Rapport final de Projet Pluridisciplinaire d'Informatique Intégrative

Développement d'un Réseau de Recharge de Véhicules Électriques

Alexis MARCEL
Lucas LAURENT
Noé STEINER
Mathias AURAND-AUGIER

Responsable du module :
Olivier FESTOR
Gerald OSTER

24 Mai 2023

Table des matières

1 Contexte du projet

Ce rapport rend compte du Projet Pluridisciplinaire d'Informatique Intégrative dans le cadre de la première année du cycle ingénieur à TELECOM Nancy. L'objectif de ce projet est de concevoir, en groupe, une application en C dédiée à la simulation d'un réseau de recharge de véhicules électriques. Ce projet est encadré par M. Olivier Festor et M. Gérald Oster.

2 Introduction

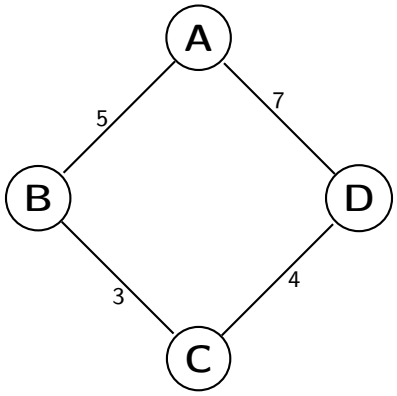
Ce rapport présente notre projet de développement d'un réseau de recharge de véhicules électriques, en réponse à l'interdiction récente de la Commission européenne de mettre sur le marché des véhicules à moteur thermique à partir de 2023. Notre objectif est de fournir un ensemble de fonctions utiles aux usagers, aux autorités de régulation et aux acteurs économiques pour faciliter le déploiement et le dimensionnement d'un réseau de recharge adapté aux besoins.

3 Structures de Données

3.1 Graph

3.1.1 Présentation

Nous avons choisi de représenter le réseau de stations de recharge à l'aide d'un graphe, une structure de données couramment utilisée pour modéliser des systèmes de points connectés. Chaque sommet représente une station de recharge et chaque arête représente un chemin direct entre deux stations. La pondération de chaque arête est la distance entre les deux stations correspondantes. Ainsi, la représentation ci-dessous, est une représentation graphique simplifiée de la structuration de notre graph. A, B, C et D sont des stations, la valuation des arcs correspond au nombre de kilomètres à vol d'oiseau entre les stations. Ainsi, la valuation de l'arc entre A et B est de 5, entre B et C de 3, entre C et D de 4 et entre D et A de 7 :



3.1.2 Implémentation

La structure Graph est composée de deux champs : V, qui représente le nombre de sommets du graphe, et adjMat, qui est une matrice d'adjacence représentant les arêtes du graphe. La matrice d'adjacence est un tableau à deux dimensions de taille V*(V+1)/2-V, où chaque élément est un entier représentant la pondération de l'arête correspondante. Si deux sommets ne sont pas connectés, la valeur de l'élément correspondant est -1 :

```
1 typedef struct Graph {
2     int V;
3     int* adjMat;
4 } Graph;
```

Listing 1 – Structure du Graph

3.1.3 Spécification algébrique

— $createGraph : Z \rightarrow Graph^*$
— $printGraph : Graph^* \rightarrow void$
— $createGraphFromStations : ChargingStation^* \times Z \rightarrow Graph^*$
— $freeGraph : Graph^* \rightarrow void$
— $dijkstra : void^* \rightarrow void^*$
— $printPath : ChargingStation^* \times Z^* \times Z \times Coordinate^* \times Coordinate^* \rightarrow void$
— $serializeGraph : Graph^* \times char^* \rightarrow void$
— $deserializeGraph : char^* \times Z \rightarrow Graph^*$

3.1.4 Parallélisation

Afin de faciliter l'instanciation du graph, nous avons également mis en place une solution pour la création des arêtes du graph. En effet, la création des arêtes du graph est une opération coûteuse en temps, et nous avons donc décidé de la parraléliser. Pour cela, nous avons réalisé ces 2 structures additionnelles :

```
1  typedef struct {
2      ChargingStation* stations;
3      int start;
4      int end;
5      Graph* graph;
6  } ThreadParamsGraph;
7
8  typedef struct {
9      Graph* graph;
10     ChargingStation* stations;
11     int autonomy;
12     int range;
13     Coordinate* src;
14     Coordinate* dest;
15     int* n;
16 } ThreadParamsDijkstra;
```

Listing 2 – Structure Annexes à Graph

La structure ThreadParamsGraph est conçue pour permettre le partage des paramètres nécessaires pour la création parallèle des arêtes du graphe. Elle contient un pointeur vers un tableau de stations de charge, des indices start et end définissant la plage de stations pour lesquelles le thread doit créer des arêtes, ainsi qu'un pointeur vers le graphe dans lequel les arêtes seront créées.

