École Nationale Supérieure d'Informatique
pour l'Industrie et l'Entreprise

Projet Recherche
Rapport

Optimisation du dernier kilomètre de livraison d'Amazon

Élèves :
Adèle Berger
Manon Chartrin
Colin Coërchon

Enseignants :
Massinissa Merabet

24 mai 2024

Table des matières

1 Introduction

Dans le cadre du cours « Projet Recherche », l'objectif principal est de nous initier à un aspect de la recherche. Ici, nous nous sommes appuyés sur un article intitulé *Solving Vehicle Routing Problem with linear programming optimized by Machine Learning* [3] dont les principales idées sont reprises en français dans un rapport très détaillé [4].

Ce projet provient d'un challenge organisé par *Amazon* et le *MIT Center for Transportation and Logistics* [5]. L'objectif était de développer des approches innovantes pour produire des solutions au problème de séquençage d'itinéraires [7].

1.1 Contexte

Amazon a un point faible quant à la livraison : il s'agit de **la livraison du dernier kilomètre**. En effet, des algorithmes d'optimisation ont permis de trouver des routes qui optimisent les trajets des livreurs et qui permettent ainsi de minimiser le temps de trajet tout en maximisant le profit. Cependant, lors du dernier kilomètre, l'entreprise s'est aperçue que la route n'est optimale que dans la théorie. Dans la réalité, le livreur fait un choix différent de celui proposé par l'algorithme. Cela peut s'expliquer par des travaux, des problèmes pour se garer ou encore des sens interdits non pris en compte.

Ainsi, le but de ce projet est d'arriver à trouver une route qui se rapproche de l'optimalité, de façon à ce que le dernier kilomètre soit en adéquation avec le choix du livreur.

1.2 Objectif du projet

Dans ce projet, nous utilisons le dataset d'entraînement fourni par Amazon. Afin de trouver la route optimale, nous avons plusieurs étapes à réaliser :

- Tout d'abord, nous devons **mettre en forme les données brutes** fournies par Amazon afin d'avoir toutes les informations nécessaires disponibles clairement.
  ↪ Choix dans la structuration des données.

- Ensuite, pour chaque route, nous devons récupérer la station et faire un **clustering** sur les arrêts dans le but de diviser l'ensemble des arrêts en différents groupes.
  ↪ Choix dans l'heuristique du clustering.

- Une fois que tous les arrêts sont divisés en sous-groupes d'arrêts, l'objectif est de déterminer **la route finale entre les différents groupes** ("clusters").
  ↪ Choix dans l'algorithme de plus court chemin à appliquer.

- À l'aide de cette route entre les différents "groupes d'arrêts", il faut aussi déterminer le premier et le dernier arrêt de chaque sous-groupe.
  ↪ Choix de l'heuristique à appliquer.

- Avec toutes ces informations, nous pouvons **appliquer le PLNE dans chaque sous-groupe**. Et ainsi déterminer la route optimale à l'intérieur de chaque sous-groupe.
  ↪ Choix du modèle mathématique pour le PLNE.

Grâce aux étapes précédentes, nous obtenons ainsi **notre route totale**. Ces étapes sont synthétisées dans le schéma ci-dessous.

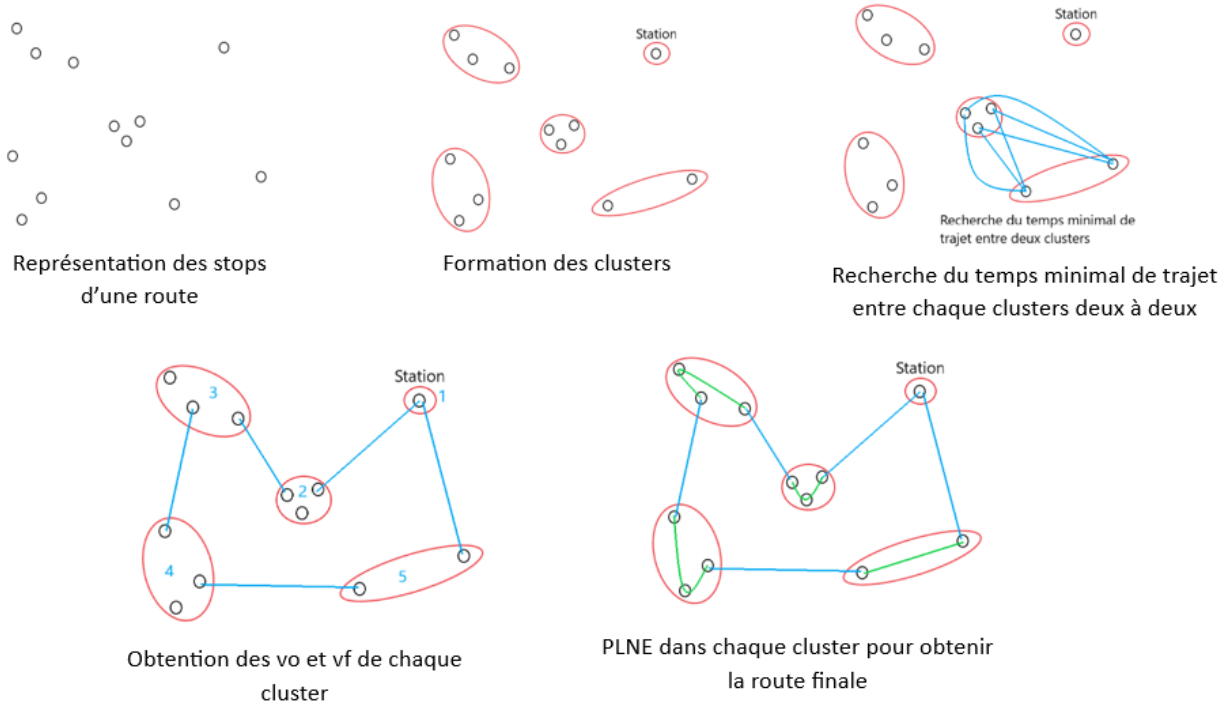


FIGURE 1 – Etapes de la recherche d'une route optimale

1.3 Choix du solveur et de l'environnement de travail

Afin de mener à bien ce projet, nous avons eu besoin de configurer notre environnement. En effet, nous avons réalisé ce projet en Python à l'aide de la bibliothèque **PuLP** [1] et du logiciel **CPLEX** [2]. Nous avons également utilisé la bibliothèque **Networkx** afin de visualiser nos solutions sous forme de graphes.

Enfin, nous avons évidemment créé un git qui nous a permis de coder à plusieurs plus facilement.

2 Découverte et mise en forme des données

2.1 Présentation des données

Dans ce projet, nous avons eu accès aux données qui étaient disponibles aux participants du challenge d'Amazon [5]. Nous avons du mettre en forme ces données qui étaient contenues dans 4 fichiers : `route_data.json`, `package_data.json`, `travel_times.json` et `actual_sequences.json`.

Les figures (2), (3), (4) et (5) donnent une présentation simplifiée de ces données brutes brutes récupérées.

