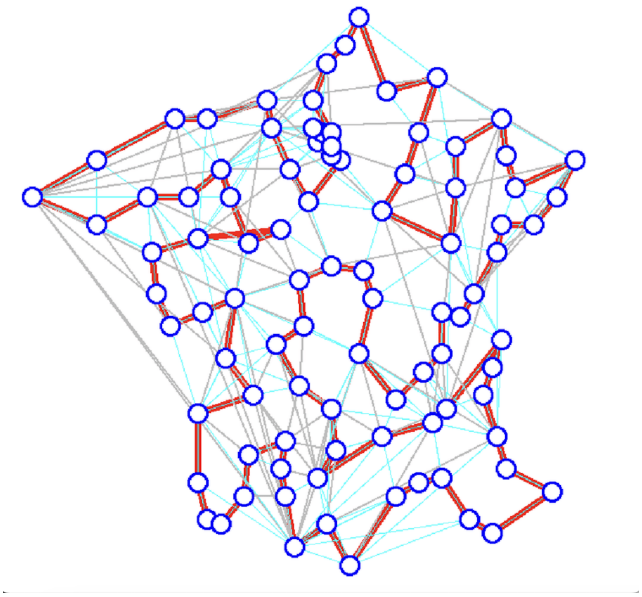
AI09 - Projet

Problème de tournées sélectives



Étudiants Clément MARTINS
Quentin VALAKOU

Enseignant responsable Aziz MOUKRIM

Semestre Automne 2024

Ce document comporte 22 pages.

utc
Université de Technologie
Compiègne

I. Introduction

L'objectif de ce rapport est de développer plusieurs méthodes de résolutions pour le problème du TOP (Team Orienteering Problem). Le TOP est modélisé par un graphe complet : $G = (V, E)$ où $V = \{1, ..., n\} \cup \{d, a\}$ est l'ensemble des sommets et $E = \{i, j\} \quad i, j \in V$ est l'ensemble des arcs.

- Une flotte F est composée de m véhicules devant aller livrer des clients. Chaque véhicule possède un parcours de temps limite L considéré qu'il ne doit pas dépasser.
- Le sommet a représente le point de départ des véhicules.
- Le sommt d représente le dépot des véhicules
- Chaque client i est associé à un profit P_i et ne peut être collecté qu'une seule fois.
- Un temps de trajet C_{ij} est associé à chaque arc (i, j).
- V est l'ensemble des sommets

Nous allons donc procéder en deux méthodes distinctes : par la programmation en variables entières, et une adaptation des algorithmes de TSP/VRP programmés en Python.

II. Modélisation pour la programmation linéaire

La programmation linéaire est une modélisation mathématique par des combinaisons linéaires de variables respectant une liste de contraintes données. Une fois la modélisation mathématique completée, il est possible de calculer les solution grâce à un solveur. La complexité est dépendante du problème.

Variables :

- $x_{i,j}^k \in \{0, 1\}$: 1 si le véhicule a parcouru l'arc (i, j), 0 sinon.
- u_i^k : 1 si le véhicule k pass par le sommet i, 0 sinon

Paramètres :

- P_i : profit associé au client i.
- C_{ij} : coût de trajet de l'arc (i, j).
- L : temps de trajet limite des véhicules.
- m : Nombre de véhicules.
- n : Nombre de sommet

Objectif :

- Maximiser : $Max \sum_{k=1}^{m} \sum_{i=1}^{n} P_i * u_i^k$: soit le profit pour l'ensemble des tournées réalisés par les m vehicules de la flotte F.

Contraintes :

1. **Limite de distance (temps) par véhicule**

$$\forall k \in \{1, \ldots, m\}, \quad \sum_{i \in \mathrm{V}} C_{i,j} * u_i^k \leq L$$

2. **Chaque sommet (sauf a et d) doit avoir exactement deux arcs entrants et sortants, si $u_i^k = 1$.**

$$\forall k \in \{1, \ldots, m\}, \forall i \in \mathrm{V} \setminus \{a, d\}, \quad 2 \cdot u_i^k = \sum_{j \in \mathrm{V}, j \neq i} x_{j,i}^k + \sum_{j \in \mathrm{V}, j \neq i} x_{i,j}^k$$

