AI09 - TEAM ORIENTEERING PROBLEM, A TSP VARIANT

December 8, 2022

- *Anne-Soline GUILBERT–LY*
- *Alexandre TAESCH*

1 Introduction

De nos jours, la gestion de la chaîne d'approvisionnement est reconnue comme un des piliers de la compétitivité et du succès des entreprises de fabrication et de distribution. Les problèmes d'acheminement des marchandises des dépôts aux consommateurs ou entre différents sites logistiques sont très importants. L'élaboration de plans adéquats peut permettre des économies importantes pour de nombreux systèmes de distribution. Par conséquent, le problème de routage des véhicules (VRP) est un problème bien connu étudié en recherche opérationnelle. Il s'agit de trouver la configuration optimale des itinéraires de livraison d'un ou de plusieurs dépôts vers un certain nombre de clients en utilisant une flotte de véhicules capacitaires, tout en satisfaisant certaines contraintes. La solution d'un VRP est un ensemble d'itinéraires à coût minimal qui répondent aux exigences des clients. Dans la version classique du problème, un seul dépôt est considéré avec une flotte illimitée de véhicules où chaque véhicule effectue exactement un circuit. Plusieurs contraintes opérationnelles peuvent être considérées dans des applications plus pratiques du VRP.

Ci-dessous quelques variantes existantes. Nous nous intéresserons au problème TOP - Team Orienteering Problem

Variants	Entire Name
ACVRP	Asymmetric Capacitated Vehicle Routing Problem
CARP-TP	Capacitated Arc Routing Problem with Turn Penalties
ConVRP	Consistent Vehicle Routing Problem
COP	Clustered Orienteering Proglem
CTTRP	Capacitated Truck-and-Trailer Routing Problem
CVRP	Capacitated Vehicle Routing Problem
CVRPTWUPI	Capacitated Vehicle Routing Problem with Time Windows, Unmatched pickups and deliveries, Priority tasks, and Inventory restrictions
CVTSP	Carrier-Vehicle Traveling Salesman Problem
FSM-DARP-RC	fleet size and mix dial-a-ride problem with reconfigurable vehicle capacity

Variants	Entire Name
GenConVRP	Generalized Consistent Vehicle Routing Problem
MDVRP	Multiple Depot Vehicle Routing Problem
MOSSP	Multi-objective shortest path problem
POP	Probabilistic Orienteering Problem
PRP	Pollution Routing Problem
PTP	Prisoner Transportation Problem
PVRP	Periodic Vehicle Routing Problem
RWCP	Rich Waste Collection Problem
SBVRP	Swap-body Vehicle Routing Problem
SDVRP (Site-Dependent)	Site-Dependent Vehicle Routing Problem
SDVRPTW (Site-Dependent)	Site-Dependent Vehicle Routing Problem with Time Windows
STTRPSD	Single Truck and Trailer Routing Problem with Satellite Depots
TOP	**Team Orienteering Problem**
TRSP	Technician Routing and Scheduling Problem
TSPPDDL	Travelling Salesman Problem with Pickup, Delivery and Draught Limits
TWAVRP	Time Window Assignment Vehicle Routing Problem
VRPDO	Vehicle Routing Problem with Delivery Options
VRPSD	Vehicle routing problem with stochastic demands
VRPTW	Vehicle Routing Problem with Time Windows

2 I- TOP - Team Orienteering Problem

Un nombre limité de véhicules sont disponibles pour visiter les clients d'un ensemble potentiel, le déplacement de chaque véhicule étant limité par son autonomie maximale, les clients ont des profits correspondants différents, et chaque client est visité au plus une fois. L'objectif du TOP est d'organiser un itinéraire de visites de manière à maximiser le profit total. Les véhicules partent d'un dépôt au début de leur tournée, et doivent y retourner à la fin de celle-ci.

2.1 I - A. Méthodes exacte prééxistantes

2.1.1 Cutting plane

La méthode cutting plane est une méthode d'optimisation utilisée pour résoudre des problèmes d'optimisation convexe. Elle consiste à diviser le problème en sous-problèmes plus faciles à résoudre en ajoutant des contraintes supplémentaires au problème initial à chaque itération. Ces contraintes sont appelées "coupes planes" car elles sont représentées par des hyperplans dans l'espace des variables du problème. La méthode cutting plane peut également être utilisée pour résoudre des problèmes d'optimisation non convexe en introduisant des approximations du problème initial.

