Optimisation de la collecte des déchets

Sommaire

I - Introduction : la collecte des déchets

II - Modélisation du problème
de tournées de véhicules

III - Résolution par deux méthodes
différentes

IV - Conclusion

V - Annexe

I - Introduction

I.A La Smart City

Définition

- Améliore la qualité des services urbains grâce à la technologie
- Réduit les coûts financiers et la pollution

Exemples

➜ Compteurs électriques, d'eau

➜ Poubelles connectées

Initiatives :

Contact de l'entreprise *Tekin* spécialisée dans les objets connectés sur Tours :
échange en visioconférence avec le directeur général Raphaël AUTALE

I.B La collecte des déchets

Service indispensable à une ville

- ➜ Augmentation des déchets

Moyens nécessaires conséquents

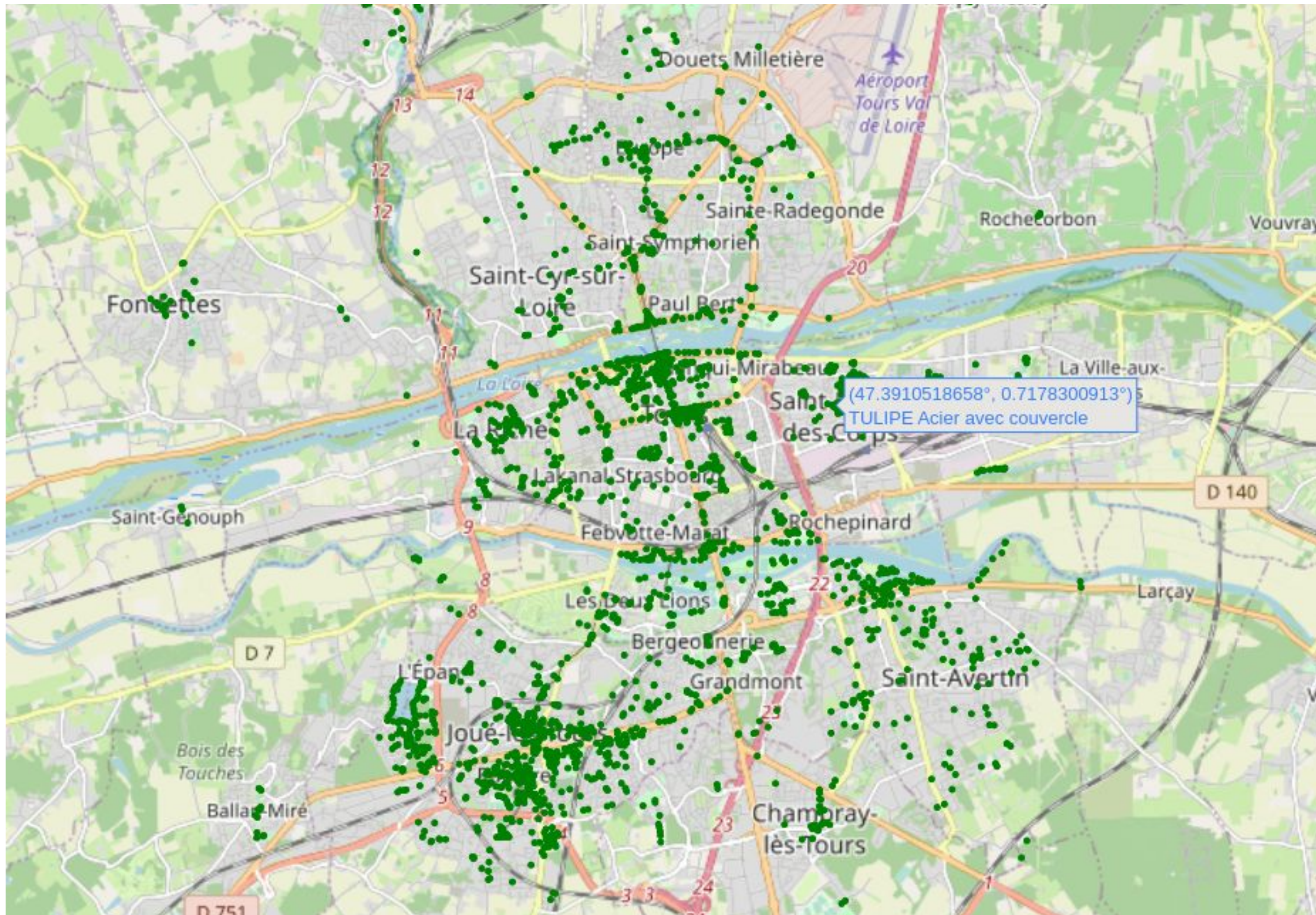
- ➜ Camion poubelle de plusieurs tonnes
- ➜ Installation des poubelles
- ➜ Entretien, carburant

Peut-on l'optimiser ?

