.)</li> </ul> </li> <li>- Exactitude des résultats</li> <li>- Effectuer des tests de performances</li> </ul>
<b>Interface graphique</b>
<ul style="list-style-type: none"> <li>- Récupérer les entrées utilisateur</li> <li>- Afficher une carte, un trajet</li> <li>- Recherches pour un affichage en C ou en python (web)</li> </ul>
<b>Rendu</b>
<ul style="list-style-type: none"> <li>- Ecrire un guide utilisateur <b>README.md</b></li> <li>- Détail de la gestion de projet (Suivi et Post Mortem)</li> <li>- Rédaction du rapport</li> </ul>

FIGURE 7.3 – Décomposition des tâches du projet

Obligatoire	Fortement recommandé	Recommandé	Optionnel
Calcul de l'itinéraire	Mode simulation	Tests	Interface graphique
Traitement des données		Adaptation d'itinéraire (Calcul d'itinéraire)	
Makefile			
Rendu			

FIGURE 7.4 – Ordre de priorité des tâches du projet

# Chapitre 8

## Conclusion

### 8.1 Retour sur les indicateurs de succès

Cette partie revient sur le tableau des indicateurs de succès défini dans la *charte de projet*.

#### Gestion de projet

La gestion de projet appliquée tout au long du projet correspond à l'indicateur **moyen** dans le tableau. En effet nos réunions sont restées organisées jusqu'au bout et des outils ont été mis en place dès le début. En revanche ces outils sont restés relativement basiques et la répartition des tâches a été compliquée.

#### Le livrable

Le livrable fournit répond à l'objectif **fort** que nous nous étions fixé, nous avons des algorithmes en C qui fonctionnent ainsi qu'une interface graphique qui permet de visualiser les résultats.

#### Compétences acquises

Les compétences acquises pendant le projet correspondent à l'objectif **moyen**, en effet nous comprenons tous nos codes mais des améliorations et des optimisations étaient encore possibles.

### 8.2 Pistes d'améliorations

#### 8.2.1 Optimisation de la recherche de trajet

Pour l'algorithme vu dans le *Chapitre 3 : Calcul d'itinéraire*, l'algorithme fonctionne correctement et trouve une solution. Cependant, certaines parties de l'implémentation auraient pu être simplifier. Notamment, une fusion entre la fonction `getBorneFromDistance()` et `plus_proche()` afin de limiter la complexité en mémoire.

#### 8.2.2 Génération de multiples trajets

Un problème persiste sur la génération des trajets avec les fork. En effet, le dernier fork ouvert met parfois de plusieurs secondes à plusieurs dizaines de secondes à s'exécuter. Ce problème concerne uniquement le dernier fork et ne survient pas de manière régulière. Le trajet calculé par ce dernier fork ne met pourtant en rien notre algorithme en difficulté. Malgré une dizaine d'heure de tentatives de débogage, la source de ce problème reste inconnue. En réglant ce ralentissement, cela nous permettrait de diminuer grandement le temps de traitement des trajets pour la simulation.

### 8.3 Bilan global

Au terme de deux mois de recherche et de mise en oeuvre, nous avons pu produire une application fonctionnelle et aboutie. C'est le résultat du mélange des connaissances des différents membres du groupe et les ressources mises à notre disposition durant l'année scolaire.

Bien que la répartition des tâches au sein du groupe fût difficile (difficulté de travailler à en même temps sur l'implémentation des algorithmes et non utilisation des branches GIT), nous avons pu finir dans les délais. Chacun a travaillé sur une partie et tout le monde a participé à la construction du projet.