2.1.2 Branch and Bound

La méthode Branch and Bound est une méthode d'optimisation utilisée pour résoudre des problèmes d'optimisation combinatoire. Elle consiste à diviser récursivement un problème en sous-problèmes plus petits en imposant des contraintes supplémentaires sur les variables du problème. Cela permet d'éviter d'avoir à explorer toutes les solutions possibles du problème initial et ainsi d'améliorer les performances de résolution du problème. La méthode Branch and Bound peut également être utilisée pour résoudre des problèmes d'optimisation non convexe en introduisant des approximations du problème initial.

2.1.3 Dynamic Programming

La méthode de programmation dynamique est une méthode d'optimisation utilisée pour résoudre des problèmes d'optimisation récurrents. Elle consiste à décomposer le problème en sous-problèmes indépendants et à stocker les solutions de ces sous-problèmes pour éviter de les recalculer à chaque itération. Cette méthode est souvent utilisée pour résoudre des problèmes d'optimisation combinatoire tels que les problèmes de chemin le plus court ou de chaîne d'assemblage.

2.2 I - B. Méthodes approximatives préexistentes

2.2.1 Genetic Algorithm

L'algorithme génétique (ou genetic algorithm) est une méthode d'optimisation inspirée de la sélection naturelle. Elle consiste à générer une population de solutions candidates pour le problème à résoudre, puis à utiliser des opérateurs de reproduction et de mutation pour générer de nouvelles solutions à partir des solutions existantes. Les solutions sont ensuite évaluées selon une fonction de coût et les meilleures solutions sont conservées pour la génération suivante. Le processus est répété jusqu'à ce que l'on trouve une solution optimale ou qu'un nombre maximal d'itérations ait été atteint.

2.2.2 Local search algorithm

L' algorithme de recherche locale (ou Local search algorithm) est une méthode d'optimisation qui consiste à partir d'une solution initiale pour un problème d'optimisation et à en chercher une meilleure en effectuant des modifications locales à cette solution. Ces modifications peuvent prendre la forme d'ajouts, de suppressions ou de modifications d'éléments de la solution initiale. L'algorithme s'arrête lorsqu'une solution optimale est trouvée ou lorsqu'un nombre maximal d'itérations a été atteint.

2.2.3 Optimisation par essaims particulaires

L'optimisation par essaims particulaires (PSO, de l'anglais Particle Swarm Optimization) est une technique d'optimisation utilisée pour trouver des solutions optimales à des problèmes complexes. Elle repose sur l'idée de simuler le comportement d'un essaim d'insectes ou de créatures, chacune suivant une stratégie pour atteindre un objectif commun. L'algorithme PSO utilise une méthode d'essai et d'erreur pour évaluer les différentes solutions possibles et déterminer celle qui est la plus optimale pour le problème à résoudre.

3 II- Notre méthode de résolution

3.0.1 Imports des librairies nécessaires

[2]:
```
import pandas as pd
import numpy as np

%matplotlib inline
%config InlineBackend.figure_format = 'retina'
import matplotlib.pyplot as plt
```

3.0.2 Création des coordonnées de tous les points

[3]:
```
# Seed to generate the values randomly and always have the same values␣
 ↪throughout the multiple tests and executions.
np.random.seed(4)

# number of customers, min and max of the axis
size, cmin, cmax = 75, -100, 100

dataHomeCoord = pd.DataFrame(dict(x=[0], y=[0]))
dataCoord = pd.DataFrame((np.random.random_sample(2*size)*(cmax-cmin)+cmin).
 ↪reshape(-1,2), columns=['x', 'y'])
data = pd.concat([dataHomeCoord, dataCoord], ignore_index=True)

data
```

[3]:
```
             x          y
0     0.000000   0.000000
1    93.405968   9.446450
2    94.536872  42.963199
3    39.545765 -56.782101
4    95.254891 -98.753949
..         ...        ...
71 -18.475665  56.306225
72 -29.170892  85.161719
73  44.090424  10.385288
74 -63.807033 -18.081722
75  52.576358  98.159364

[76 rows x 2 columns]
```