- ➜ Trajets plus judicieux, plus courts
- ➜ Données de remplissage en temps réel

II - Modélisation

II.A Données Open Data



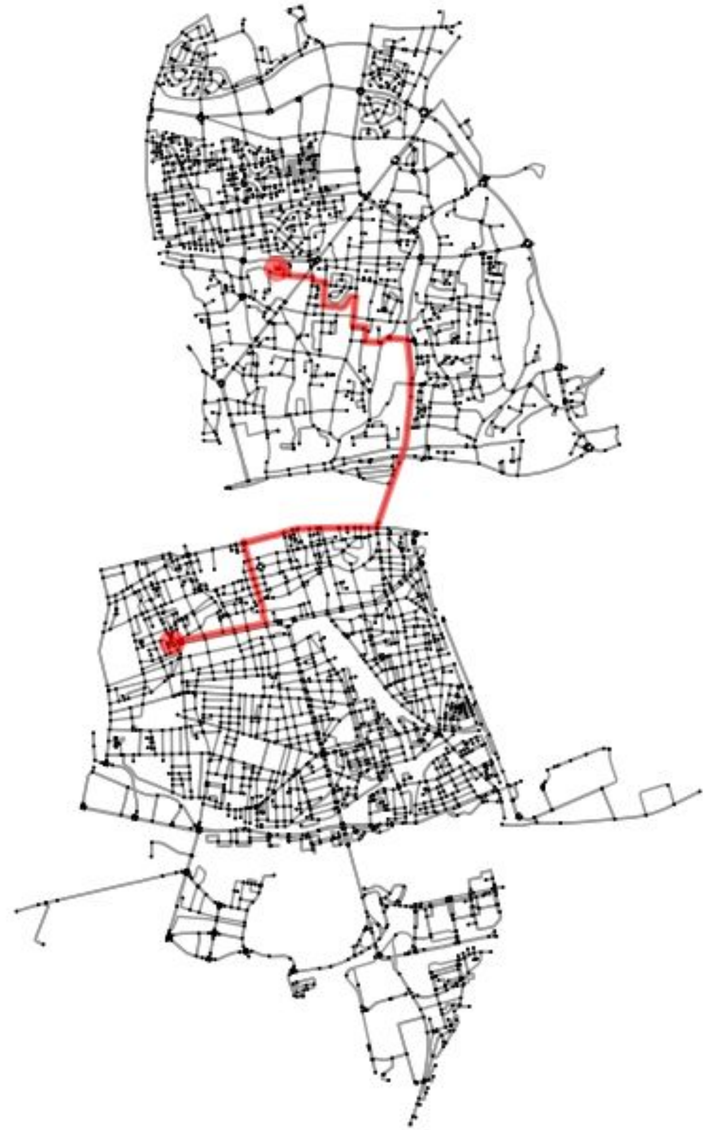
Emplacement des corbeilles publiques à Tours
(Source des données : *data.tours-metropole.fr*)

II.B Représentation avec des graphes

La ville de Tours affichée en tant que graphe :

- Les sommets sont les intersections
- Les arêtes sont les rues

En rouge : un plus court chemin entre deux sommets de ce graphe.



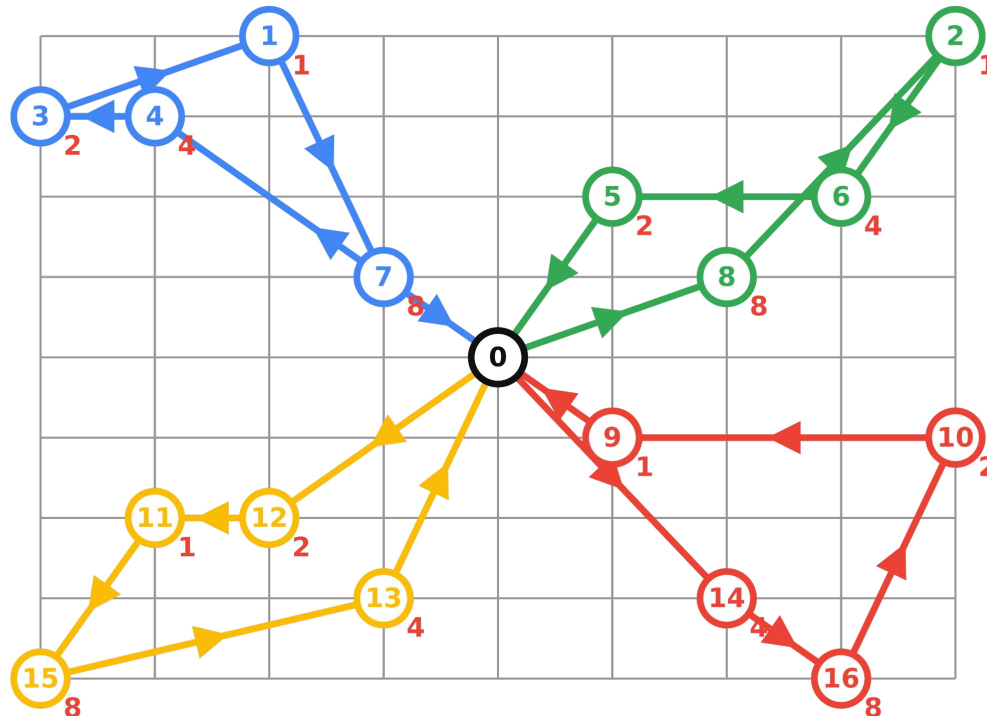
Hypothèse :