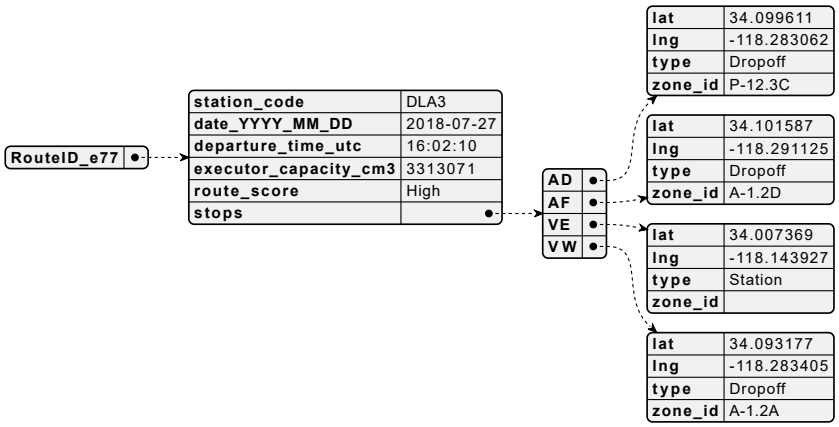


FIGURE 2 – Informations brutes contenues dans `route_data.json`

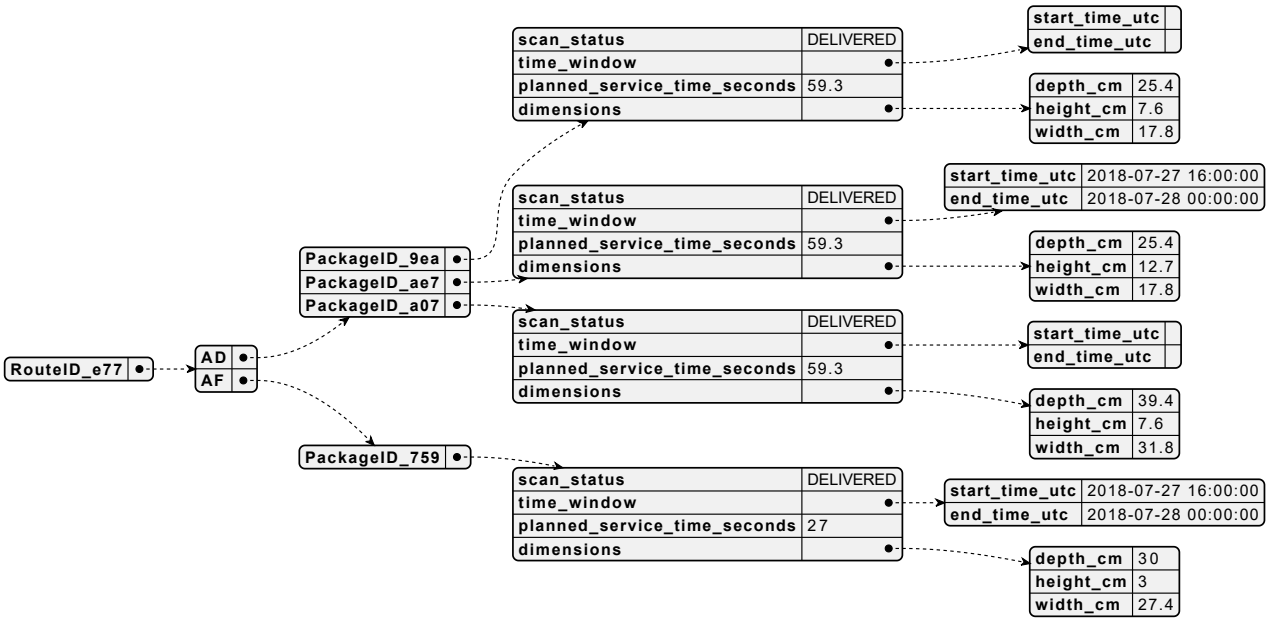


FIGURE 3 – Informations brutes contenues dans `package_data.json`

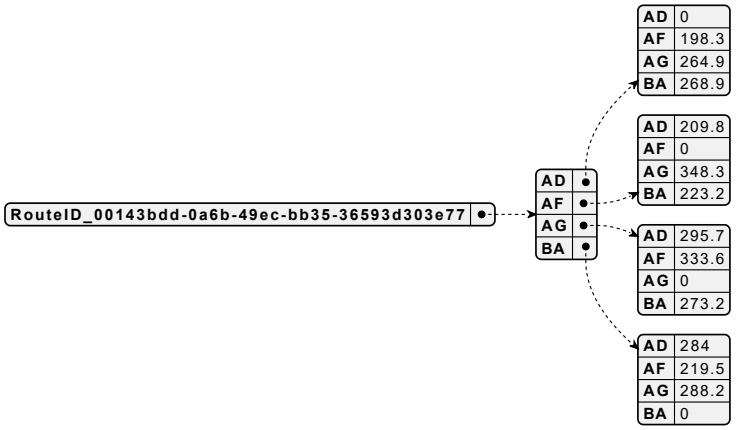


FIGURE 4 – Informations brutes contenues dans `travel_times.json`

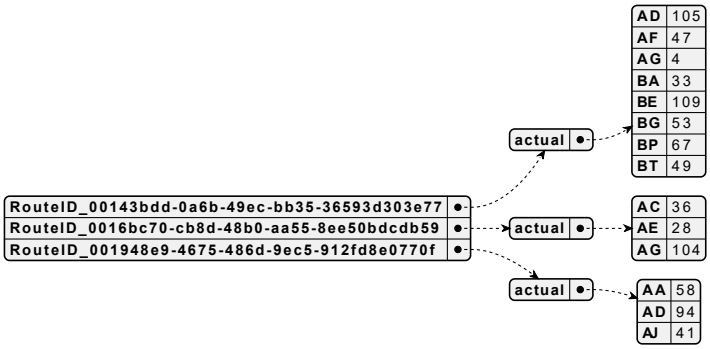


FIGURE 5 – Informations brutes contenues dans `actual_sequences.json`

Dans les 3 premiers fichiers, il y a des informations sur différentes **routes**. Chaque route possède, en particulier, un identifiant de route et **une liste de stops**. Ces stops sont chacun représentés par un identifiant, un emplacement, un numéro de zone et un type ("Dropoff" ou "Station"). Sur chaque stop, on trouve également diverses informations sur **les packages à livrer**. Les détails de toutes les informations sur chaque package sont donnés sur la figure (3).

Pour finir, le dernier fichier, `actual_sequences.json`, nous a permis de récupérer les informations nécessaires à la comparaison de nos résultats théoriques optimisés avec **le chemin réellement parcouru par le livreur** durant son travail sur cette route.

Nous avons été confrontés à des données manquantes concernant les attributs **package.time**. Notre approche pour remédier à cette situation était la suivante : en premier lieu, nous vérifions la présence d'autres colis à livrer au même arrêt, pour lesquels des informations de temps étaient disponibles. Si tel était le cas, nous utilisions **ces informations similaires** pour combler les données manquantes. Cependant, en l'absence de temps spécifiés pour d'autres colis à cet arrêt, nous comblions les lacunes en utilisant **le temps minimum et maximum** observé sur l'ensemble de la route. Cette méthode garantissait que même en l'absence de données spécifiques

pour un colis à un arrêt donné, nous disposions de valeurs cohérentes pour assurer l'intégrité des calculs subséquents.

Toutes les informations relatives aux données récupérées ont été détaillées dans le tableau 1 ci-dessous.

Description des variables du Dataset	
Informations sur la route	
Code Station	une chaîne alphanumérique identifie de manière unique la station (ou le dépôt) où la route a commencé
Date	la date à laquelle le véhicule de livraison est parti de la station
Heure de Départ	l'heure à laquelle le véhicule de livraison est parti de la station, spécifiée en UTC
Capacité du Véhicule	la capacité en volume du véhicule de livraison, spécifiée en cm^3
Arrêts à desservir sur la route	séquence observée dans laquelle les arrêts sur la route ont été desservis
Score de la Route	variable catégorielle indiquant la qualité de la séquence d'arrêts observée (élevée, moyenne, faible)
Informations sur les arrêts	
ID de l'Arrêt	un code d'identification de chaque arrêt sur une route (unique dans chaque route)
Latitude/Longitude	latitude et longitude de chaque arrêt spécifiées via le système de projection WGS 84
Type	variable catégorielle indiquant le type d'arrêt (station ou livraison)
ID de la Zone	identifiant unique de la zone de planification géographique dans laquelle se trouve l'arrêt
Colis	colis à desservir à chaque arrêt
Distances	distances à tout autre arrêt sur la même route, moyenne des temps de trajet historiquement réalisés (en secondes)
Informations sur les colis	
ID du Colis	identifiant unique de chaque colis (unique dans l'ensemble des données)
Fenêtre de temps	heures de début et de fin : contraintes de fenêtre de temps de livraison sur certains colis
Temps de service prévu	temps que le service de ce colis est censé prendre
Dimensions	largeur, longueur, hauteur maximales du colis

TABLE 1 – Description des variables du dataset

Nous nous sommes alors posés la question sur la meilleure manière de mettre en forme nos données dans notre code. En pensant, d'abord, simplement à l'utilisation de gigantesques matrices avec la bibliothèque Pandas, puis en se penchant sur la solution de gros dictionnaires rassemblant toutes les informations, nous sommes finalement partis sur une autre idée : **les classes** !

Ainsi, nous avons récupéré ces données, puis nous les avons stockées et structurées dans différentes classes afin de pouvoir les exploiter par la suite.

2.2 Diagramme de classes

Afin de mener à bien ce projet, nous avons créé différentes classes :

`Route`, `Stop`, `Package` et `Cluster`.

Toutes les informations récupérées sont stockées dans ces classes, et même plus. En effet, nous stockons aussi l'ensemble des informations que nous accumulons lors du traitement de ces données (comme les données sur les "clusters" par exemple).

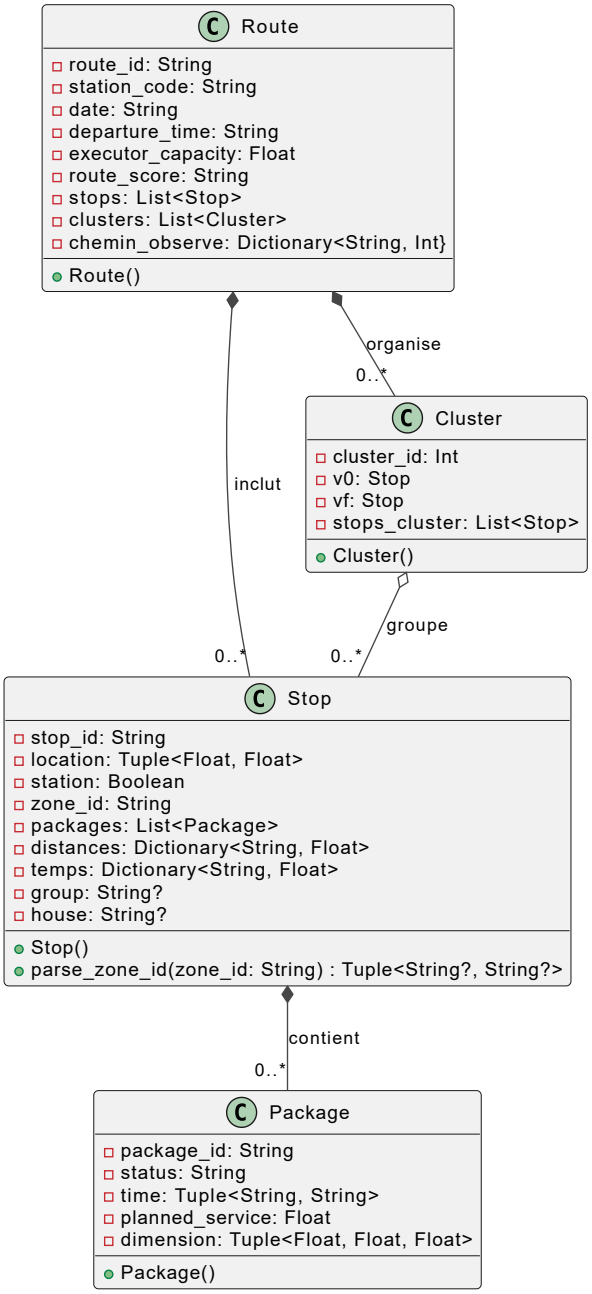


FIGURE 6 – Diagramme de classes de notre projet *(réalisé avec PlantText [6])*

La figure (6) permet de mettre en lumière tous les attributs de chacune de ces quatre classes, et elle montre surtout les **liens** qui les séparent. Voici un bref résumé de chacune d'elle :