3.0.3 Création des intérêts de tous les points

[115]:
```
np.random.seed(4)
# min and max of the customers' interests
imin, imax = 1, 100
dataHomeInterest = pd.DataFrame(dict(interest=[1]))
dataOtherInterest = pd.DataFrame((np.random.
 ↪random_sample(size)*(imax-imin)+imin).reshape(-1,1), columns=['interest'])
dataInterest = pd.concat([dataHomeInterest, dataOtherInterest],␣
 ↪ignore_index=True)

dataInterest
```

[115]:
```
     interest
0    1.000000
1   96.735954
2   55.175993
3   97.295752
4   71.766783
..        ...
71  73.526751
72  89.570478
73  51.958663
74  60.752788
75   7.441713

[76 rows x 1 columns]
```

3.0.4 Fonction utilitaire pour le facteur d'intérêt

Cette fonction prend en argument une lambda fonction qui calculera le facteur d'intérêt pour chaque trajet entre 2 points. Le facteur d'intérêt est de la forme : distance/intérêt, avec l'intérêt du client d'arrivé qui varie de 0 à 100.

Donc plus le facteur est petit, plus le camion devrait aller vers ce client en question. Plus il est grand, moins le camion a intérêt à aller le visiter.

[131]:
```
distance_func = lambda a,b,interest: (np.sqrt((a.x-b.x)**2 + (a.y-b.y)**2))/
 ↪interest
def compute_interest_factor(dfunc=distance_func):
    for i in range(size+1):
        current=data.iloc[i]
        interest = dataInterest.iloc[i].interest
        data[i]=dfunc(current, data, interest)

%time compute_interest_factor()
data
```

```
CPU times: user 195 ms, sys: 0 ns, total: 195 ms
Wall time: 208 ms
```

[131]:

	x	y	0	1	2	3	4	\
0	0.000000	0.000000	0.000000	0.970502	1.882005	0.711192	1.911849	
1	93.405968	9.446450	93.882428	0.000000	0.607797	0.877374	1.507887	
2	94.536872	42.963199	103.841498	0.346674	0.000000	1.170655	1.974715	
3	39.545765	-56.782101	69.195914	0.882451	2.064299	0.000000	0.971906	
4	95.254891	-98.753949	137.207276	1.118676	2.568490	0.716892	0.000000	
..	...	...	...	...	...	...	...	
71	-18.475665	56.306225	59.259945	1.253914	2.062446	1.306369	2.679477	
72	-29.170892	85.161719	90.019217	1.489374	2.368911	1.620855	3.094067	
73	44.090424	10.385288	45.297017	0.509888	1.088360	0.691921	1.679566	
74	-63.807033	-18.081722	66.319576	1.649903	3.075676	1.134282	2.485132	
75	52.576358	98.159364	111.353195	1.009529	1.256610	1.598101	2.807500	

	5	6	7	...	66	67	68	\
0	0.729159	3.676547	1.238114	...	0.394655	0.783579	10.750748	
1	2.063060	3.543360	0.918380	...	1.039651	1.978097	3.877157	
2	2.204089	4.930837	0.594198	...	1.021409	2.305605	0.568332	
3	1.414512	0.747582	1.607573	...	1.110833	0.875440	12.168434	
4	2.399491	2.452829	2.014753	...	1.841957	1.992649	15.403277	
..	...	...	...	...	...	...	...	
71	1.083579	6.181952	1.020184	...	0.397772	1.701346	11.611315	
72	1.430835	7.530996	1.048828	...	0.732937	2.198085	13.375924	
73	1.375443	3.207359	0.930679	...	0.505803	1.317144	6.165346	
74	0.217764	5.669928	1.825482	...	1.014334	0.981355	17.740082	
75	2.153155	7.085146	0.205598	...	0.919209	2.624532	6.868746	

	69	70	71	72	73	74	75
0	2.114806	1.972671	0.805964	1.005010	0.871789	1.091630	14.963381
1	3.525689	3.350462	1.649720	1.608521	0.949302	2.627121	13.123021
2	3.810854	4.318391	1.547702	1.459265	1.155752	2.793345	9.317035
3	2.332286	0.772009	1.728678	1.760651	1.295664	1.816556	20.894171
4	3.241037	3.518925	2.615340	2.479067	2.319864	2.935667	27.075117
..	...	...	...	...	...	...	...
71	2.632591	4.284455	0.000000	0.343571	1.493679	1.433877	11.081127
72	2.982530	5.498410	0.418539	0.000000	2.014755	1.792485	11.122990
73	2.812179	2.200932	1.055529	1.168733	0.000000	1.836781	11.849869
74	1.184239	3.690372	1.184766	1.215785	2.147661	0.000000	22.103831
75	3.889044	5.651404	1.121532	0.924123	1.697182	2.707536	0.000000