➜ On considère qu'il y a une poubelle à chaque intersection

II.C Problème de tournées de véhicules

Classification

- ➔ Abrégé **CVRP** (Capacited Vehicle Routing Problem)
- ➔ Problème d'optimisation combinatoire
- ➔ Généralisation du problème du voyageur de commerce à une flotte de plusieurs véhicules ayant une capacité limitée



Source :
Google OR-TOOLS

II.D Stratégies et choix

Un modèle simplifié :

- Minimiser uniquement la distance parcourue
- Ignorer les contraintes de temps

Différentes stratégies :

- Réduire le nombre de nœuds :
  ➔ création de clusters avec l'algorithme des k-moyennes
- Une première approche intuitive :
  ➔ via un algorithme glouton
- Une deuxième approche plus évoluée :
  ➔ en utilisant le clustering

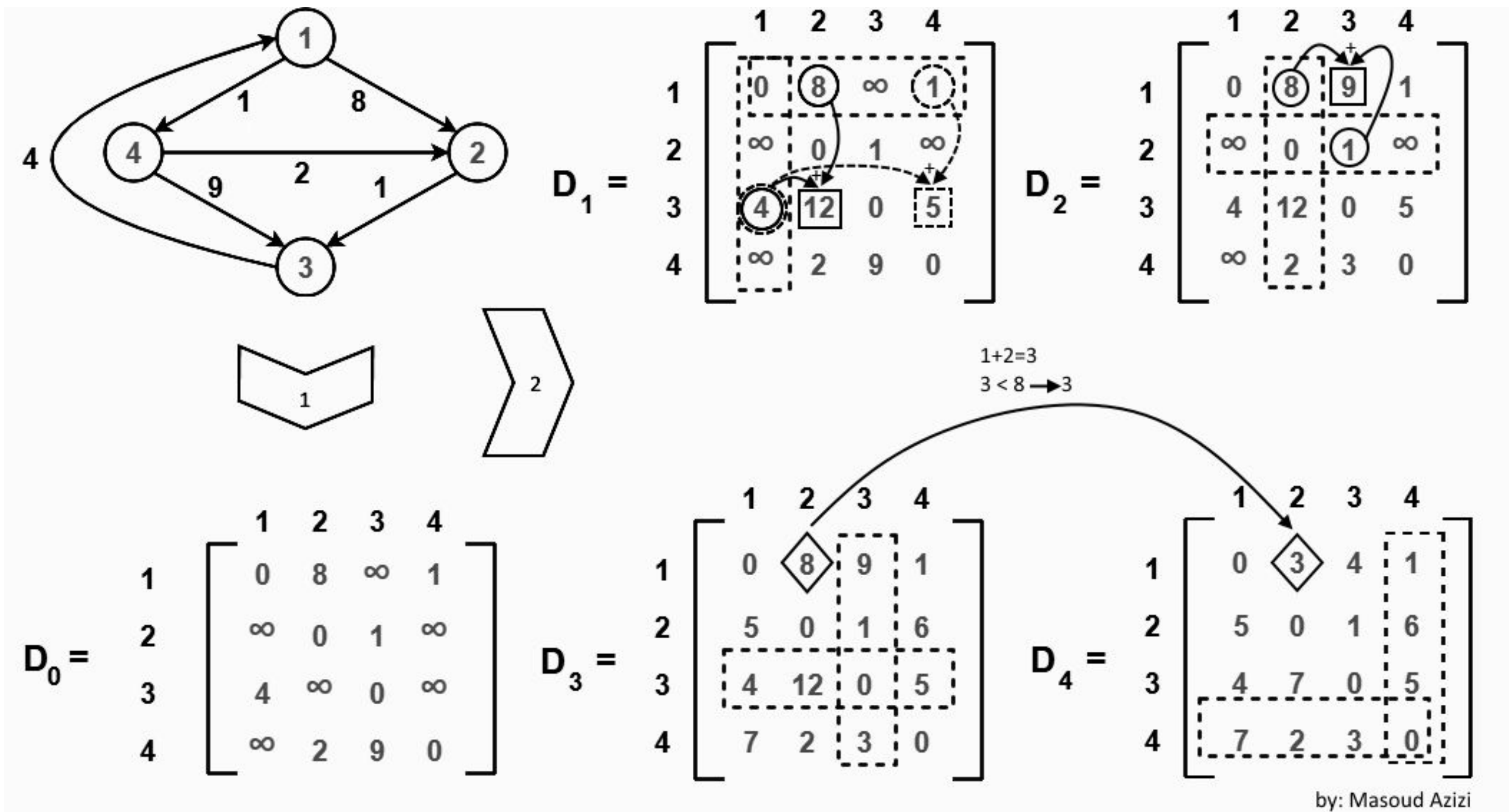
III - Simulation

III.A Méthode gloutonne (1)

Préparation :

➔ On récupère les données de la ville (topographie, distances, intersections...)