- La classe **Stop** : Représente un arrêt sur une route, identifié par `stop_id`. Contient des informations telles que la localisation et si c'est une station ou non. Les arrêts sont associés à des paquets (`Package`) dans une relation de composition, indiquant que les paquets existent uniquement dans le contexte d'un arrêt.
- La classe **Package** : Détaille un paquet avec un identifiant unique (`package_id`) et des informations sur le statut, le temps de traitement et les dimensions. La classe est utilisée pour définir les éléments transportés pour chaque arrêt.
- La classe **Cluster** : Groupe plusieurs arrêts (`Stops`) qui sont logiquement connectés ou proches géographiquement (cf. notre logique de formation des clusters dans la partie 4.1). Une relation d'agrégation relie ainsi logiquement chaque stop à un et un seul cluster.
- La classe **Route** : Décrit une route complète avec un identifiant (`route_id`), incluant une liste d'arrêts (`Stop`) et de clusters (`Cluster`). Les arrêts et les clusters sont inclus dans une route par une relation de composition.

3 Structuration des fichiers du code

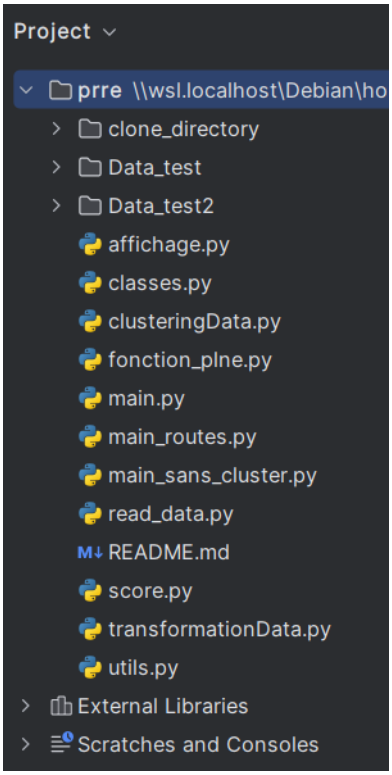


FIGURE 7 – Organisation des fichiers internes du projet

La figure (7) nous donne un aperçu global de l'ensemble des fichiers python de notre projet.

- **Récupération et Transformation des données** :
  - ▷ `read_data.py` : Chargement et lecture des données initiales (en fonction des données importées).
  - ▷ `transformationData.py` : Transformation et préparation des données pour le traitement. Contient aussi l'initialisation des instances de la classe `Cluster` une fois ceux-ci déterminés.
- **Utilitaires et Classes** :
  - ▷ `utils.py` : Fonctions utilitaires générales utilisées à travers le projet.
  - ▷ `classes.py` : Définition des 4 classes utilisées pour structurer les données récupérées (cf. partie 2.2).
- **Clustering et Optimisation** :
  - ▷ `ClusteringData.py` : Un fichier très important dans notre projet. Il y a dans ce fichier trois fonctions extrêmement importantes :

    - Implémentation de l'heuristique de clustering avec la fonction `cluster_zone_id` (cf. partie 4.1).
    - Implémentation de l'heuristique utilisée pour réaliser un problème du voyageur de commerce entre les différents clusters construits. Cela se fait à l'aide de la fonction `find_optimal_path` (cf. partie 4.2).
    - Choix des points de départ et d'arrivée dans chaque cluster avec la fonction `get_v0_vf_for_optimal_path` (cf. partie 4.3).
  - ▷ `fonction_plne.py` : Exécution de l'algorithme TSPTW avec CPLEX pour chaque cluster (cf. partie 4.4).

- **Affichage et Interface** :
  - ▷ `affichage.py` : Regroupe toutes les fonctions d'affichage pour les fichiers main.

- **Exécution Principale** :
  - ▷ `main.py` : Exécution principale pour une route unique (la première route).
  - ▷ `main_routes.py` : Gère l'exécution sur plusieurs routes (souvent les 3 premières routes).
  - ▷ `main_sans_cluster.py` : Test de l'algorithme TSPTW sur l'ensemble des stops d'une route sans pré-cluster.

- **Données de travail et Logs** :
  - ▷ Dossiers `Data_test` `Data_test2` et `100routes` : Contiennent des données pour tester l'exécution des fichiers main. Le premier possède des données extrêmement rudimentaires qui ont simplement été utilisées au tout début du projet. Très vite, nous avons utilisé les données réelles des trois premières routes fournies par Amazon (dans `Data_test2`). Enfin `100routes` contient les données des 100 premières routes d'Amazon.
  - ▷ `clone_directory` : Utilisé pour récupérer les logs de CPLEX lors de l'exécution de la programmation linéaire.

4 Détails des étapes du projet

4.1 Formation des clusters

Une fois les données mises en forme dans les différentes classes, il a été **nécessaire** (cf. partie 4.1.4) de réaliser différents clusters parmis les nombreux stops présents dans chaque route.

Dans chaque route, il y a plusieurs stops chez des particuliers ou autres (notés `Dropoff` dans les fichiers JSON dans la figure 2) et une seule et unique station. La station sera automatiquement assignée au **cluster 0**. Le cluster 0 sera donc toujours composé d'un seul stop. Une fois ce premier cluster déterminé, il a été nécessaire de définir une **heuristique** permettant de réaliser un clustering sur les autres stops afin de former différents groupes.

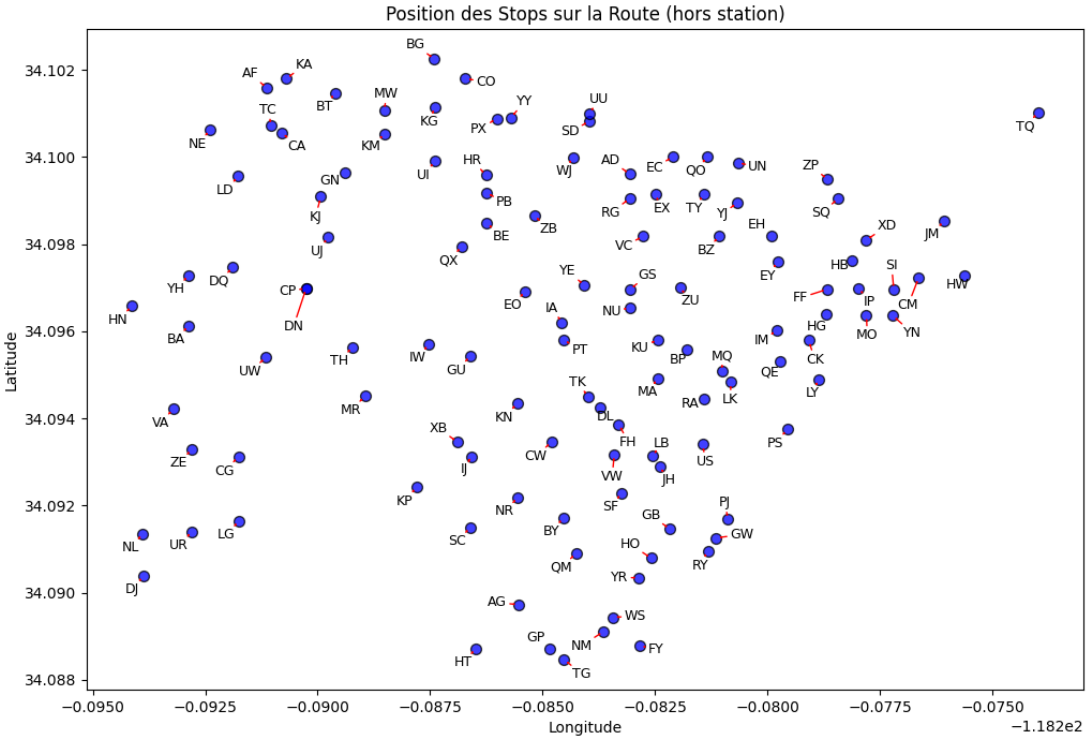


FIGURE 8 – Emplacement des stops de la première route (hors station)

4.1.1 Première approche

Pour réaliser des groupes des différents stops de notre route (hors station), notre première idée a été de s'intéresser à la donnée `zone_id` de chaque stop (cf. figure 2). En effet, après avoir affiché l'ensemble des stops de la route avec leurs `stop_id` respectifs (cf. figure 8), nous avons également affiché ces mêmes stops mais avec leurs `zone_id` :