## Chapitre 9

# Bilans personnels

### 9.1 Aurélien GINDRE

<b>Points positifs</b>	- Sujet intéressant avec des bases reposant sur du cours (graphes, listes)
<b>Difficultés rencontrées</b>	- Gestion des pointeurs pointant sur le mauvais objet - Oublis de libération d'allocation de mémoire - Impossibilité de travailler sur un même algorithme en même temps
<b>Expérience personnelle</b>	- Meilleure maîtrise des pointeurs et de l'allocation mémoire - Un regard différent lors de la conception, le papier est plus clair que le code
<b>Axes d'amélioration</b>	- Une meilleure répartition des tâches pour ne plus avoir le problème de se marcher dessus - Prévoir au préalable les différentes attentes globales avant de se lancer dans le code

### 9.2 Adrien LAROUSSE

<b>Points positifs</b>	- Sujet intéressant qui demande un peu de travail d'adaptation et de recherche (notamment au niveau algorithmique) - Approfondissement de certaines compétences : C, Javascript, écriture de Makefile, Module <code>snow.h</code>
<b>Difficultés rencontrées</b>	- Communication entre les différents langages - Affichage de la carte, première utilisation de ce système - Difficultés à travailler simultanément sur le projet sans prendre le risque d'avoir des divergences de branches sur le dépôt
<b>Expérience personnelle</b>	- Prise de conscience de l'importance de mettre ses idées sur papiers avant de se mettre à coder - Mélange des compétences du 1er et 2ème semestre
<b>Axes d'amélioration</b>	- Travailler avec les branches sur Git - Prévoir au préalable les différentes attentes globales avant de se lancer dans le code - Plus inclure les autres membres du groupe dans mes portions de travail (session de travail en groupe par exemple)



### 9.3 Hélène BARBILLON

<b>Points positifs</b>	<ul style="list-style-type: none"><li>- Manipulations du langage C sur des problèmes concrets</li><li>- Sujet intéressant et d'actualité</li></ul>
<b>Difficultés rencontrées</b>	<ul style="list-style-type: none"><li>- Problèmes d'allocation de mémoire</li><li>- Problèmes de merge sur Git</li><li>- Difficultés de compréhension quand il fallait travailler sur le code d'un autre membre du groupe</li></ul>
<b>Expérience personnelle</b>	<ul style="list-style-type: none"><li>- Meilleure maîtrise du langage C et de Git</li><li>- Expérience de travail de groupe</li><li>- Se rendre compte de l'importance de la phase de conception, et du décalage entre ce qu'on imagine et ce qui est faisable</li></ul>
<b>Axes d'amélioration</b>	<ul style="list-style-type: none"><li>- Meilleure répartition des tâches</li><li>- Travailler avec des branches sur Git</li></ul>

### 9.4 Mathis MANGOLD

<b>Points positifs</b>	<ul style="list-style-type: none"><li>- Gain d'une grande fluidité en programmation C</li><li>- Sujet très pertinent et motivant</li></ul>
<b>Difficultés rencontrées</b>	<ul style="list-style-type: none"><li>- Le multithreading et les forks</li><li>- Débogage d'un programme incluant des forks</li><li>- Travail se reposant sur des fonctions codées par une autre personne, compréhension de la manière de coder</li></ul>
<b>Expérience personnelle</b>	<ul style="list-style-type: none"><li>- Expérience supplémentaire de travail en groupe</li><li>- Meilleure compréhension du C par la pratique</li><li>- Apprentissage du multithreading et des forks en C</li><li>- Apprentissages sur l'approche et la réflexion autour d'un problème</li></ul>
<b>Axes d'amélioration</b>	<ul style="list-style-type: none"><li>- Encore davantage se reposer sur la gestion de projet</li></ul>

## Chapitre 10

## Annexes

### Comptes-rendus de réunion

# PROJET PPII - Compte-rendu N°1

<u>Motif / type de chantier :</u> Réunion de lancement	<u>Lieu :</u> Télécom Nancy
<u>Présents :</u> Mathis MANGOLD Hélène BARBILLON Adrien LAROUSSE Aurélien GINDRE	<u>Date :</u> 21/03/2023 <u>Durée :</u> 17h00 à 18h00