➔ On calcule la matrice de distance avec l'algorithme de Floyd-Warshall :



Source : *www.wikidata.org*

III.A Méthode gloutonne (2)

Algorithme glouton pour résoudre CVRP :

➔ On choisit un dépôt de départ.

➔ On boucle tant qu'il reste des poubelles non visitées.

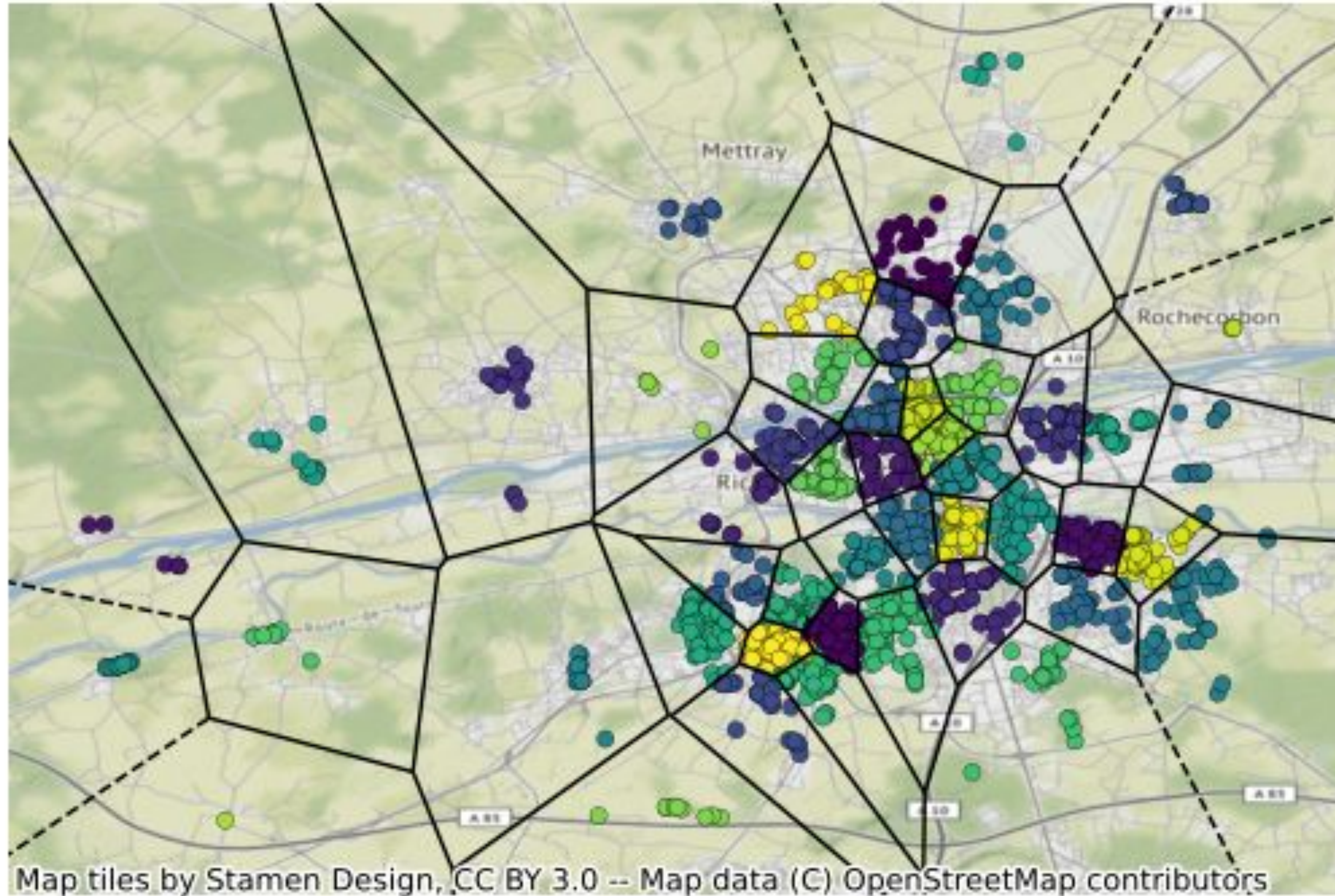
➔ À chaque itération, on cherche la poubelle la plus proche de la dernière visitée.

➔ On vérifie si elle respecte les contraintes de capacité du véhicule. Si oui, on l'ajoute à la route et on met à jour la capacité du véhicule. Sinon, on termine la route actuelle en ajoutant le dépôt et on démarre une nouvelle route.

➔ On renvoie la liste des routes.

Complexité **quadratique** en le nombre de poubelles.

III.B Algorithme des k-moyennes (1)



Convergence avec $k = \lceil \frac{N}{50} \rceil$ avec N = nombre de points

III.B Algorithme des k-moyennes (2)

1. Choisir **k points** appelés *centroïdes***,** qui représenteront les centres initiaux

2. Chaque point est attribué au centroïde le plus proche en termes de distance euclidienne. Cela crée k clusters initiaux.