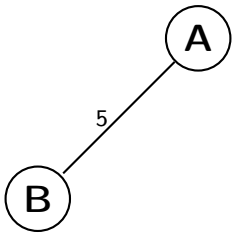
D'autre part, la structure ThreadParamsDijkstra est utilisée pour le partage des paramètres nécessaires à l'exécution parallèle de l'algorithme de Dijkstra. Elle contient un pointeur vers le graphe sur lequel l'algorithme doit être exécuté, un pointeur vers un tableau de stations de charge, ainsi que les paramètres autonomy et range utilisés dans l'algorithme. De plus, elle contient des pointeurs vers les coordonnées source et destination pour le chemin à trouver par l'algorithme de Dijkstra, ainsi qu'un pointeur vers un entier n qui stockera la longueur du chemin trouvé.

Ces structures permettent une modularité et une flexibilité accrue lors de la mise en place de la parallélisation dans le traitement du graphe, ce qui peut conduire à une amélioration significative des performances, en particulier pour les grands graphes.

3.2 ChargingStation

3.2.1 Présentation

La structure ChargingStation représente une station de recharge, avec des informations telles que le nom de la station, les coordonnées géographiques, le nombre de points de charge et le nombre de points de charge disponibles. Elle contient également une file d'attente pour gérer les véhicules en attente de recharge. Les ChargingStation sont utilisés comme noeuds du graphe :



3.2.2 Implémentation

La structure ChargingStation est composée de cinq champs : name, qui est un pointeur vers une chaîne de caractères contenant le nom de la station, coord, qui est un pointeur vers une structure Coordinate contenant les coordonnées géographiques de la station, nbChargingPoints, qui est un entier représentant le nombre de points de charge de la station, nbAvailableChargingPoints, qui est un entier représentant le nombre de points de charge disponibles, et queues, qui est un tableau de pointeurs vers des files d'attente, une pour chaque point de charge.

```
1 typedef struct ChargingStation {
2     char* name;
3     Coordinate* coord;
4     int nbChargingPoints;
5     int nbAvailableChargingPoints;
6     Queue** queues ;
7 } ChargingStation;
```

Listing 3 – Structure ChargingStation

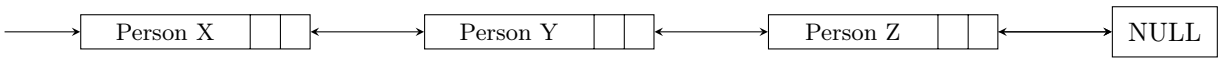
3.2.3 Spécification algébrique

- $readJSONstations : char^* \times Z \rightarrow ChargingStation^*$
- $serializeStations : char^* \times ChargingStation^* \times Z \rightarrow void$
- $deserializeStations : char^* \times Z \rightarrow ChargingStation^*$
- $addPersonToStation : ChargingStation^* \times Person^* \times Z \times Z \rightarrow void$
- $removePersonFromStation : ChargingStation^* \times Person^* \rightarrow void$
- $getBestQueue : ChargingStation^* \rightarrow Queue^*$

3.3 Queue

3.3.1 Présentation

La structure Queue est utilisée pour gérer la liste des véhicules en attente de recharge à une station donnée. Elle est basée sur une liste doublement chaînée, qui permet des opérations d'ajout et de suppression efficaces à la fois en tête et en queue de liste :



3.3.2 Implémentation

La structure Queue est composée de trois champs : data, qui est un pointeur vers une structure Person contenant les informations sur le véhicule en attente, next, qui est un pointeur vers la file d'attente suivante, et prev, qui est un pointeur vers la file d'attente précédente.

```
1 typedef struct Queue {
2     Person* data;
3     struct Queue* next;
4     struct Queue* prev;
```

```
5 } Queue;
```

Listing 4 – Structure Queue

3.3.3 Spécification algébrique

- $createQueue : void \rightarrow Queue^*$
- $del\_person : Queue^* \rightarrow void$
- $push : Queue^* \times Person^* \times Z \rightarrow void$
- $last : Queue^* \rightarrow Person^*$
- $index\_of\_from : Queue^* \times Z \rightarrow Person^*$
- $timeToWait : Queue^* \rightarrow Z$
- $first : Queue^* \rightarrow Person^*$
- $pop : Queue^* \rightarrow void$

4 Fonctionnalités et algorithmes

4.1 Algorithme de Dijkstra

4.1.1 Présentation

Pour déterminer le parcours optimal d'une station à une autre, nous avons utilisé l'algorithme de Dijkstra. C'est un choix naturel pour ce problème, car il trouve le chemin le plus court entre deux sommets d'un graphe pondéré, ce qui est exactement ce dont nous avons besoin pour minimiser la distance de conduite et donc la consommation d'énergie. Nous avons également exploré un potentiel choix alternatif, l'algorithme A*, mais nous avons finalement décidé de nous en tenir à Dijkstra, en implémentant une version parallèle pour améliorer les performances, mais également en retirant un grand nombre de sommets du graphe pour réduire le temps d'exécution.