## Ordre du jour :

- Discussion générale de lancement de projet

## Résumé de la réunion :

- Nous avons discuté des différents outils que nous allons utiliser : overleaf, drive, discord.
- Nous avons discuté de l'Etat de l'art, c'est-à-dire chercher ce qui existe et ce que nous allons faire grâce à ces enseignements
- Nous avons abordé les différents documents de gestion de projet que nous allons utiliser : charte projet, swot, wbs
- Nous avons parlé de l'organisation des réunions : une hebdomadaire le plus souvent possible ; une personne qui préside, une fait le secrétaire, avec des roulement chaque semaine.
- Nous avons abordé la méthode agile sur laquelle nous pouvions nous appuyer, basé sur des cycles de travail, une modification assez régulière des lots...
- Par rapport aux avancements dans le projet, il faudrait expliquer chaque commit ; il faudrait aussi commenter notre code pour le rendre compréhensible pour les autres. Pour éviter un effet tunnels et de gros conflits, il nous paraît essentiel de faire des commits réguliers.
- Nous avons abordé la contrainte majeure qu'étaient les partiels de mai
- Notre charte projet comprendra :
  - Le cadrage (contexte, finalité et importance du projet, tableau indicateurs de succès du projet)
  - Le déroulement (ressources, dates)
- Nous ferons une matrice SWOT
- Il a été décidé de ne pas faire de Gantt car peu utilisé lors du précédent projet
- La prochaine réunion portera sur l'établissement des lots du wbs ensemble
  - **Première étape du projet**
    - Nous avons discuté des chemins reliant chaque station, des possibles idées de calcul
    - Pour bien réussir cette première étape, il faudra séparer en sous-tâches bien pensées.
    - Une des prochaines réunion portera sur la définition des fonctions : définir les fonctions (headers), faire un tableau des dépendances, et autres outils utiles à la conception
    - Lors de la découpe de la charge de travail, il faut faire attention à ce que tout le monde puisse avancer sur des fonctions en même temps (pas de personne bloquée qui attend que les autres aient fini pour pouvoir faire sa fonction)
    - Il ne faudra pas oublier d'effectuer des tests, avec Snow par exemple
  - **Seconde étape**
    - Pour bien réussir cette seconde étape, il faudra impérativement avoir une première étape performante et bien finalisée.
    - Nous avons discuté du fait qu'une interface graphique permettrait de voir les charges de chaque station, les voitures et avoir une meilleure lisibilité et compréhension.

## TO DO list :

Description	Personne(s) concernée(s)	Date
Etat de l'art	Mathis MANGOLD, Aurélien GINDRE	Mardi 28 Mars
Cahier des charges / Charte de projet	Adrien Larousse	Mardi 28 Mars
SWOT et analyse	Hélène Barbillon	Mardi 28 Mars

# PROJET PPII - Compte-rendu N°2

<u>Motif / type de chantier :</u> Fin état de l'art & lancement de la construction du projet	<u>Lieu :</u> Télécom Nancy
<u>Présents :</u> Mathis MANGOLD Hélène BARBILLON Adrien LAROUSSE Aurélien GINDRE	<u>Date :</u> 28/03/2023 <u>Durée :</u> 17h20 à 18h00

## Ordre du jour :

- Reprise de l'état de l'art
- Cahier des charges
- Matrice S.W.O.T.
- Dépendance du WBS
- [Traitement des données] Type de stockage/traitement des données
- [Gestion de projet] Deadline de la 1ère étape

## Résumé de la réunion :

- Charte de projet : Adrien présente son travail sur la charte de projet. Elle a été reprise point par point. Certains ont été modifiés pour plus de compréhension
- Matrice SWOT : La réunion s'est poursuivie sur l'explication de la matrice SWOT par Hélène, elle avait déjà proposé une première version au groupe entre les deux réunions. La deuxième relecture a été validée.
- Etat de l'art : Aurélien commence à présenter son travail sur l'état de l'art :
  - Aurélien a regardé les algorithmes déjà existants. Il a trouvé un document reprenant de nombreux algorithmes de plus court chemin d'un graphe. Il nous présente l'algorithme de A\* à l'aide d'un exemple.
  - Mathis a de son côté regardé les applications qui existent déjà. Il nous présente le résultat de ses recherches.
- Nous avons écrit dans une réunion dédiée le WBS, nous sommes revenus dessus pour définir les liens entre les lots de travail.
- [Traitement de données] On a discuté de comment récupérer, stocker et traiter les données, on a parlé d'implémentation et d'utilisation de la base de données.
- Deadline de la première partie du projet : 17 avril 2023