3. Calculer l'**isobarycentre** de chaque cluster : ce sont les nouveaux centroïdes.

4. Les points sont réattribués au cluster associé au centroïde le plus proche.

5. Les étapes 3 et 4 sont répétées jusqu'à ce qu'il y ait **convergence** : il n'y a plus de changement dans l'attribution des points.

III.C Méthode gloutonne avec clustering

Méthode “cluster first - route second” :

➔ Attribuer un ensemble de clusters à chaque camion poubelles de manière gloutonne selon l’algorithme précédent en considérant chaque cluster comme un nœud :
- Il est localisé par son centroïde
- La capacité de chaque cluster est égale à la somme des capacités des poubelles qu’il contient.

➔ Résoudre le problème du voyageur de commerce pour chaque cluster.

III.D Algorithme de Christofides

Problème du voyageur de commerce

➔ Trouver le cycle hamiltonien de poids minimal

Algorithme

➔ Constitue une 3/2 approximation du problème du voyageur de commerce

➔ Complexité temporelle cubique en la taille du graphe

➔ Implémenté dans le module Networkx

III.E Résultats : algorithme glouton

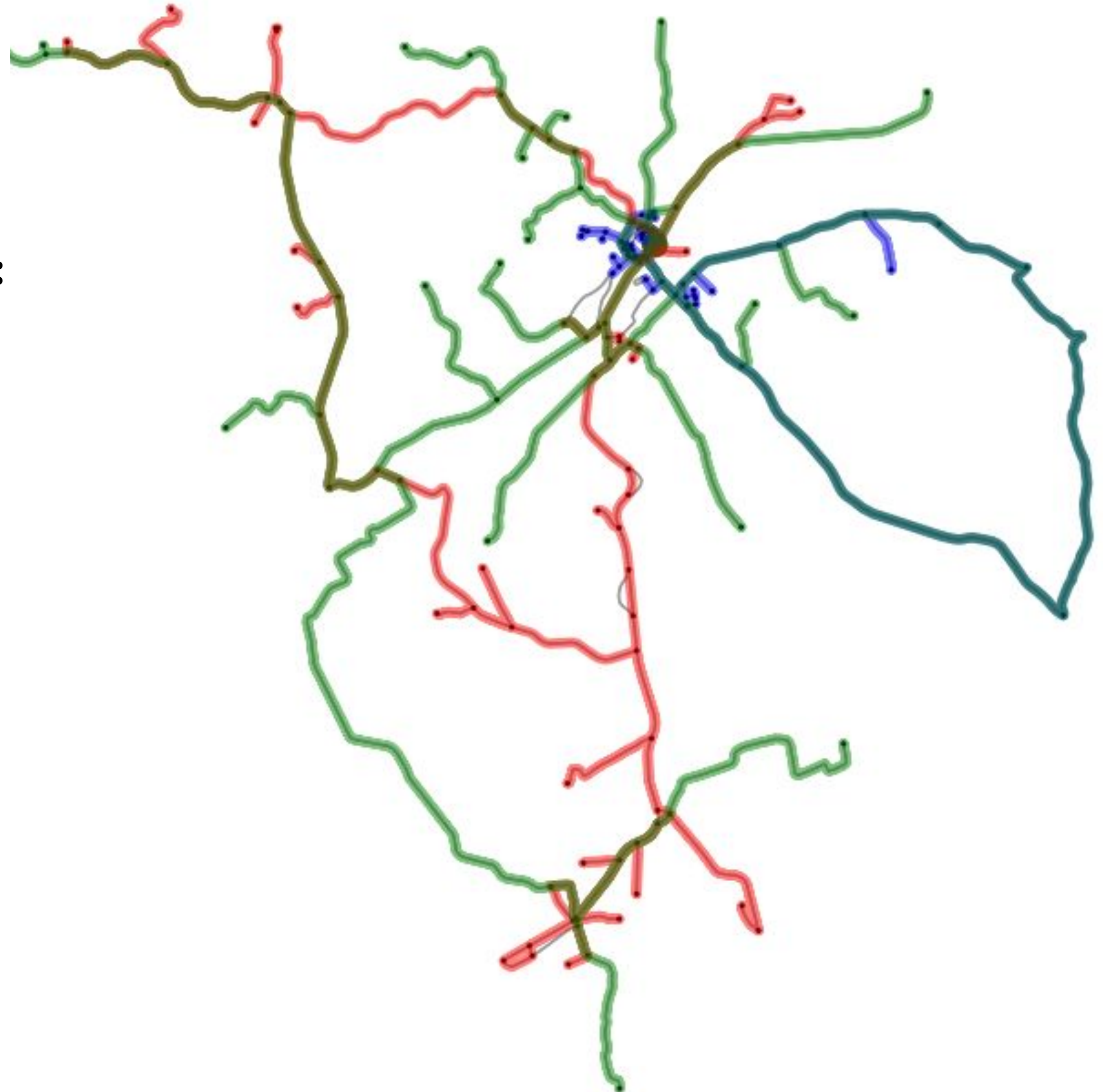
Ville d'Agonac (Dordogne) :