[76 rows x 78 columns]

[132]:
```
estimate_cost = lambda route: sum(data.iloc[i][j] for i,j in zip(route, route[1:
↪]))
```

```
get_coords = lambda route: list(zip(*[(data.iloc[i].x,data.iloc[i].y) for i in␣
 ↪route]))
```

3.0.5 Préparation des données : clustering

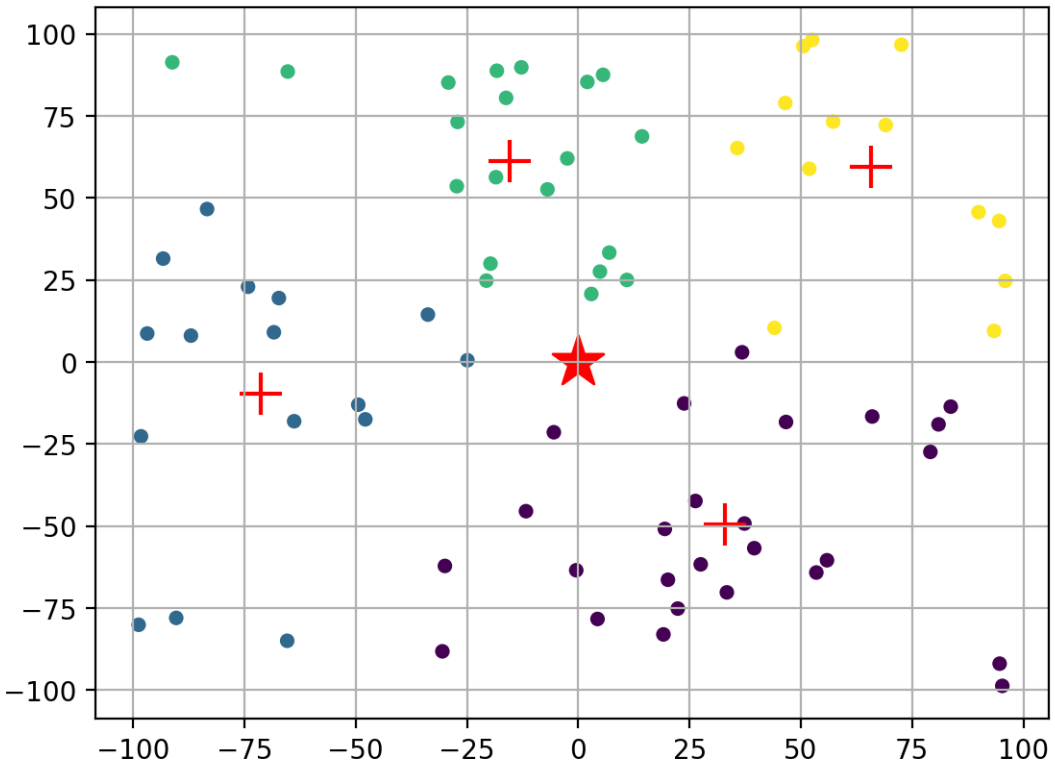
L'objectif est de **découper le nuage de point en plusieurs aires** tel que :
* Le nombre de zones est déterminé par le nombre de camions. * Chaque camion fera une tournée dans une et une seule zone

[155]:
```
from sklearn.cluster import KMeans

nb_trucks = 4
max_distance = 110
model = KMeans(n_clusters=nb_trucks, init='k-means++', random_state=0)
%time model.fit(data[['x', 'y']])
```

```
CPU times: user 1.51 s, sys: 0 ns, total: 1.51 s
Wall time: 326 ms
```

[155]:
```
KMeans(n_clusters=4, random_state=0)
```

[156]:
```
plt.scatter(data.x, data.y, marker='o', c=model.labels_, s=20)
plt.scatter(model.cluster_centers_[:,0], model.cluster_centers_[:,1],␣
 ↪marker='+', c='r', s=250)
plt.scatter(data.iloc[0].x, data.iloc[0].y, marker='*', c='r', s=400)
plt.grid(True)
```