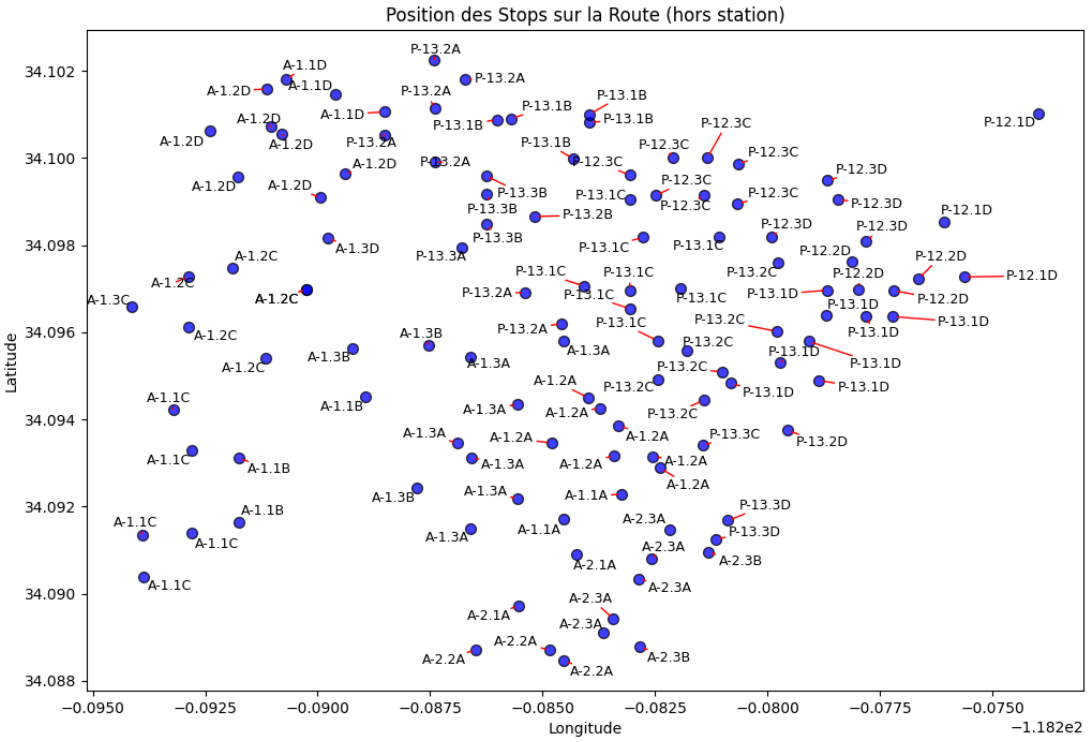


FIGURE 9 – Emplacement des stops de la première route (hors station) avec leurs `zone_id` repectifs

Ce qui ressort de la figure 9, c'est que les stops avec des `zone_id` similaires sont souvent très proches en termes de distance. **Réaliser nos premiers clusters à l'aide de ces** `zone_id` nous a ainsi semblé une très bonne idée.

Pour cela, nous avons considéré les données fournies par la `zone_id` de chaque stop et les avons découpées en deux pour les mettre dans la classe `Stop`. C'est de cette façon que sont nés les attributs `route.group` et `route.house` de la classe `Stop`.

Par exemple, la donnée `P-13.3C` devenait :

`route.group` = `P-13` et `route.house` = `3C`

Voici comment nous avons implémenté ceci en Python :

```
stations = [stop for stop in route.stops if stop.station]
non_station_stops = [stop for stop in route.stops if not stop.station]

initial_clusters = {}
for stop in non_station_stops:
    group = stop.group if stop.group is not None else 'Default'
    if group not in initial_clusters:
```

```
        initial_clusters[group] = []
    initial_clusters[group].append(stop)
```

Les résultats sur la première route sont donnés dans la figure 10.

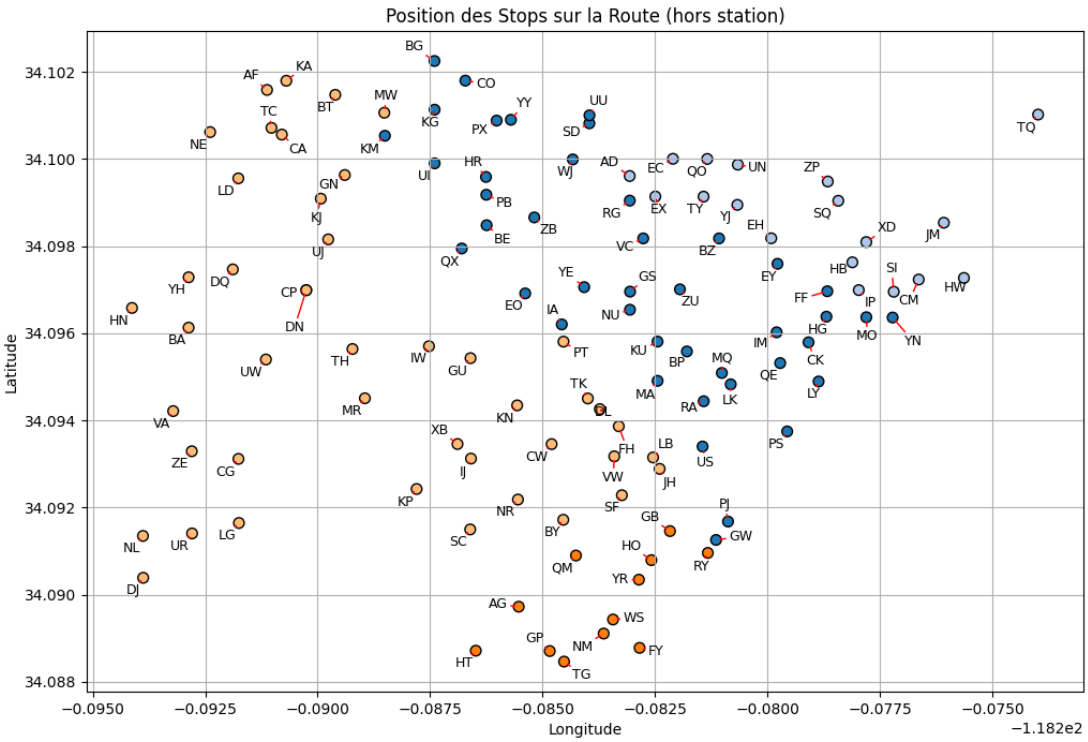


FIGURE 10 – Clustering naïf des stops par `zone_id`

On remarque que nous avons donc 4 clusters qui sont ressortis à l'aide de l'information donnée par `zone_id`. Mais ces clusters sont de très grande taille. Il va donc être nécessaire de les réduire. Il était dit dans l'article donné [3] [4] que la limite de stops par clusters, avant que le temps de calcul, n'explose était **vers les 15 stops par clusters**. Il va donc être nécessaire de s'assurer que nos clusters ne sont pas trop gros.

4.1.2 Heuristique finale

Nous avons nos clusters initiaux à l'aide des `zone_id`, et il est maintenant nécessaire de déterminer une heuristique pour réduire la taille de ces clusters initiaux (tout en s'assurant qu'ils ne deviennent pas trop petits).

Il est important de noter qu'on se base sur **les temps de trajets entre stops**, et non sur les distances qui les séparent.

Voici les principales étapes de l'heuristique choisie dans la fonction `cluster_zone_id` :

- **Création des Clusters initiaux** : Déjà vu dans la partie précédente 4.1.1.
- **Vérification de la taille** : Chaque cluster initial est examiné pour déterminer s'il dépasse le nombre maximal d'arrêts autorisés (`stops_max_per_cluster`).
- **Division des Clusters trop grands** : Si un cluster est trop grand, il est divisé en plusieurs sous-clusters.
  ↪ Cette division repose sur la fonction `AgglomerativeClustering` de la bibliothèque `sklearn.cluster`. Pour fonctionner correctement, elle utilise **la matrice de distance entre les différents stops** (accessible via les données `stop.temps` de chaque stop).
- **Arrêt en cas de conformité** : Le processus s'arrête si tous les clusters sont de taille inférieure au maximum spécifié. Ceci est une condition de terminaison qui assure que l'on ne procède pas à des divisions inutiles.
- **Gestion des Clusters trop petits** : Cependant, les clusters qui sont sous la taille minimale requise (`stops_min_per_cluster`) sont ensuite examinés. On les fait fusionner avec le cluster le plus proche jusqu'à atteindre la taille minimale requise.
- **Finalisation** : On retourne les clusters ainsi créés.

4.1.3 Résultats du clustering

Nous avons testé notre algorithme de clustering avec un nombre maximal de **15 stops par cluster**, et un nombre minimal de **3 stops par cluster**. La figure 11 et 12 donnent les premiers résultats pour les routes 1 et 3.