Tour du véhicule 1:
Distance totale: 10539.2

Tour du véhicule 2:
Distance totale: 41623.3

Tour du véhicule 3:
Distance totale: 73657.1

Total : 125819.6 m



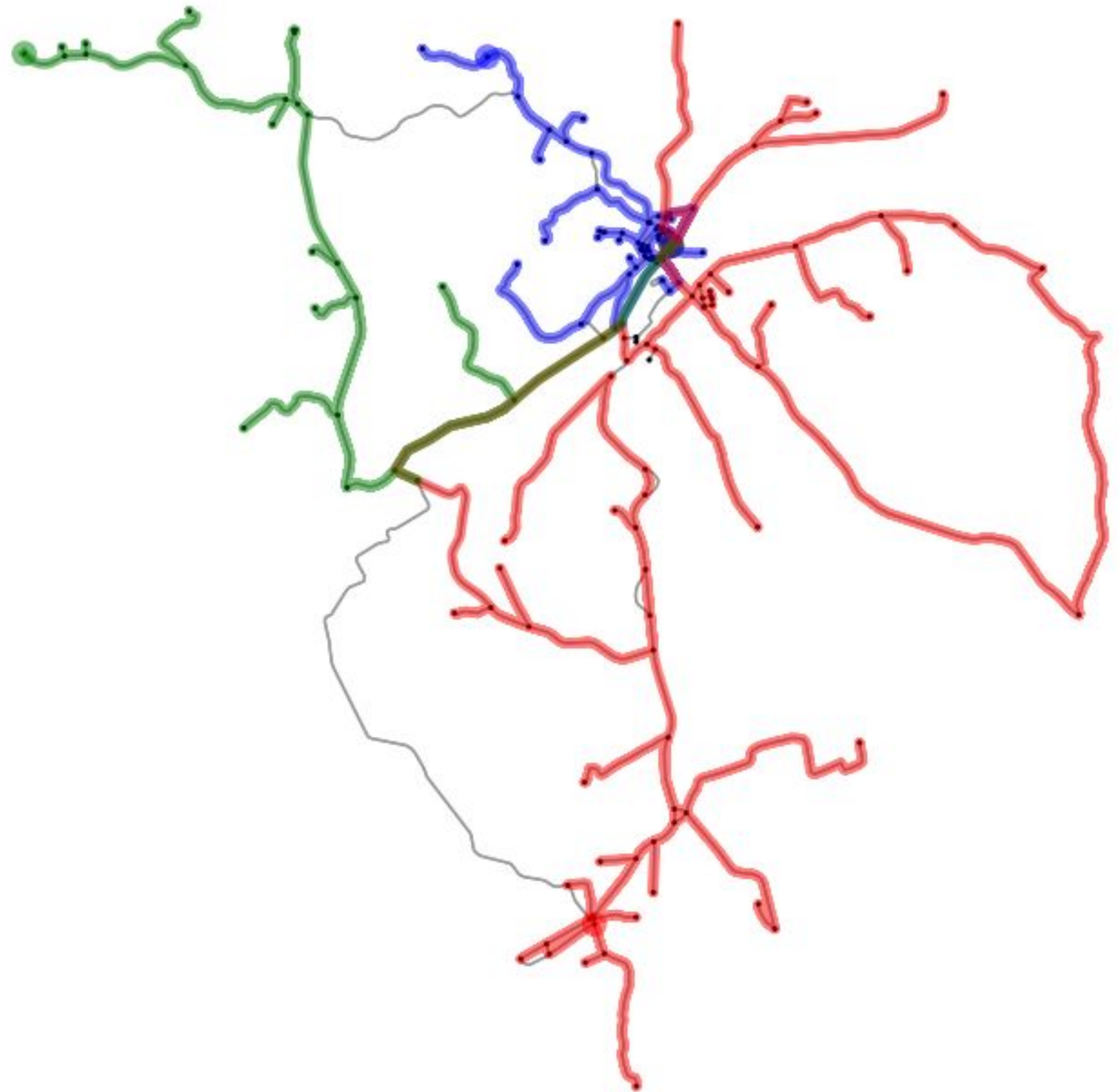
III.D Résultats : “cluster first - route second”

<u>Tour du véhicule 1:</u>
Distance totale: 18505.8

<u>Tour du véhicule 2:</u>
Distance totale: 72002.7

<u>Tour du véhicule 3:</u>
Distance totale: 22130.5

Total : 112639 m



IV - Conclusion

IV Conclusion

- ➔ Algorithme glouton en $O(n^3)$
- ➔ Algorithme “cluster first - route second” en $O(n^4)$

Optimisations possibles :

- ➔ Calculer de manière concurrente le chemin de chaque camion poubelle dans le cluster