4.1.2 Analyse de compléxité

La complexité de l'algorithme de Dijkstra est en général de $O(V^2)$, où V est le nombre de sommets du graphe. Cependant, si le graphe est implémenté à l'aide d'une liste d'adjacence et d'une file de priorité, la complexité peut être réduite à O((V+E) log V), où E est le nombre d'arêtes. Dans notre cas, nous avons utilisé une matrice d'adjacence pour représenter le graphe, donc la complexité est de $O(V^2)$. Cependant, nous avons également utilisé une version parallèle de l'algorithme, qui réduit considérablement le temps d'exécution. En effet, l'algorithme de Dijkstra est composé de deux boucles imbriquées, et la boucle interne peut être parallélisée, ce qui permet d'obtenir une complexité de $O(V^2/p)$, où p est le nombre de threads utilisés. En outre, nous avons également réduit le nombre de sommets du graphe, ce qui réduit encore le temps d'exécution.

4.2 Instanciation du Graph

4.2.1 Load des stations

Pour instancier le graphe, nous avons besoin de charger les stations de charge depuis un fichier JSON. Pour cela, nous avons utilisé la bibliothèque cJSON, qui permet de lire et d'écrire des fichiers JSON en C. Nous avons créé une fonction readJSONstations qui prend en paramètre le nom du fichier JSON et le nombre de stations à charger, et qui renvoie un tableau de stations de charge. Cette fonction utilise la bibliothèque cJSON pour lire le fichier JSON et créer un tableau de stations de charge à partir des données du fichier. Elle renvoie ensuite ce tableau.

4.2.2 Instanciation de la matrice d'adjacence

Une fois les stations de charge chargées, nous pouvons instancier la matrice d'adjacence. Pour cela, nous avons créé une fonction createGraphFromStations qui prend en paramètre un tableau de stations de charge et le nombre de stations, et qui renvoie un graphe. Cette fonction crée un graphe vide, puis crée les arêtes du graphe en utilisant la fonction createGraphFromStations. Enfin, elle renvoie le graphe.

4.2.3 Analyse de compléxité

La fonctionnalité d'instanciation du graphe est composée de deux fonctions majeures : createGraphFromStations et createGraphFromStationsThread.

createGraphFromStationsThread est une fonction qui remplit une partie de la matrice d'adjacence du graphe. En particulier, chaque thread exécute deux boucles imbriquées, où l'index i varie de start à end - 1 et l'index j varie de i + 1 à end. Par conséquent, cette partie a une complexité de $O((end - start)^2)$. Dans le cas le plus équilibré, chaque thread doit gérer $n/numThreads$ sommets, ce qui donnerait une complexité d'environ $O((n/numThreads)^2)$ pour chaque thread.

Cependant, l'aspect important de la parallélisation est que les threads s'exécutent simultanément. Par conséquent, bien que chaque thread individuel puisse avoir une complexité de $O((n/numThreads)^2)$, l'ensemble de l'opération createGraphFromStations a une complexité approximative de $O((n/numThreads)^2)$.

Il est à noter que l'efficacité de la parallélisation dépend de nombreux facteurs, dont la disponibilité des ressources du système, la gestion des threads par le système d'exploitation et la concurrence d'accès aux ressources partagées. Par conséquent, dans la pratique, le gain de performance peut ne pas être aussi important que suggéré par cette analyse.

En ce qui concerne la complexité spatiale, chaque thread n'utilise que très peu de mémoire supplémentaire (juste quelques variables locales). Ainsi, la complexité spatiale globale est dominée par la taille du graphe, qui est $O(n^2)$, où n est le nombre de sommets (stations) dans le graphe, car nous utilisons une matrice d'adjacence pour représenter le graphe.

5 Tests

5.1 Introduction

Nous avons effectué des tests sur chacune de nos fonctions pour nous assurer qu'elles fonctionnent correctement. Ces tests comprenaient des cas de test simples ainsi que des tests de stress pour évaluer la performance et l'efficacité des fonctions. En outre, nous avons utilisé des outils d'analyse dynamique pour détecter les fuites de mémoire et autres problèmes liés à la gestion de la mémoire.