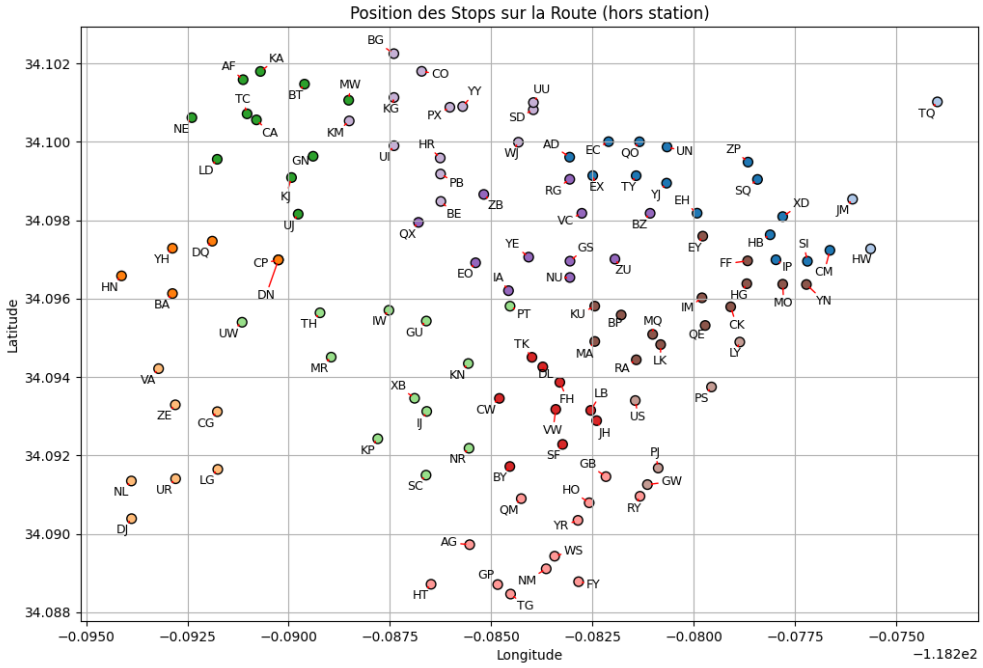


FIGURE 11 – Affichage du Clustering obtenu pour la première route

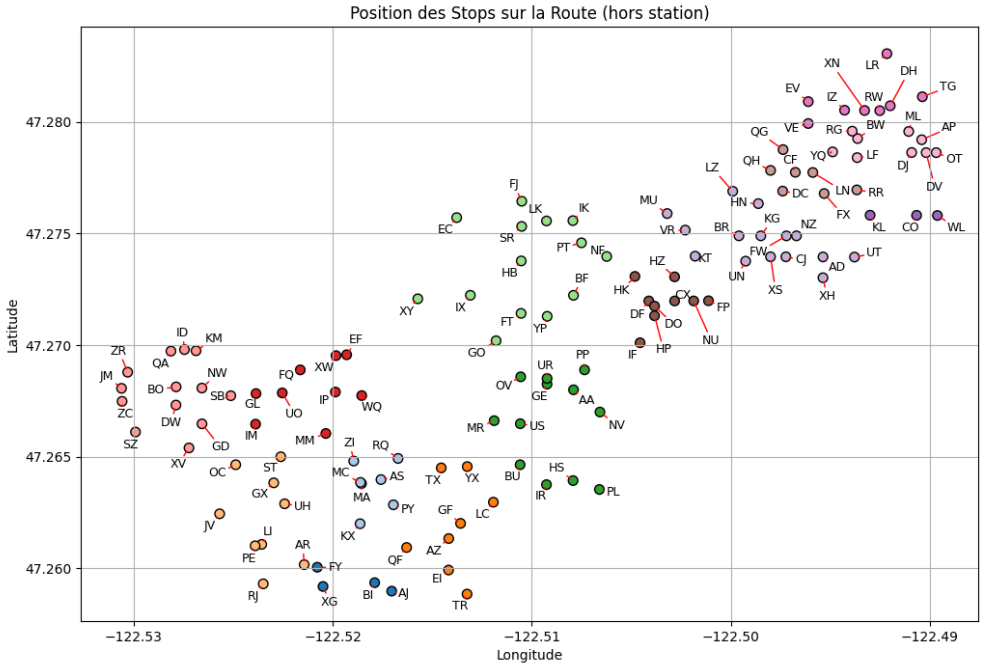


FIGURE 12 – Affichage du Clustering obtenu pour la troisième route

Dans le terminal, nous avons fait en sorte d'avoir également les résultats du clustering

affichés :

```
Nombre de clusters: 13
Temps pour clustering: 0.00 secondes
Cluster 0: ['VE']
Cluster 1: ['AD', 'CM', 'EC', 'EH', 'EX', 'HB', 'IP', 'QO', 'SI', 'SQ', 'TY', 'UN', 'XD', 'YJ', 'ZP']
Cluster 2: ['HW', 'JM', 'TQ']
Cluster 3: ['BA', 'CP', 'DN', 'DQ', 'HN', 'YH']
Cluster 4: ['CG', 'DJ', 'LG', 'NL', 'UR', 'VA', 'ZE']
Cluster 5: ['AF', 'BT', 'CA', 'GN', 'KA', 'KJ', 'LD', 'MW', 'NE', 'TC', 'UJ']
Cluster 6: ['GU', 'IJ', 'IW', 'KN', 'KP', 'MR', 'NR', 'PT', 'SC', 'TH', 'UW', 'XB']
Cluster 7: ['BY', 'CW', 'DL', 'FH', 'JH', 'LB', 'SF', 'TK', 'VW']
Cluster 8: ['AG', 'FY', 'GB', 'GP', 'HO', 'HT', 'NM', 'QM', 'RY', 'TG', 'WS', 'YR']
Cluster 9: ['BZ', 'EO', 'GS', 'IA', 'NU', 'QX', 'RG', 'VC', 'YE', 'ZB', 'ZU']
Cluster 10: ['BE', 'BG', 'CO', 'HR', 'KG', 'KM', 'PB', 'PX', 'SD', 'UI', 'UU', 'WJ', 'YY']
Cluster 11: ['BP', 'CK', 'EY', 'FF', 'HG', 'IM', 'KU', 'LK', 'MA', 'MO', 'MQ', 'QE', 'RA', 'YN']
Cluster 12: ['GW', 'LY', 'PJ', 'PS', 'US']
```

FIGURE 13 – Clustering obtenu pour la première route

On remarque dans la figure 13 que **le temps de calcul pour avoir ces clusters est quasiment immédiat** (moins de 0.01 secondes).

4.1.4 Justification du clustering

Pourquoi avons nous réalisé un clustering ?
Dans ce projet, nous avons eu besoin d'appliquer un PLNE. Cependant, nous travaillons avec un très grand jeu de données et **nous ne pouvons pas appliquer tel quel le PLNE sur toutes ces données**. Nous avons donc eu besoin de séparer nos données en différents groupes afin de pouvoir par la suite appliquer le PLNE sur chaque groupe de données.

En fait, en toute honnêteté, nous avons essayé de faire tourner le problème du TSPTW sur une route entière en oubliant volontairement l'étape de "grouper les stops" en différents clusters. Nous avons laissé tourner l'algorithme **durant 1 heure**, sans succès. Cela montre clairement l'importance de **créer des clusters** pour diviser notre problème en différents **sous-problèmes de plus petite taille**.

4.2 Algorithme de chemin des clusters

Après avoir effectué le **processus de clustering** sur nos données, nous avons développé une méthode pour **trouver le chemin optimal entre les clusters obtenus** *(ou du moins, un chemin pseudo-optimal)*. Cette méthode repose sur deux principaux composants : la **recherche des temps de trajet minimaux** entre chaque paire de clusters, suivie de **l'identification du chemin optimal** à travers ces clusters.

4.2.1 Recherche des temps de trajet minimaux entre les clusters

Pour calculer les temps de trajet minimaux entre les clusters, nous avons mis au point une fonction `find_min_travel_time_between_clusters(cluster1, cluster2)`. Cette fonction prend en entrée deux clusters et parcourt tous les arrêts des **deux clusters** pour déterminer le **temps de trajet minimal entre eux**. Nous avons exclu de cette recherche l'arrêt spécifié comme point de départ (`v0`) dans le premier cluster, car c'est la station.

4.2.2 Création de la matrice des temps de trajet minimaux

À l'aide de la fonction précédente, nous avons développé la méthode `create_min_travel_time_matrix(clusters)` pour générer une **matrice des temps de trajet minimaux** entre chaque paire de clusters. Cette matrice présente les temps de trajet bidirectionnels entre deux clusters, avec les temps de trajet diagonaux mis à zéro pour indiquer qu'un cluster n'a pas besoin de voyager vers lui-même.

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0.0	1719.0	1676.5	1696.2	1557.8	1722.7	1614.1	1597.4	1509.7	1676.2	1755.6	1624.6	1612.3
1	1869.3	0.0	127.3	217.6	298.9	134.5	107.7	126.6	192.9	17.0	38.4	17.7	109.7
2	1866.8	42.4	0.0	287.4	314.0	211.6	126.2	124.1	172.6	114.8	211.5	42.3	76.6
3	1908.4	242.8	294.4	0.0	95.2	61.5	64.5	227.6	282.6	153.2	179.5	203.9	238.1
4	1783.8	306.2	303.6	91.1	0.0	136.9	71.7	156.3	192.7	211.6	238.4	213.1	247.3
5	1922.1	148.9	236.5	55.7	137.0	0.0	100.7	178.5	222.2	57.3	12.6	146.0	162.2
6	1724.3	127.8	121.2	53.1	79.7	95.4	0.0	45.8	72.2	32.7	84.4	30.7	64.9
7	1732.8	137.9	119.2	215.8	171.8	168.5	43.8	0.0	72.1	89.0	157.0	22.3	61.1
8	1651.4	186.3	167.6	268.4	165.0	203.5	56.5	43.5	0.0	124.0	203.4	70.7	6.9
9	1827.3	9.3	131.8	157.5	215.1	41.7	27.3	83.7	127.4	0.0	41.6	39.1	67.4
10	1872.9	38.4	248.6	147.1	228.4	12.9	85.9	169.4	216.9	49.4	0.0	140.7	160.3
11	1764.8	27.5	49.6	197.7	223.7	121.9	36.5	22.1	88.4	25.1	121.8	0.0	33.2
12	1759.8	101.9	75.5	237.1	243.8	143.3	75.9	68.6	6.3	63.8	143.2	33.7	0.0

FIGURE 14 – Matrice des temps minimaux entre chaque cluster (pour la Route 1)

4.2.3 Détermination du chemin optimal

Une fois la matrice des temps de trajet minimaux établie, nous avons implémenté l'algorithme `find_optimal_path(matrix)` pour déterminer le chemin optimal à travers les clusters. Cet algorithme utilise la méthode du voyageur de commerce (TSP) et nous avons initialement envisagé une approche "brute-force". C'est-à-dire, **comparer toutes les solutions entre elles**. Cependant, pour 12 clusters, cette méthode s'avère excessivement longue et donc impraticable ($11! = 39\,916\,800$, *c'est assez énorme !*).