Autre approche : Utiliser un algorithme génétique

V - Annexe

II.A Données Open Data

simulation/utils/map_plotly.py

```python
import geopandas
import plotly.graph_objects as go

df = geopandas.read_file("data/corbeilles-tours-metropole.geojson")
# Setting default value of capacity if not set
df.loc[df["contenance"].isnull(), "contenance"] = 10

fig = go.Figure(
    go.Scattermapbox(
        lon=df.geometry.x,
        lat=df.geometry.y,
        text=df["type"],
        hovertext=["type", "commune"],
        marker=dict(color='green', size=6),
        hoverlabel=dict(font=dict(color='blue'), bgcolor='white',
bordercolor='blue'),
    ),
    go.Layout(
        title="Corbeilles Tours Métropole",
        autosize=True,
        mapbox=dict(style="open-street-map", center=dict(lat=47.40, lon=0.70),
zoom=10),
    ),
)

if __name__ == "__main__":
    fig.show()

```

II.B Représentation avec des graphes

simulation/utils/graph_osmnx.py

```
1  import osmnx as ox
2  import networkx as nx
3  import numpy as np
4
5  COLORS = ["blue", "red", "green", "yellow", "brown", "purple"]
6
7  def plot_graph(city):
8      G = ox.graph_from_place(city, network_type='drive')
9
10     # Plot the shortest path between two random nodes
11     origin, destination = np.random.choice(G.nodes, 2)
12     route =  nx.shortest_path(G, origin, destination)
13
14     ox.plot_graph_route(G, route, route_color='red', route_linewidth=3,
15                         node_size=2, bgcolor='white', node_color='black')
16
17
18 def plot_graph_routes(graph, tours, depot):
19     """
20     Rebuild routes from nodes indices and plot them on the graph
21     """
22     routes = [[depot] for _ in range(len(tours))]
23
24     for i, tour in enumerate(tours):
25         for j in range(len(tour)-1):
26             intermediates_nodes = nx.shortest_path(graph, tour[j], tour[j+1])
27             routes[i].extend(intermediates_nodes[1:])
28
29     ox.plot_graph_routes(graph, routes, route_colors=COLORS[:len(routes)],
30                          route_linewidth=1.5, node_size=5,
31                          bgcolor='white', node_color='black', route_alpha=0.5)
```

III.A Algorithme de Floyd-Warshall

simulation/utils/floyd_warshall.py

```
1  import numpy as np
2  import networkx as nx
3
4
5  def floyd_warshall(graph):
6      """
7      Return the distance matrix of the graph
8      """
9      num_nodes = graph.number_of_nodes()
10     distance_matrix = nx.to_numpy_array(graph, weight='length')
11     # Replace all zeros with infinite distances
12     distance_matrix[distance_matrix == 0] = np.inf
13
14     for k in range(num_nodes):
15         for i in range(num_nodes):
16             for j in range(num_nodes):
17                 distance_matrix[i, j] = min(distance_matrix[i, j],
18                                             distance_matrix[i, k] +
distance_matrix[k, j])
19     return distance_matrix
20
```

III.A Méthode gloutonne (1)

simulation/cvrp_greedy.py

```python
import numpy as np
import networkx as nx
import osmnx as ox

from utils.floyd_warshall import floyd_warshall
from utils.graph_osmnx import plot_graph_routes


def cvrp_greedy(distance_matrix, demands, capacities):
    """
    Solve the CVRP problem with a greedy algorithm
    """
    num_clients = len(demands)
    num_vehicles = len(capacities)
    depot = 0  # Assuming depot is at index 0
    vehicle_id = 0
    unvisited_clients = list(range(1, num_clients))  # List of unvisited clients
    routes = []  # List of routes

    while vehicle_id < num_vehicles and unvisited_clients:
        current_route = [depot]  # Start a new route from the depot
        current_load = 0

        while current_load <= capacities[vehicle_id] and unvisited_clients:
            last_client = current_route[-1]
            closest_client = min(unvisited_clients, key=lambda client:
distance_matrix[last_client][client])

            if current_load + demands[closest_client] <= capacities[vehicle_id]:
                current_route.append(closest_client)
                current_load += demands[closest_client]
                unvisited_clients.remove(closest_client)
```

III.A Méthode gloutonne (2)

```
32                  else:
33                      current_route.append(depot)  # Complete the current route at the
depot
34                      routes.append(current_route)
35                      break
36          # Skip to the next vehicle
37          vehicle_id += 1
38
39      if unvisited_clients:
40          raise Exception("Pas de solution trouvée. Augmentez le nombre des
véhicules.")
41      else:
42          current_route.append(depot)
43          routes.append(current_route)
44
45      return routes
46
47
48  graph = ox.graph_from_place('Agonac, France', network_type='drive')
49  distance_matrix = floyd_warshall(graph)
50
51  # Generate random demands and capacities
52  num_clients = len(distance_matrix)
53  demands = np.random.randint(1, 10, size=num_clients)
54  demands[0] = 0  # Depot demand is zero
55  NUM_VEHICLES = 2
56  INCREASE_FACTOR = 1.1
57  capacities = np.ones(NUM_VEHICLES) * sum(demands) * INCREASE_FACTOR / NUM_VEHICLES
58
59  # Run the algorithm
60  tours = cvrp_greedy(distance_matrix, demands, capacities)
61
```