3.0.6 Fonction outil : vérification autonomie du camion pour aller au client suivant

[157]:
```python
# Check if the truck has enough fuel to go to the next customer
def is_too_far(result_path, max_distance):
    result_path.append(0)
    if estimate_cost(result_path) > max_distance:
        # the truck cannot go to the next customer
        result_path.pop() # pop the home
        result_path.pop() # pop the customer he was supposed to go to
        result_path.append(0) # add the home
        return True
    else:
        # the truck can go to the next customer
        result_path.pop() # pop the home because he won't go there yet
        return False
```

3.0.7 Algorithme du plus proche voisin (KNN), avec une distance maximale

Cette fonction prend en argument un tableau des indices de tous les points, un entier représentant la distance maximale, et une fonction lambda. Et elle retourne les indices ordonnés des points par lesquels le camion va passer.

[158]:
```
def nearest_neighbor(unvisited, max_distance, stop_condition=lambda route,␣
↪target: False):
    current=0 # The truck starts at the depot
    result_path=[]
    while True:
        result_path.append(current)
        unvisited.remove(current)
        if not unvisited: # Check if there is no more unvisited customer
            break

        # Choose the nearest neighbor from current, and replace the current by␣
↪the found one (index)
        # => the truck is now on the nearest neighboor
        current=data.iloc[unvisited,current+2].idxmin()

        # Check if the truck has enough fuel to go to the next chosen customer.
        if is_too_far(result_path, max_distance):
            break

    return result_path
```

3.0.8 Test de la fonction *nearest_ neighbor* sur tous les points de la carte

[159]:
```
trajet = nearest_neighbor(list(range(size+1)), 150)
print('Coût = {}\nTrajet = {}'.format(estimate_cost(trajet),trajet))
```

```
Coût = 139.62479623305396
Trajet = [0, 37, 19, 66, 25, 20, 21, 32, 23, 5, 51, 74, 18, 8, 39, 38, 64, 65,
54, 10, 60, 72, 59, 11, 52, 47, 71, 40, 61, 44, 43, 45, 16, 55, 58, 12, 33, 36,
62, 75, 7, 68, 2, 53, 1, 29, 31, 26, 30, 15, 42, 73, 50, 70, 35, 27, 14, 57, 34,
17, 28, 67, 48, 56, 0]
```

[160]:
```
def sum_interest(path):
    return sum(dataInterest.iloc[point[1]].interest for point in zip(path,␣
↪path[1:]))-1
```

3.0.9 Recherche du plus proche voisin pour chaque cluster

Il s'agit ici de définir un premier ordre de clients à visiter pour chacun des clusters définis en utilisant l'algorithme de K-plus proches voisins.

[173]:
```
def greedy_nn_clusters():
    result=[]
    for i in range(model.n_clusters): # for each cluster
        unvisited=data[model.labels_ == i].index.tolist()
        if 0 not in unvisited:
            unvisited.append(0)
```