Nous avons alors développé deux fonctions alternatives utilisant des heuristiques plus rapides pour résoudre le TSP :

- `find_path1` : Implémente l'heuristique du **plus proche voisin**. Cette méthode gloutonne commence par le premier cluster et sélectionne à chaque étape le cluster non visité le plus proche, jusqu'à ce que tous les clusters soient visités.

- `find_path2` : Utilise l'heuristique d'**insertion du plus proche voisin**. Cette méthode, plus sophistiquée, examine l'impact de l'insertion de chaque cluster sur le coût total du trajet et choisit l'insertion qui minimise cet impact.

Ces méthodes nous permettent d'obtenir un chemin optimal à travers les clusters ainsi que le coût total du trajet de manière plus efficace et réaliste que l'approche "brute-force".

Voici le chemin optimal ainsi que le temps total minimal pour l'exemple que nous avons pris depuis le début :

```
Chemin optimal : [0, 8, 12, 2, 11, 7, 6, 4, 3, 9, 5, 10, 1, 0]
```

Ainsi, dans cet exemple, la route va passer par la station puis par les clusters 8, 12, 2, 11, 7, 6, 4, 3, 9, 5, 10 puis 1 avant de revenir à la station.

4.3 Trouver les points de départ et d'arrivée dans chaque cluster

La fonction `get_v0_vf_for_optimal_path` calcule les points de départ (`v0`) et d'arrivée (`vf`) pour chaque segment du chemin optimal, déterminé au préalable par la fonction `find_optimal_path`. Le point de départ (`v0`) du premier cluster du chemin optimal est déjà connu. Les étapes suivantes décrivent le processus :

1. **Initialisation** : Le `v0` du premier cluster du chemin optimal est déjà connu.

2. **Itération sur les clusters** : Pour chaque paire de clusters consécutifs dans le chemin optimal :

   (a) Utilisation de la fonction `find_min_travel_time_between_clusters` pour identifier les arrêts spécifiques (`vf` pour le cluster courant et `v0` pour le cluster suivant) qui minimisent le temps de trajet entre le cluster de départ (`start_cluster`) et le cluster d'arrivée (`end_cluster`).

   (b) La sous-routine `find_min_travel_time_between_clusters` :

   - Recherche le temps de trajet minimal en excluant l'arrêt spécifié comme `v0` dans le cluster de départ.
   - Retourne le couple d'arrêts (`vf` et `v0`) correspondant à ce trajet minimal.

   (c) Mise à jour des identifiants des arrêts `v0` et `vf` pour chaque cluster.

3. **Enregistrement des résultats** : Les identifiants des arrêts `v0` et `vf` sont enregistrés dans une liste, permettant ainsi de détailler précisément les points de départ et d'arrivée pour chaque segment du chemin optimal.

4.4 Implémentation du PLNE

Lors de ce projet, nous n'avons pas eu besoin de créer le programme linéaire car il nous était fourni. Nous l'avons simplement implémenté.

Le modèle est formulé comme suit :

Minimiser la fonction objectif :

$$\sum_{(i,j)\in E} x_{i,j}\, p_{i,j} \tag{B.1}$$

sous les contraintes :

$$\sum_{(i,j)\in E} x_{i,j} = 1 \qquad \forall i \in V - \{v_f\} \tag{B.2a}$$

$$\sum_{(i,j)\in E} x_{i,j} = 1 \qquad \forall j \in V - \{v_o\} \tag{B.2b}$$

$$\sum_{(i,j)\in E} x_{i,j} = \sum_{(i,j)\in E} x_{j,i} \qquad \forall i \in V - \{v_o, v_f\} \tag{B.2c}$$

$$\sum_{(i,j)\in E} x_{i,j} - \sum_{(i,j)\in E} x_{j,i} = 1 \qquad i = v_o \tag{B.2d}$$

$$\sum_{(i,j)\in E} x_{i,j} - \sum_{(i,j)\in E} x_{j,i} = -1 \qquad i = v_f \tag{B.2e}$$

$$x_{i,i} = 0 \qquad \forall i \in V \tag{B.2f}$$

$$t_i + p_{i,j} - t_j \leq M \cdot (1 - x_{i,j}) \qquad \forall (i,j) \in E \tag{B.2g}$$

$$a_i \leq t_i \leq b_i \qquad \forall (i,j) \in E \tag{B.2h}$$

$$x_{i,j} \in \{0,1\} \qquad \forall (i,j) \in E \tag{B.2i}$$

Nous avons donc implémenté ce programme linéaire en Python à l'aide de PuLP [1] et de CPLEX [2].

Voici quelques étapes importantes de l'implémentation du PLNE :

- **Les variables**

```
x = pulp.LpVariable.dicts("x", E, cat=pulp.LpBinary)
t = pulp.LpVariable.dicts("t", V, lowBound=0, cat=pulp.LpContinuous)
```

- **Les contraintes**

```python
# Chaque sommet, à l'exception de vf, doit être quitté exactement une fois (B.2a)
for i in V:
    if i != vf:
        prob += pulp.lpSum(x[i, j] for j in V if (i, j) in E) == 1, f"Départ de {i}"

# Chaque sommet, à l'exception de v0, doit être atteint exactement une fois (B.2b)
for j in V:
    if j != v0:
        prob += pulp.lpSum(x[i, j] for i in V if (i, j) in E) == 1, f"Arrivée à {j}"

# Contrainte d'équilibre de flux pour chaque sommet (à l'exception de v0 et vf) (B.2c)
for i in V:
    if i not in {v0, vf}:
        prob += (pulp.lpSum(x[i, j] for j in V if (i, j) in E) -
                 pulp.lpSum(x[j, i] for j in V if (j, i) in E) == 0), f"Flux équilibré à {i}"

# Contrainte de départ unique depuis le sommet de départ v0 (B.2d)
prob += ((pulp.lpSum(x[v0, j] for j in V if (v0, j) in E) - pulp.lpSum(x[j, v0] for j in V if (j, v0) in E) == 1),
         "Départ de v0")

# Contrainte d'arrivée unique au sommet d'arrivée vf (B.2e)
prob += ((pulp.lpSum(x[vf, j] for j in V if (vf, j) in E) - pulp.lpSum(x[j, vf] for j in V if (j, vf) in E) == -1),
         "Arrivée à vf")

# Contrainte de non-utilisation des boucles sur chaque sommet (B.2f)
for i in V:
    prob += x[i, i] == 0, f"Non utilisation de boucle en {i}"

# Contraintes d'ordre temporel entre les sommets pour éviter les incohérences temporelles (B.2g)
for i, j in E:
    prob += t[i] + p[i, j] - t[j] <= M * (1 - x[i, j]), f"Contrainte de temps renforcée entre {i} et {j}"

# Contraintes de fenêtre de temps pour le début au sommet i (B.2h)
for i in V:
    prob += a[i] <= t[i], f"Plage horaire au plus tôt pour {i}"
    prob += t[i] <= b[i], f"Plage horaire au plus tard pour {i}"

```

- **La fonction objectif**

```
prob += pulp.lpSum(x[i, j] * p[i, j] for i, j in E), "Minimiser le temps de
↪  parcours total"
```

Après avoir séparé l'ensemble des stops en clusters et déterminé dans chaque cluster quel sont le premier point (vo) et le dernier (vf) par lesquels la route doit passer, nous pouvons alors appliquer notre PLNE à chacun des clusters. Ce dernier nous permet de déterminer dans quel ordre la route va passer par les arrêts afin d'obtenir la route optimale.

En réalisant cette étape dans chaque sous-groupe, nous obtenons alors la route optimale dans chaque sous-groupe. Grâce au choix des premiers et derniers arrêts de chaque groupe réalisé précédemment, nous obtenons ainsi notre route totale optimale.

5 Présentation et analyse des résultats

5.1 Résultats avec une route

Nous avons ainsi obtenu une route totale qui part de la station, parcourt chaque stop d'après le trajet déterminé préalablement puis revient à la station. Nous avons pu visualiser cette route à l'aide de la bibliothèque *matplotlib* mais également *Networkx*.

Dans cette première visualisation, nous avons affiché les différents stops sur la route (avec leur position). Les stops sont coloriés selon le cluster auquel ils appartiennent et le trajet total est représenté par des flèches entre chaque stop.