## TO DO list :

Description	Personne(s) concernée(s)	Date
Traitement de données et écriture de la matrice d'adjacence	Mathis MANGOLD et Adrien LAROUSSE	Mercredi 5 Avril
Écriture et implémentation de l'algorithme A* en C	Hélène BARBILLON et Aurélien GINDRE	Mercredi 5 Avril

# PROJET PPII - Compte-rendu N°3

<u>Motif / type de chantier :</u> Réunion de chantier	<u>Lieu :</u> Télécom Nancy
<u>Présents :</u> Mathis MANGOLD Hélène BARBILLON Adrien LAROUSSE Aurélien GINDRE	<u>Date :</u> 14/04/2023 <u>Durée :</u> 18h-18h45

## Ordre du jour :

- Avancées du projet
- Contraintes à ajouter à l'algorithme
- Problèmes sur l'algorithme à régler

## Résumé de la réunion :

- Avancées du projet :  
Nous avons un algorithme qui permet de trouver un itinéraire d'un point A à un point B en passant par les bornes de la base de donnée qui nous a été fournie. Le fonctionnement de l'algorithme est semblable à celui du A\*.  
Nous utilisons le module python geopy pour récupérer une latitude et longitude à partir d'une adresse. Par soucis de facilité d'utilisation de l'algorithme par l'utilisateur, nous réaliserons une page web sur laquelle il pourra saisir un point de départ et un point d'arrivée, et obtiendra un trajet. Cela nous permettra aussi de mieux contrôler les entrées utilisateur et d'éviter des problèmes avec l'algorithme en C. Au moment de la réunion, la commande pour lancer le programme est : `make main && python3 interface/app.py`
- Contraintes à ajouter à l'algorithme :  
Nous avons discuté des contraintes à ajouter au programme comme le temps de recharge borné, des options concernant l'autonomie de la voiture, et avons commencé à réfléchir à des solutions.
- Problèmes à régler :  
Adrien a évoqué plusieurs cas d'erreurs de l'algorithme : quand deux points se situent à des latitudes/-longitudes proches, l'algorithme cherche des bornes qui se situent sur cette ligne et n'en trouve pas. Une macro va être ajoutée pour palier à ce problème. Un autre cas d'erreur est que l'algorithme ne garde pas en mémoire les bornes déjà visitées, et qu'il y a un risque qu'il fasse des allers-retours si le véhicule a peu d'autonomie et qu'il ne trouve pas de borne. Un tableau des bornes visitées va être ajouté, ainsi que des conditions supplémentaires.

## TO DO list :

Description	Personne(s) concernée(s)	Date
Interface graphique	Adrien	Lundi 24/04
Bornage du temps de recharge	Mathis	Lundi 24/04
Gestion des bornes visitées	Hélène	Lundi 24/04
Début de la 2e partie du projet	Aurélien	Lundi 24/04

Prochaine réunion en ligne (vacances) : lundi 24/04 (heure à choisir)

# PROJET PPII - Compte-rendu N°4

<u>Motif / type de chantier :</u> Réunion de chantier	<u>Lieu :</u> En ligne
<u>Présents :</u> Mathis MANGOLD Hélène BARBILLON Adrien LAROUSSE Aurélien GINDRE	<u>Date :</u> 24/04/2023 <u>Durée :</u> 15h-16h35