6 Conclusion

En conclusion, ce projet s'est révélé être une expérience d'apprentissage extrêmement précieuse, nous permettant d'affiner et de développer nos compétences en programmation en langage C. Nous avons approfondi notre compréhension des structures de données, notamment comment les utiliser efficacement pour manipuler, stocker et accéder aux informations. Ce projet nous a aussi donné l'opportunité d'appliquer des algorithmes complexes, comme celui de Dijkstra, dans un contexte réel, nous permettant d'apprécier l'importance de ces outils dans la résolution de problèmes concrets.

En travaillant sur un problème réel - l'optimisation du placement des stations de charge pour véhicules électriques - nous avons pu saisir l'importance des implications de notre travail. Cela a ajouté une dimension plus large à notre apprentissage, en nous faisant comprendre comment les compétences techniques que nous avons acquises peuvent avoir un impact sur des questions d'une grande pertinence sociétale, comme la transition vers une mobilité plus durable.

De plus, la mise en évidence des enjeux de l'énergie durable dans notre projet nous a permis de nous immerger dans un secteur d'une importance cruciale pour l'avenir de notre société. L'efficacité de la distribution de l'énergie, en particulier pour les véhicules électriques, est une question clé dans la lutte contre le changement climatique. En participant à la résolution de ce problème à travers notre projet, nous avons non seulement appliqué nos compétences techniques, mais également contribué à un domaine qui aura un impact durable et positif sur notre avenir.

En somme, ce projet nous a offert une occasion inestimable de croissance personnelle et professionnelle. Il nous a permis d'acquérir et de consolider des compétences techniques, tout en nous faisant prendre conscience de l'importance des implications de notre travail dans le monde réel, en particulier dans le domaine de la mobilité et de l'énergie durables.

7 Annexes

Compte rendu de réunion

Noé Steiner - Alexis Marcel - Lucas Laurent - Mathias Aurand-Augier

5 Avril 2023

Projet PPII - Compte rendu n°01 - réunion de lancement

Participants:	Lieu:
• Alexis : Présent • Noé : Présent • Lucas : Présent • Mathias : Présent	• Le 5 avril 2023 • De 10h à 12h • Visioconférence sur Discord

Ordre du jour:

- Présentation du projet et des objectifs
- Répartition des tâches
- Établissement du calendrier prévisionnel

Compte rendu de réunion

Noé Steiner - Alexis Marcel - Lucas Laurent - Mathias Aurand-Augier

19 Avril 2023

Projet PPII - Compte rendu n°02 - réunion d'avancement

Participants:	Lieu:
• Alexis : Présent • Noé : Absent • Lucas : Présent • Mathias : Présent	• Le 19 avril 2023 • De 14h à 16h • Visioconférence sur Discord

Ordre du jour:

- Mise à jour sur le progrès de la Partie 1
- Sérialisation des données pour gagner en performance
- Discussion des problèmes rencontrés et des solutions possibles

Compte rendu de réunion

Noé Steiner - Alexis Marcel - Lucas Laurent - Mathias Aurand-Augier

3 Mai 2023

Projet PPII - Compte rendu n°3 - réunion technique

Participants:	Lieu:
• Alexis : Présent • Noé : Présent • Lucas : Absent • Mathias : Présent	• Le 3 mai 2023 • De 15h à 17h • Visioconférence sur Discord

Ordre du jour:

- Établir un plan pour la deuxième étape du projet
- Discussion sur la logique de programmation pour la simulation de plusieurs utilisateurs et la gestion des files d'attente
- Etudes des différentres structures de données possibles

Compte rendu de réunion

Noé Steiner - Alexis Marcel - Lucas Laurent - Mathias Aurand-Augier

17 Mai 2023

Projet PPII - Compte rendu n°4 - réunion d'avancement

Participants:	Lieu:
• Alexis : Présent • Noé : Présent • Lucas : Présent • Mathias : Présent	• Le 17 mai 2023 • De 14h à 16h • Visioconférence sur Discord

Ordre du jour:

- Présentation des avancements sur le module de simulation
- Discussion des problèmes rencontrés et des solutions possibles
- Préparation de la présentation finale du projet

Compte rendu de réunion

Noé Steiner - Alexis Marcel - Lucas Laurent - Mathias Aurand-Augier

24 Mai 2023

Projet PPII - Compte rendu n°5 - réunion de clôture

Participants:	Lieu:
• Alexis : Présent • Noé : Présent • Lucas : Présent • Mathias : Présent	• Le 24 mai 2023 • De 10h à 12h • Visioconférence sur Discord

Ordre du jour:

- Révision du travail accompli
- Discussion des points à améliorer pour les futurs projets
- Clôture du projet