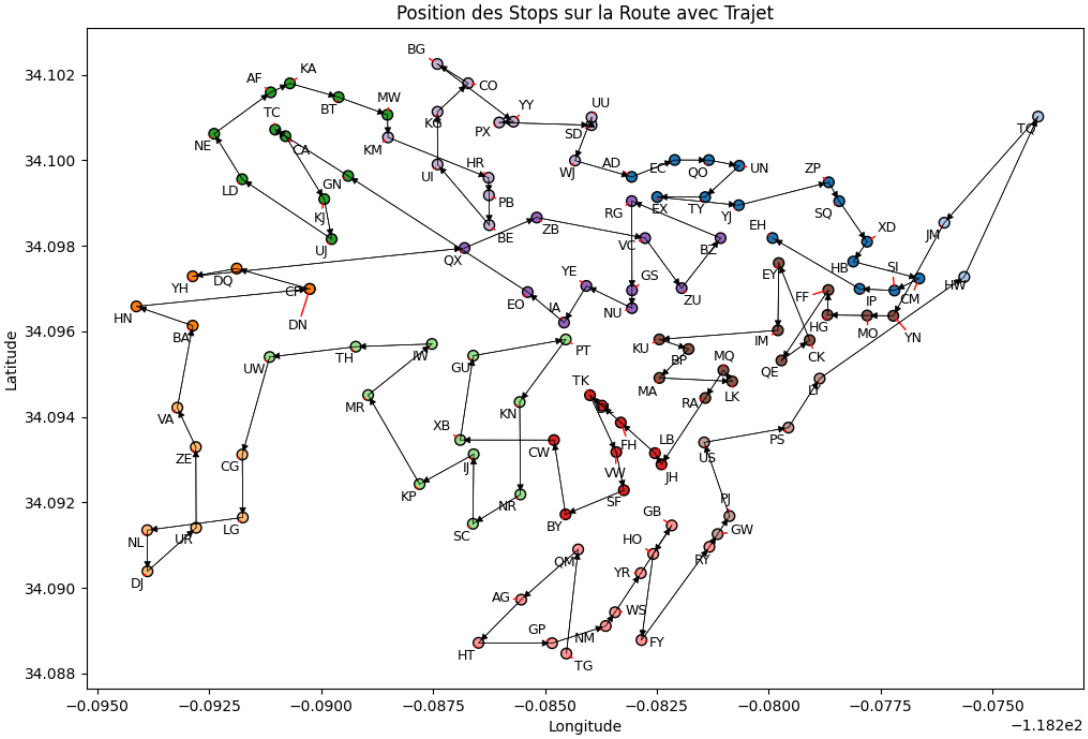


FIGURE 15 – Résultat final de la route 1 avec *matplotlib*

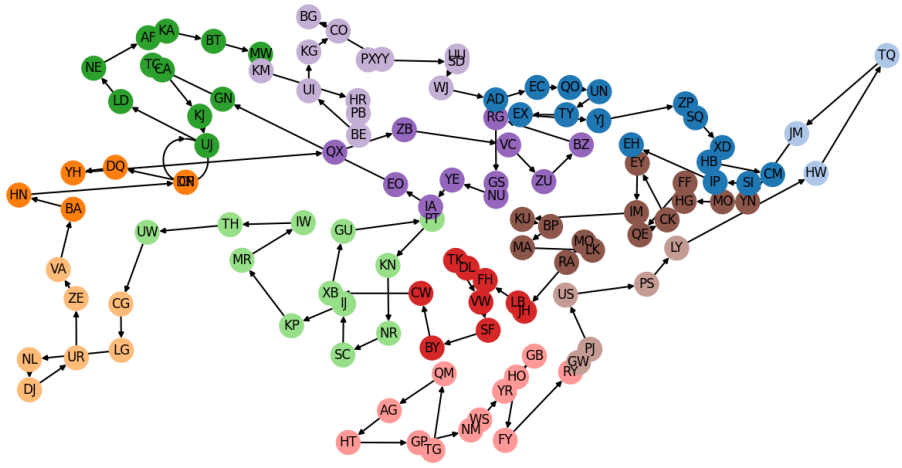


FIGURE 16 – Résultat final de la route 1 avec *Networkx*

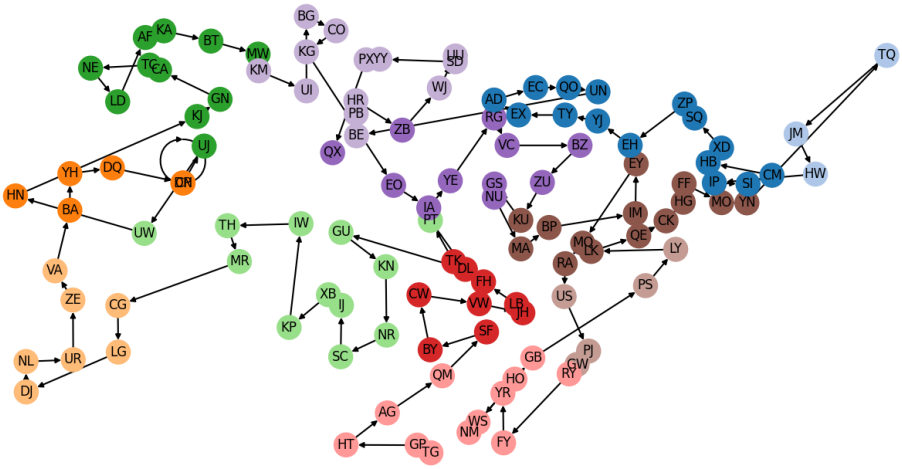


FIGURE 17 – Résultat observé de la route 1 avec *Networkx*

Ces deux figures 16 et 17 ont été réalisées à l'aide de la bibliothèque *Networkx* qui permet

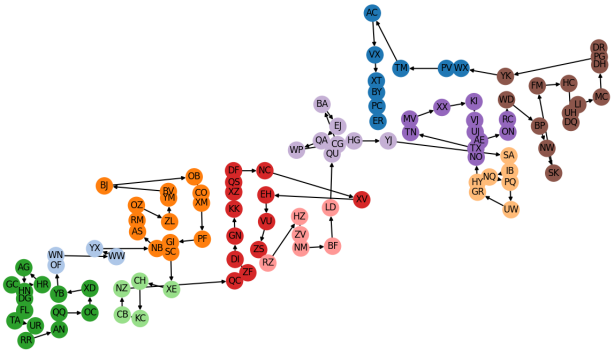
de bien visualiser les graphes.

Nous visualisons correctement sur la figure 16 que les stops sont répartis en clusters de couleurs différentes et que la route passe bien par chaque point, cluster après cluster.

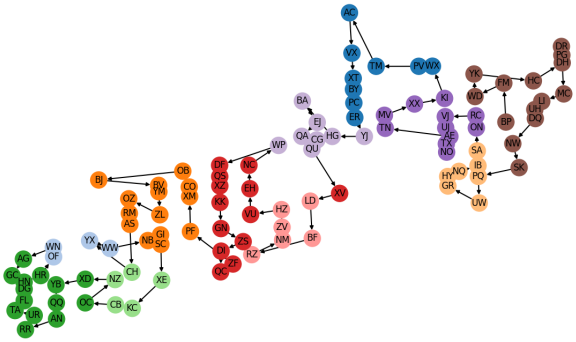
On remarque que le résultat que nous avons obtenu est proche du résultat attendu mais également que le temps de trajet total est plus faible (540.8 secondes de moins soit environ 9 minutes de moins).

5.2 Résultats avec 3 routes

Après avoir affiché notre solution pour la première route, nous avons décidé de visualiser les routes 2 et 3.



(a) Résultat final de la route 2 avec *Networkx*



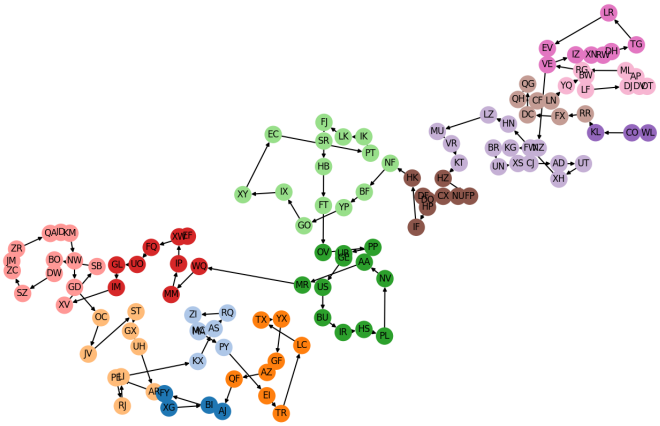
(b) Résultat final de la route 2 avec *Networkx*

FIGURE 18 – Comparaison des résultats pour la route 2

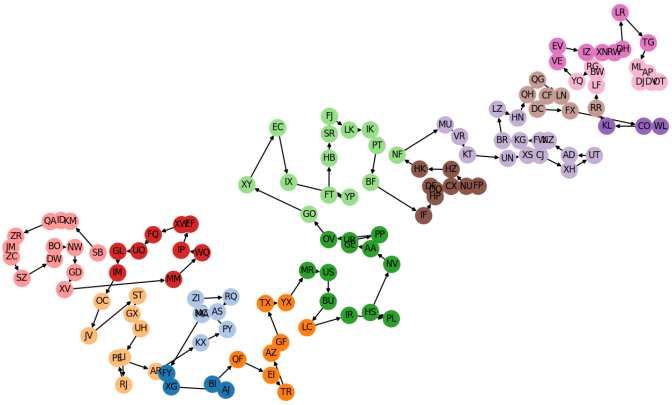
Ainsi, pour les routes 1,2 et 3, nos résultats sont cohérents avec les résultats attendus. En effet, on gagne exactement **9 minutes, 4 min 17, et 1 min 46** par rapport au chemin observé des livreurs. Le tableau 2 regroupe tous les informations relatives aux résultats de nos algorithmes sur ces 3 premières routes.