III.B Clustering

simulation/k_means_clustering.py

```
import sys

import numpy as np
import geopandas
import contextily as cx
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from scipy.spatial import Voronoi, voronoi_plot_2d

# Pandas dataframe from GeoJSON file
df = geopandas.read_file("data/corbeilles-tours-metropole.geojson")

# Array of bins coordinates
X = np.column_stack((df.geometry.x, df.geometry.y))
# 50 bins per cluster seems a good mean
k = len(X) // 50 + 1

# https://scikit-learn.org/stable/modules/clustering.html?highlight=clustering#k-means
partitions = KMeans(n_clusters=k, n_init='auto').fit(X)

# Plot the bins with a cmap from cluster id
ax = df.plot(edgecolor="black", c=partitions.labels_,
             markersize=20, linewidth=0.2)

# Add the voronoi diagram https://fr.wikipedia.org/wiki/Diagramme_de_Vorono%C3%AF
voronoi = Voronoi(partitions.cluster_centers_)
voronoi_plot_2d(voronoi, ax, show_points=False, show_vertices=False)

cx.add_basemap(ax, crs=df.crs)
ax.set_title("Corbeilles Tours Metropole")
ax.set_axis_off()

plt.show()
```

III.B Algorithme des k-moyennes

simulation/utils/k_means.py

```
1  import numpy as np
2
3
4  def distance(point1, point2):
5      # Calcul de la distance euclidienne entre deux points
6      return np.sqrt(np.sum((point1 - point2) ** 2))
7
8
9  def kmeans(X, k, max_iters=100):
10     # Initialisation aléatoire des centroides
11     centroids = X[np.random.choice(range(X.shape[0]), size=k, replace=False)]
12
13     for _ in range(max_iters):
14         # Étape d'affectation des clusters
15         distances = np.array([np.array([distance(X[i], centroids[j]) for j in
   range(k)]) for i in range(X.shape[0])])
16         cluster_labels = np.argmin(distances, axis=1)
17
18         # Mise à jour des centroides
19         new_centroids = np.array([X[cluster_labels == j].mean(axis=0) for j in
   range(k)])
20
21         # Vérification de la convergence
22         if np.all(centroids == new_centroids):
23             break
24
25         centroids = new_centroids
26
27     return centroids, cluster_labels
28
```

III.C Méthode “cluster first - route second” (1)

simulation/cvrp_clustering.py

```python
import numpy as np
import osmnx as ox
import networkx as nx
from networkx.algorithms.approximation.traveling_salesman import christofides

from utils.floyd_warshall import floyd_warshall
from utils.graph_osmnx import plot_graph_routes
from utils.k_means import kmeans, distance
from cvrp_greedy import cvrp_greedy

NUM_VEHICLES = 3
INCREASE_FACTOR = 1.1


def tsp_christofides(graph, subset):
    """
    Résout le problème du voyageur de commerce (TSP) avec l'algorithme de Christofides
    sur un sous ensemble de points du graphe
    """
    complete_subgraph = nx.complete_graph(graph.subgraph(subset)).to_undirected()
    return christofides(complete_subgraph)


graph = ox.graph_from_place('Agonac, France', network_type='drive')
graph_nodes = np.array(graph.nodes())
distance_matrix = floyd_warshall(graph)

# Génère des demandes et des capacités aléatoires
num_clients = len(distance_matrix)
demands = np.random.randint(1, 10, size=num_clients)
demands[0] = 0  # depot
capacities = np.ones(NUM_VEHICLES) * sum(demands) * INCREASE_FACTOR / NUM_VEHICLES

# Obtiens la position des clients
points = np.array([[graph.nodes[node]['x'], graph.nodes[node]['y']] for node in graph.nodes])
k = num_clients // 10 # Nombre de clusters souhaités
centroids, cluster_labels = kmeans(points, k)
```

III.C Méthode “cluster first - route second” (2)

```
39 # Créer les nouveaux clients pour chaque cluster
40 cluster_demands = [0] * k
41 cluster_clients = [[] for _ in range(k)]
42 for j in range(k):
43     for i in range(len(cluster_labels)):
44         if cluster_labels[i] != j:
45             continue
46         cluster_demands[j] += demands[i]
47         cluster_clients[j].append(i)
48
49 # distance matrix of centroids points
50 cluster_distance_matrix = np.array(
51     [[distance(centroids[i], centroids[j]) for j in range(k)] for i in range(k)]
52 )
53 # Appliquer l'algorithme CVRP greedy
54 cluster_routes = cvrp_greedy(cluster_distance_matrix, cluster_demands, capacities)
```

```
56 # Compléter les routes pour chaque cluster avec l'algorithme de Christofides
57 routes = [[graph_nodes[0]] for _ in range(len(cluster_routes))]
58 for i, route in enumerate(cluster_routes):
59     for cluster in route:
60         if cluster == 0:
61             continue
62         clients = cluster_clients[cluster]
63         cluster_route = tsp_christofides(graph, graph_nodes[clients])
64         routes[i].extend(cluster_route[1:])
65
66 # Affiche les routes et les résultats
67 plot_graph_routes(graph, routes, graph_nodes[0])
```