## Ordre du jour :

- Todo list et avancées diverses
- Répartition partie 2 du projet

## Résumé de la réunion :

- Interface graphique : L'interface graphique a été améliorée et nous disposons maintenant d'un site internet plutôt abouti qui gère les entrées utilisateur sous différentes formes : possibilité de rentrer des villes de départ et d'arrivée, réglage des options d'itinéraires à la main.
- Bornage du temps de recharge : Les paramètres supplémentaires demandés dans le sujet ont été ajoutés à l'algorithme. Il est désormais possible pour l'utilisateur de spécifier une autonomie initiale de la voiture et un pourcentage de batterie à garder en réserve. De plus, un paramètre de bornage des temps de recharge a été ajouté. Cet ajout s'accompagne d'une nouvelle option algorithmique : un calcul d'itinéraire optimisé en temps est maintenant disponible, en complément de celui optimisé en distance.
- Gestion des bornes visitées : Une liste simplement chaînée contenant toutes les bornes déjà visitées a été ajoutée. Ainsi, il n'est plus possible pour l'algorithme de rentrer dans une boucle infinie d'allers-retours entre 2 bornes.
- Début de la 2e partie du projet : L'outil de simulation tournera autour d'une structure de données avec différents sets de données. Premier set de données pour chaque véhicule : à quel tick il arrive et part de quelle borne. Deuxième set pour chaque borne : les informations de tous les véhicules qui y arrivent et en repartent avec les ticks correspondants. Troisième set pour pour chaque tick : quelle borne subit un changement entre le passage du tick précédent au tick actuel.
- Améliorations à implémenter et problèmes à régler :  
Il faut pouvoir rentrer directement des coordonnées de départ et d'arrivée sur le site. Il serait intéressant de faciliter la lecture des différentes étapes par l'utilisateur. Il faudrait également trier les voitures par ordre alphabétique et afficher les paramètres d'entrée quand l'itinéraire est retourné. Mais l'interface reste optionnelle et il ne faut donc pas trop s'attarder sur des détails.  
L'algorithme optimisé en temps nécessite que l'utilisateur donne un temps maximal passé à la borne. Il crash donc dès que ce paramètre n'est pas spécifié.  
Pour l'outil de simulation, il faut intégrer le nombre de points de recharge disponibles par borne.

## TO DO list :

Description	Personne(s) concernée(s)	Date
Travail avec Aurélien sur la simulation	Adrien	Vendredi 12/05
Implémenter fork et gérer l'accessibilité pour la simulation	Mathis	Vendredi 12/05
Travail sur le site pour la visualisation de la simulation	Hélène	Vendredi 12/05
Travail avec Adrien sur la simulation	Aurélien	Vendredi 12/05

Prochaine réunion : vendredi 12/04 (heure à déterminer)

# PROJET PPII - Compte-rendu N°5

<u>Motif / type de chantier :</u> Réunion de chantier	<u>Lieu :</u> S0.qqch + Mathis en ligne
<u>Présents :</u> Mathis MANGOLD Hélène BARBILLON Adrien LAROUSSE Aurélien GINDRE	<u>Date :</u> 12/05/2023 <u>Durée :</u> 13h-14h

## Ordre du jour :

- Fork (Mathis)
- Partie simulation (Aurélien)
- Affichage de la simulation (Adrien)
- Tests
- Ce qu'il reste à faire
- Rapport du projet

## Résumé de la réunion :

- Fork de Mathis :

Mathis explique le fork = plusieurs programmes qui tournent en même temps, c'est plus rapide que si on lançait un programme plusieurs fois d'affilées, puisque plus de ressources de l'ordinateur sont utilisées.

Aurélien demande si on ne peut pas plutôt allouer plus de ressources au programme pour le lancer plusieurs fois d'affilée : c'est possible mais compliqué et pas l'approche qui a été choisie.

Adrien demande si ça ne pose pas problème d'utiliser 100% de la capacité de l'ordinateur. Mathis veut exploser les ordinateurs.

2ème option : le multithreading = permet de créer aussi des autres processus appelés threads (mais ce n'est pas un processus à part entière qui tourne à côté, ça utilise un peu moins de mémoire).

Il faudra mettre la fonction qui permet de calculer les trajets dans un autre fichier que le main.c sinon on ne peut pas la tester.
- Partie simulation (Aurélien) :

L'implémentation de la file d'attente est presque finie.

Le plus gros soucis que pose le multithreading est que pour la partie simulation, il y a besoin du trajet précédent pour le suivant, on ne peut pas tout faire en parallèle.

Il faut aussi modifier ce qui est renvoyé en sortie pour que ce soit exploitable pour la partie affichage de la simulation. Plusieurs formats pour la sortie ont été évoqués, on retient celui envoyé par Aurélien sur discord.

Aurélien va modifier les indices id\_bornes de 0 à 17000 au lieu de 0 à 63000.
- Affichage de la simulation (Adrien) :

Une page web en plus a été ajoutée au site qui nous permet d'afficher les trajets, naviguer entre les ticks ne sera pas un problème si les données sont bien envoyées sur le site.