Route	Nombres de Stops	Heure de Début	Heure de Fin	Temps de Calcul	Temps Gagné
Route 1	119	16 : 02 : 10	18 : 33 : 28	4.73 s.	09 : 00
Route 2	106	15 : 44 : 41	18 : 08 : 21	5.51 s.	04 : 17
Route 3	128	15 : 32 : 04	18 : 57 : 29	6.34 s.	01 : 46

TABLE 2 – Résumé des performances pour les 3 premières routes



(a) Résultat final de la route 3 avec *Networkx*



(b) Résultat observé de la route 3 avec *Networkx*

FIGURE 19 – Comparaison des résultats pour la route 2

5.3 Résultats avec 100 routes

Nous n'avons malheureusement pas réussi à importer le dossier contenant toutes les routes (plusieurs gigas), mais il nous semblait important d'avoir un grand échantillon de routes pour pouvoir analyser des résultats significatifs.

Nous avons donc décidé de faire tourner notre code sur **un ensemble de 100 routes**.

Voici les résultats obtenus :

```
Moyenne des temps par cluster: 0.01 secondes
Moyenne des temps pour le chemin optimal: 0.00 secondes
Moyenne des temps pour le trajet du point initial au point final: 0.00 secondes
Moyenne des temps totaux pour TSP avec fenêtres de temps: 7.08 secondes
Moyenne des temps de calcul total: 7.09 secondes
Le temps moyen gagné est : 00:10:21

Process finished with exit code 0
```

FIGURE 20 – Temps de calculs nécessaires pour les 100 premières routes

On peut résumer ces informations dans un tableau :

TABLE 3 – Résumé des Temps Moyens de Calcul

Description	Temps Moyen (secondes)
Moyenne des temps pour tous les cluster de chaque route	0.01
Moyenne des temps pour trouver le chemin optimal	0.00
Moyenne des temps pour trouver tous les `v0` et `vf`	0.00
Moyenne des temps totaux pour les TSP avec fenêtres de temps	7.08
Moyenne des temps de calcul	7.09
Temps moyen gagné	00 : 10 : 21

Ainsi, en moyenne, sur des tranches de travail allant de 1h30 à 3h30 :

Notre travail pourrait permettre de faire gagner
10 minutes de travail au livreur durant son travail.

6 Limites et difficultés rencontrées

6.1 Limites

Dans ce projet, nous avons subdivisé nos données en clusters en fixant un nombre maximal de stops à ne pas dépasser dans chaque cluster. Nous étions obligé de faire cela sous peine de devoir attendre une éternité pour calculer la solution optimale pour une unique route. Notre heuristique de clustering a été choisie pour sa **simplicité** et son **efficacité**, bien qu'il en existe sûrement beaucoup d'autres. Nous ne pouvons pas garantir que notre idée était la meilleur.

De plus, il est important de noter que certaines données importantes n'ont pas été exploitées dans notre approche. Par exemple, les informations sur les capacités des camions des livreurs ainsi que la qualité de la route n'ont pas été prises en compte (`executor_capacity_cm3` et `route_score` dans la figure 2). De même, la taille des colis dans le camion n'a pas été intégrée dans notre analyse (`dimensions` dans la figure 3). L'inclusion de ces informations aurait pu potentiellement améliorer l'analyse de nos résultats et permettre d'expliquer davantage le temps perdu par les livreurs dans certaines routes.

6.2 Difficultés rencontrées

Lors de ce projet, l'une des principales difficultés que nous avons rencontrée était le traitement d'un très grand jeu de données. Même en nous limitant à une seule route pour nos premiers tests, la liste des arrêts était tout de même conséquente. Cela a posé un défi initial, notamment pour la visualisation de nos résultats et la vérification de nos tests.

De plus, une complication supplémentaire est survenue au début du projet : le fichier permettant de lire les données n'était pas fonctionnel. Cette situation a ralenti notre progression, car nous ne pouvions pas tester les fonctions développées par les autres membres de l'équipe pendant que l'un d'entre nous s'efforçait de résoudre ce problème technique.

Enfin, nous avons également dû faire face à la question du traitement des valeurs manquantes (notamment dans les données `start_time_utc` et `end_time_utc`). Dans certains cas, il n'était pas si évident de déterminer quelle valeur utiliser pour remplacer les données manquantes, ce qui a nécessité une réflexion approfondie et des choix judicieux pour garantir la qualité de nos analyses et résultats.

D'une manière plus générale, l'implémentation de tous les fichiers `clusteringData.py` et `fonction_plne.py` ont été difficiles, et ont nécessité de longs moments de débogage.

7 Conclusion

En conclusion, ce projet de recherche nous a permis d'explorer et de mettre en œuvre des approches innovantes pour résoudre cet exemple de problème de séquençage d'itinéraires, en nous appuyant sur un article de référence et en travaillant sur des données réelles fournies par Amazon. Le projet était très stimulant, autant dans la réflexion qu'il a fallu mettre en oeuvre que dans l'implémentation concrète des fonctions et algorithme.

Nous avons suivi une démarche structurée en plusieurs étapes clés : la mise en forme des données brutes, le clustering des arrêts, la détermination de la route finale entre les différents groupes, la sélection des premiers et derniers arrêts de chaque sous-groupe, et l'application du PLNE dans chaque sous-groupe pour obtenir l'itinéraire optimal.

Grâce à cette approche, nous avons pu obtenir des résultats intéressants et prometteurs, puisqu'ils sont (au moins) meilleurs que ceux observés concrètement avec les temps de trajets des livreurs d'Amazon. Cependant, nous avons également rencontré des défis et des limites dans notre travail, notamment en ce qui concerne la taille des données.

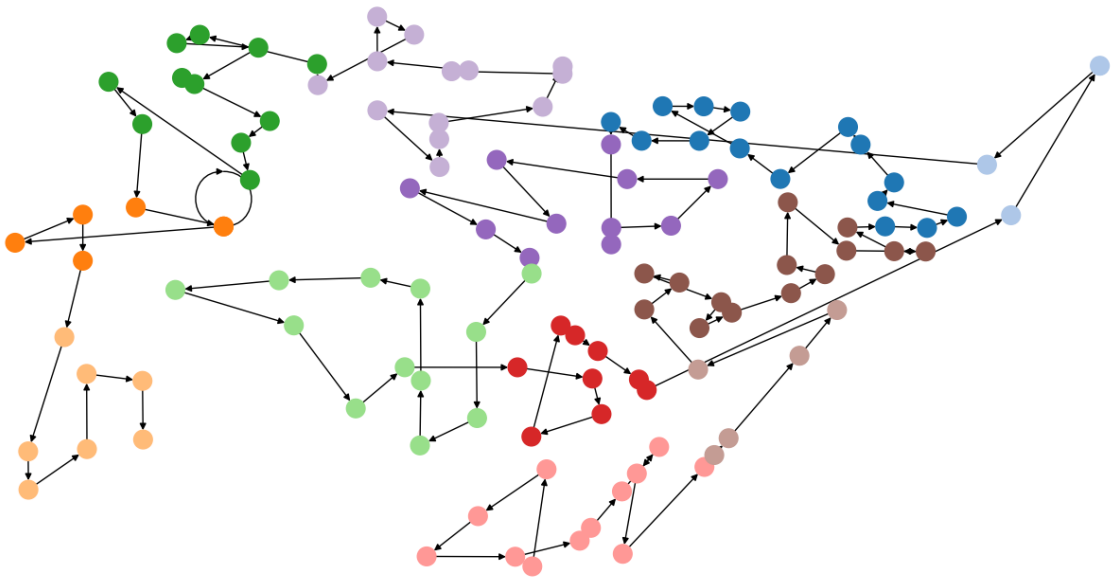


FIGURE 21 – Un dernier rendu visuel pour conclure - Première route

Références

[1] *Bibliothèque PuLP pour Python*. https://pypi.org/project/PuLP/. Jan. 2024.

[2] *IBM ILOG CPLEX Optimisation Studio*. https://www.ibm.com/fr-fr/products/ilog-cplex-optimization-studio. Déc. 2022.

[3] Romain Loirs, Rathea Uth, Massinissa Merabet, Hông-Lan Botterman et Nicolas Brunel. « Solving Vehicle Routing Problem with linear programming optimized by Machine Learning ». In : *Quantmetry* (2022).

[4] Massinissa Merabet. *Amazon Last-Mile Routing Research Project*. https://acrobat.adobe.com/id/urn:aaid:sc:eu:876c3c1e-8685-458e-a403-a70f29a5c425. Accédé le : 19/04/2024.

[5] *MIT Routing Challenge*. https://routingchallenge.mit.edu. 2021.

[6] PlantText. *Site officiel*. https://www.planttext.com. Sept. 2023.

[7] Quantmetry. *Solution au problème de routage des véhicules par programmation linéaire optimisée par l'apprentissage automatique*. https://www.quantmetry.com/blog/solution-routage-programmation-lineaire-optimisee-apprentissage-automatique/. Jan. 2022.