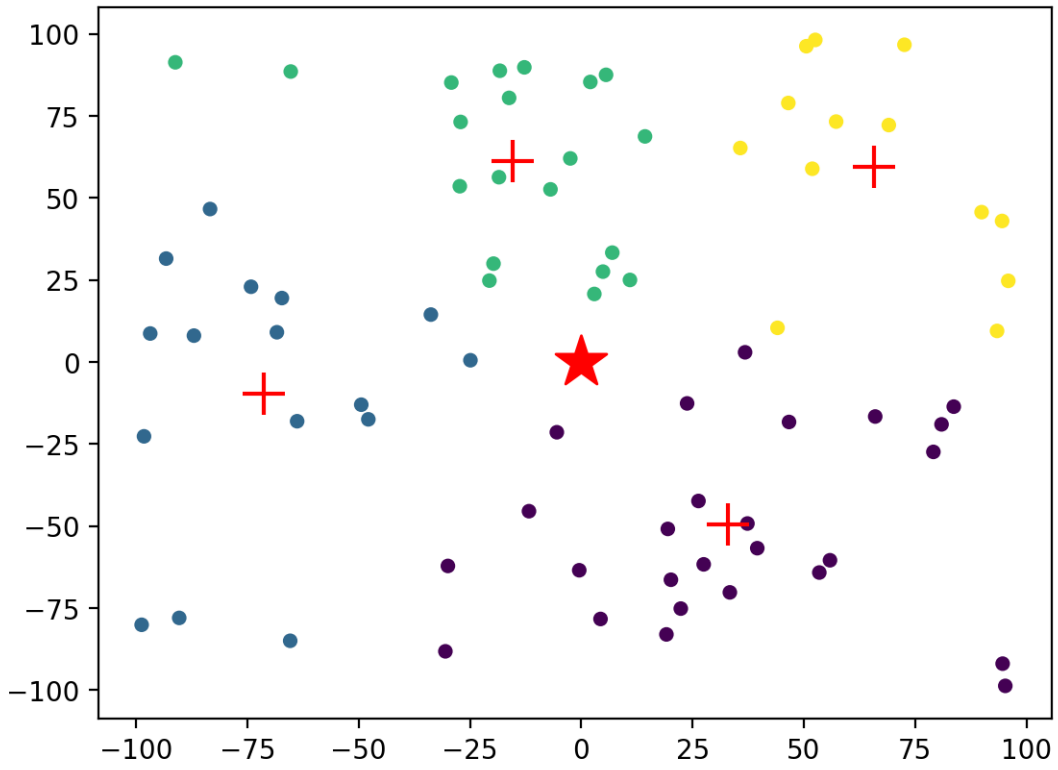
```
        result_path = nearest_neighbor(unvisited, max_distance)
        result.append(result_path)
        print('Coût du trajet du camion {}: {}'.format(i+1,␣
 ↪estimate_cost(result_path)))
        print('Intérêt du trajet du camion {}: {}'.format(i+1,␣
 ↪sum_interest(result_path)))
        print()
    return result

paths_to_improve = []
compare_algo_before = greedy_nn_clusters()
```

```
Coût du trajet du camion 1: 93.248758798498
Intérêt du trajet du camion 1: 1137.6813923714778

Coût du trajet du camion 2: 105.79319002824772
Intérêt du trajet du camion 2: 701.1991046770438

Coût du trajet du camion 3: 95.73011302910469
Intérêt du trajet du camion 3: 1082.8084794341612

Coût du trajet du camion 4: 78.09708142338158
Intérêt du trajet du camion 4: 274.18446655639013
```

3.0.10 Affichage des résultats

[174]:
```
# Plot customers' positions
plt.scatter(data.x, data.y, marker='o', c=model.labels_, s=20)

# Plot clusters' centers
plt.scatter(model.cluster_centers_[:,0], model.cluster_centers_[:,1],␣
 ↪marker='+', c='r', s=250)

# Plot paths
for path in paths_to_improve:
    coords=get_coords(path)
    plt.plot(coords[0], coords[1], linewidth=2)

# Plot home
plt.scatter(data.iloc[0].x, data.iloc[0].y, marker='*', c='r', s=400)
```

[174]: <matplotlib.collections.PathCollection at 0x7fc60aae41c0>



3.0.11 Amélioration de l'algorithme KNN avec l'heuristique 2-opt

2-opt est un algorithme itératif : à chaque étape, on supprime deux arêtes de la solution courante et on reconnecte les deux tours ainsi formés. Cette méthode permet, entre autres, d'améliorer le coût des solutions en supprimant les arêtes sécantes lorsque l'inégalité triangulaire est respectée.

Plus généralement, lorsqu'on inverse l'ordre de parcours de deux villes, il faut aussi inverser l'ordre de parcours de toutes les villes entre ces deux villes.