Est-ce que ce sera l'utilisateur qui choisira le nombre de trajets qu'il lance ? Oui, il faut donc rajouter des entrées utilisateurs.

Il reste également à personnaliser les points affichés sur la carte en fonction de si les bornes sont en attente ou occupées, on ne représentera pas les bornes vides.
- Tests :

Contenu du dossier app : src, data, test

Oster nous a aidé à faire fonctionner le makefile (ce boss)

On teste 2 choses : le temps de calcul de l'algorithme de la partie 1 et la précision de la fonction de calcul de distance (distance du trajet trouvée comparée à celle à vol d'oiseaux).
- Ce qu'il reste à faire :
  - génération aléatoire de trajets (utiliser une base de donnée des villes en France, sinon des points aléatoires dans le fichier des bornes) | Hélène

- finir programme simulation | Aurélien
  - sortie de la simulation (ce qu'on envoie pour l'affichage) | Aurélien
  - implémentation du multithreading dans la fonction de simulation | Mathis
  - affichage simulation (fichier txt, chaque ligne correspond à un tick, 1er caractère tick, un caractère séparateur entre bornes + données) | Adrien
- Rapport du projet :
- On va commencer à rédiger le rapport sur leaf. Aperçu des parties du rapport :
- état de l'art
  - partie trajet
  - partie simulation
  - tests (complexité mémoire et temps)

### TO DO list :

Description	Personne(s) concernée(s)	Date
affichage simulation	Adrien	prochaine réu
finir partie simulation+gérer sorties	Aurélien	prochaine réu
multithreading	Mathis	prochaine réu
génération aléatoire de trajets	Hélène	prochaine réu
Rédaction du rapport	Tout le monde	24/05

Prochaine réunion : mercredi après-midi



# PROJET PPII - Compte-rendu N°6

<u>Motif / type de chantier :</u> Réunion de chantier	<u>Lieu :</u> 1.2
<u>Présents :</u> Mathis MANGOLD Hélène BARBILLON Adrien LAROUSSE Aurélien GINDRE	<u>Date :</u> 17/05/2023 <u>Durée :</u> 14h-15h

## Ordre du jour :

- Partie simulation (Aurélien)
- Génération aléatoire (Hélène & Mathis)
- Rapport du projet

## Résumé de la réunion :

- Partie simulation (Aurélien) :  
Les fonctions sont en train d'être adaptées pour recevoir les données générées par le fork/threading.  
La file d'attente pose des soucis à cause de différentes situations expliquées au travers d'un diaporama.  
Il reste à se coordonner avec Mathis pour le type de retour de sa fonction
- Tests (Mathis) :  
Pour réaliser 20 trajets il faut 9 secondes pour l'algorithme de Mathis, Adrien ne trouve ce résultat pas très optimisé par rapport à un lancement 1 par 1.  
A cause de la longue durée de génération, il faudra lancer les tests en amont pour avoir un jeu de données conséquent.
- Génération trajets aléatoires (Hélène) :  
Hélène explique des bizarreries de la fonction random en C, par rapport à la seed qui reste statique si on ne la change pas manuellement.  
Elle explique aussi le fonctionnement des choix de trajet par rapport à la moyenne des trajets aléatoires.  
Il reste à ajouter un paramètre de distance minimale pour la génération.
- Rapport :  
Répartition de la rédaction du rapport :  
Adrien : 1er algorithme de trajet (et structures originelles)  
Mathis : modifications de l'algorithme et mode "optimisation temps" de l'algorithme  
Hélène : état de l'art, gestion de projet  
partie simulation : génération aléatoire (Hélène), génération trajets (Mathis), structures et fonctionnement (Aurélien)

## TO DO list :

Description	Personne(s) concernée(s)	Date
affichage simulation	Adrien	Fin projet 24/05
finir partie simulation+gérer sorties	Aurélien	Fin projet 24/05
entrées pour algo simulation	Mathis	Fin projet 24/05
Rédaction du rapport	Tout le monde	Fin projet 24/05