[163]:
```
def twoOptSwap(route,i,j):
    t=route[i:j+1];
    t.reverse();
    route[i:j+1]=t

def twoOpt(route):
    cost=estimate_cost(route)

    def inner_function():
        nonlocal cost # refers to the cost above
        for i in range(1,len(route)-2):
            nodei=data.iloc[route[i]]
            for j in range(i+1,len(route)-1):
                nodej=data.iloc[route[j]]
```

```
                save=nodei.iloc[route[i-1]+2]+nodej.iloc[route[j+1]+2] - (nodej.
↪iloc[route[i-1]+2]+nodei.iloc[route[j+1]+2])
                if save>0:
                    twoOptSwap(route,i,j)
                    cost-=save
                    print('On échange le trajet {}-{},{}-{} avec {}-{},{}-{} =>␣
↪On sauve = {} => Coût : {}'.format(
                                ␣
↪route[i-1],route[i],route[j],route[j+1],route[i-1],route[j],route[i],route[j+1],save,cost))
                    return False
        return True

    while True:
        if inner_function():
            break
    return route

compare_algo_after = []

def improve_seq_nn():
    for i,path in enumerate(paths_to_improve):
        print('Trajet {}/{}'.format(i+1, len(paths_to_improve)))
        compare_algo_after.append(twoOpt(path))
    return compare_algo_after

%time compare_algo_after=improve_seq_nn()
```

```
Trajet 1/4
On échange le trajet 49-50,67-48 avec 49-67,50-48 => On sauve =
1.751041547081732 => Coût : 91.49771725141628
On échange le trajet 49-48,50-0 avec 49-50,48-0 => On sauve = 41.78473014961796
=> Coût : 49.71298710179832
On échange le trajet 14-35,27-22 avec 14-27,35-22 => On sauve =
0.041067306709026086 => Coût : 49.671919795089295
On échange le trajet 24-27,3-22 avec 24-3,27-22 => On sauve =
0.016061438431312247 => Coût : 49.65585835665798
Trajet 2/4
On échange le trajet 23-54,32-0 avec 23-32,54-0 => On sauve = 58.0503129544192
=> Coût : 47.742877073828524
On échange le trajet 23-38,54-39 avec 23-54,38-39 => On sauve =
0.00978754158167483 => Coût : 47.73308953224685
On échange le trajet 23-18,38-74 avec 23-38,18-74 => On sauve =
0.10965232655535262 => Coût : 47.623437205691495
On échange le trajet 23-5,18-32 avec 23-18,5-32 => On sauve =
0.17035541732220538 => Coût : 47.45308178836929
Trajet 3/4
On échange le trajet 37-66,19-25 avec 37-19,66-25 => On sauve =
```

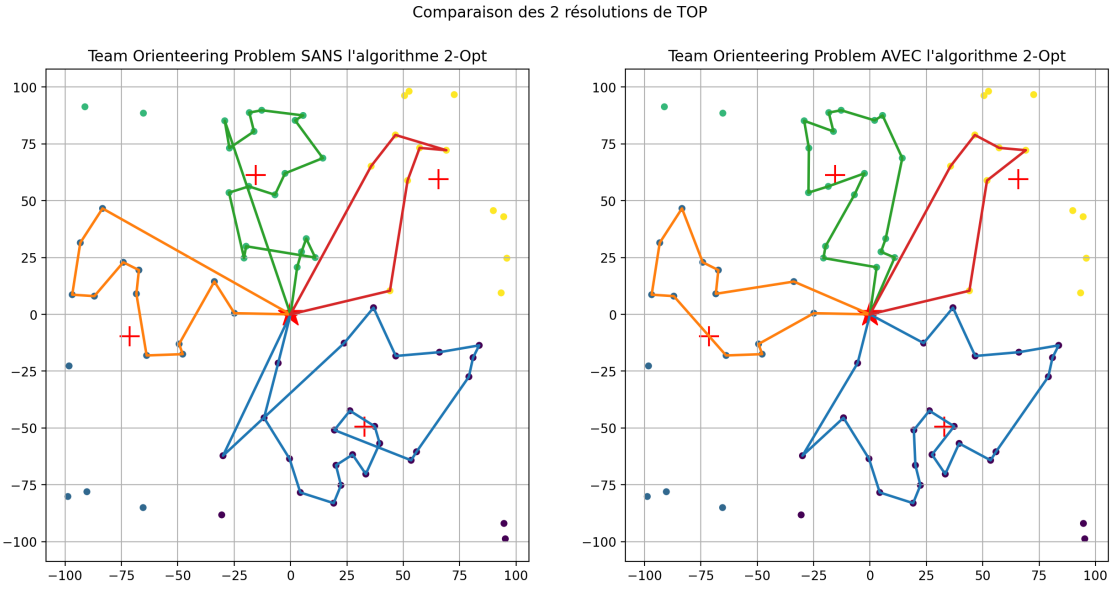
```
0.02964550909682573 => Coût : 95.70046752000786
On échange le trajet 37-25,66-20 avec 37-66,25-20 => On sauve =
0.12203188207660653 => Coût : 95.57843563793125
On échange le trajet 37-19,25-66 avec 37-25,19-66 => On sauve =
0.0021239024510673887 => Coût : 95.57631173548019
On échange le trajet 37-72,19-0 avec 37-19,72-0 => On sauve = 60.840696194717 =>
Coût : 34.73561554076319
On échange le trajet 37-47,72-52 avec 37-72,47-52 => On sauve =
0.20076235633255268 => Coût : 34.53485318443064
On échange le trajet 37-44,47-61 avec 37-47,44-61 => On sauve =
0.15166321584359999 => Coût : 34.38318996858704
On échange le trajet 37-61,44-71 avec 37-44,61-71 => On sauve =
0.09063989388269811 => Coût : 34.292550074704344
On échange le trajet 37-43,61-44 avec 37-61,43-44 => On sauve =
0.10638098622428527 => Coût : 34.18616908848006
On échange le trajet 37-20,43-66 avec 37-43,20-66 => On sauve =
0.3627211686009969 => Coût : 33.82344791987906
On échange le trajet 37-21,20-40 avec 37-20,21-40 => On sauve =
0.08308188463834987 => Coût : 33.74036603524071
On échange le trajet 20-61,40-47 avec 20-40,61-47 => On sauve =
0.12061996659979535 => Coût : 33.619746068640914
On échange le trajet 11-45,16-43 avec 11-16,45-43 => On sauve =
0.0378607295207849 => Coût : 33.58188533912013
On échange le trajet 66-19,25-0 avec 66-25,19-0 => On sauve = 0.7075139018313017
=> Coût : 32.87437143728883
Trajet 4/4
On échange le trajet 58-33,12-36 avec 58-12,33-36 => On sauve =
0.06110636283925075 => Coût : 78.03597506054233
CPU times: user 287 ms, sys: 60 µs, total: 287 ms
Wall time: 286 ms
```

3.0.12 Comparaison de performances (KNN seul) VS (KNN + 2-opt)

[175]:
```
fig, axs = plt.subplots(1, 2, figsize=(15, 15))
fig.set_figheight(7)
fig.set_figwidth(15)

fig.suptitle('Comparaison des 2 résolutions de TOP')

################################################
### Plot the graph BEFORE the 2-Opt algorithm ###
################################################

# Plot customers' positions
axs[0].scatter(data.x, data.y, marker='o', c=model.labels_, s=20)

# Plot clusters' centers
```

```
axs[0].scatter(model.cluster_centers_[:,0], model.cluster_centers_[:,1],␣
 ↪marker='+', c='r', s=250)

# Plot paths
for path in compare_algo_before:
    coords=get_coords(path)
    axs[0].plot(coords[0], coords[1], linewidth=2)

# Plot home
axs[0].scatter(data.iloc[0].x, data.iloc[0].y, marker='*', c='r', s=400)

axs[0].grid(True)

axs[0].set_title("Team Orienteering Problem SANS l'algorithme 2-Opt")

################################################
### Plot the graph AFTER the 2-Opt algorithm ###
################################################

# Plot customers' positions
axs[1].scatter(data.x, data.y, marker='o', c=model.labels_, s=20)

# Plot clusters' centers
axs[1].scatter(model.cluster_centers_[:,0], model.cluster_centers_[:,1],␣
 ↪marker='+', c='r', s=250)

# Plot paths
for path in compare_algo_after:
    coords=get_coords(path)
    axs[1].plot(coords[0], coords[1], linewidth=2)

# Plot home
axs[1].scatter(data.iloc[0].x, data.iloc[0].y, marker='*', c='r', s=400)

axs[1].grid(True)

axs[1].set_title("Team Orienteering Problem AVEC l'algorithme 2-Opt")
```

[175]: Text(0.5, 1.0, "Team Orienteering Problem AVEC l'algorithme 2-Opt")



On remarque que les chemins qui se croisent dans le premier graphe, ne se croisent plus dans le second. Nous avons en effet un chemin plus optimisé grâce à l'algorithme 2-Opt.

3.1 Conclusion

L'association de l'algorithme des K-plus proches voisins et de l'algorithme 2-opt est une solution intéressante car ils permettent un résultat satisfaisant pour un nombre d'instances contenu (quelques centaines au plus). Dans notre cas d'étude, notre nombre de clients est limité à 75 et ils ne sont pas tous visités car chacun des camions respectent bien leur contrainte de distance parcourable.

On obtient plusieurs tournées que l'on peut qualifier de "réalistes" et réalisables dans un cas concret.