3. **Passe par a**

$$\forall k \in \{1, \ldots, m\}, \quad u_{k,a} = 1$$

4. **Passe par d**

$$\forall k \in \{1, \ldots, m\}, \quad u_{k,d} = 1$$

5. **Chaque sommet (sauf a et d) est visité au plus une fois**

$$\forall i \in \mathrm{V} \setminus \{a, d\}, \quad \sum_{k=1}^{m} u_i^k \leq 1$$

6. **Élimination des sous-tours** Si un véhicule traverse de i à j, il ne peut pas revenir immédiatement de j à i dans le même trajet.

$$\forall k \in \{1, \ldots, m\}, \forall i, j \in \mathrm{V}, i \neq j, \quad x_{k,i,j} + x_{k,j,i} \leq 1$$

Cependant pour éliminer définitivement les sous tour il faut aussi forcer le client de départ (a) à ne pas avoir d'arc entrant, et le client d'arriver (d) à ne pas avoir d'arc sortant

— **Pas d'arc entrant pour a**

$$\forall k \in 1, \ldots, m, \sum_{i=1}^{n} \quad k_{i,a}^k = 0$$

— **Pas d'arc sortant pour d**

$$\forall k \in 1, \ldots, m, \sum_{i=1}^{n} \quad k_{d,i}^k = 0$$

III. Algorithmes pour le TOP

Dans cette partie, nous allons développer deux algorithmes pour résoudre le problème de tournées sélectives.

— **Algorithme 1 : algorithme de programmation linéaire**. À l'aide de la modélisation présentée en partie 1, nous allons écrire un algorithme en langage Python avec les libaires numpy (représentation matricielle, matplotlib (création de graphique), xpress (API de programmation linéaire avec contraintes).

— **Algorithme 2 : algorithme glouton (ou heuristique gloutonne)** qui va chercher à chaque itération une solution locale optimale. Les algorithmes gloutons ne garantissent pas l'obtention d'une solution optimale et peuvent avoir une complexité supérieure mais sont plus aisés à implémenter. Cette algorithme sera écrit grâce à la librairie networkX pour la création de graphe.

L'objectif est de comparer les deux algorithmes sur deux critères : leur capacité à trouver une (ou plusieurs) solutions (optimale ou non) et leur complexité temporelle.

Pour exécuter le code sur votre machine, assurez vous d'installer Python et les dépendances suivantes grâce à Pip sur votre environnement :

```
pip install matplotlib
pip install numpy
pip install networkx
pip install xpress
pip install itertools
```

Le code est disponible sur Gitlab : Problème de tournées

III.1. Algorithme 1 : par la programmation linéaire

III.1.1. Visualisation des Points et des Gains

Nous avons modélisé les points du problème, incluant les clients, ainsi que les profits associés à chaque client. Les clients sont représentés par des points dans un plan 2D, et chaque point a un profit associé qui est affiché sur le graphique. Le point de départ a est situé en $(0, 0)$ et le point d'arrivée d en $(10, 10)$.

Voici la visualisation des points (clients, départ, arrivée) et des gains associés :

III.1.2. Génération du modèle

```python
import matplotlib.pyplot as plt
import numpy as np
import xpress as xp

# Nombre de clients
n_clients = 5

# Positions des clients et points de départ (a) et d'arrivée (d)
points = {
    'a': (0, 0, 0),  # Point de départ
    'd': (10, 10, 0),  # Point d'arrivée
}

# Génération de positions aléatoires pour les clients
#ainsi que le gain associé (3 ème chiffre)
np.random.seed(50)  # Pour la reproductibilité
for i in range(1, n_clients + 1):
    points[i] = (np.random.uniform(1, 10),
                 np.random.uniform(1, 10),
                 np.random.randint(1, 100))

# Création du graphique
plt.figure(figsize=(8, 8))
for key, value in points.items():
    plt.scatter(value[0], value[1],
        label=f'Client {key}'
        if isinstance(key, int) else key)
    plt.text(value[0] + 0.2, value[1] + 0.2,
            f'Gain: {value[2]}',
            fontsize=12)

# Titres et légende
```

```
plt.title('Représentation des points',
        fontsize=14)
plt.xlabel('Coordonnée X')
plt.ylabel('Coordonnée Y')
plt.legend(loc='upper left')
plt.grid(True)
plt.show()
```

III.1.3. Affichage des Coordonnées et Profits des Clients

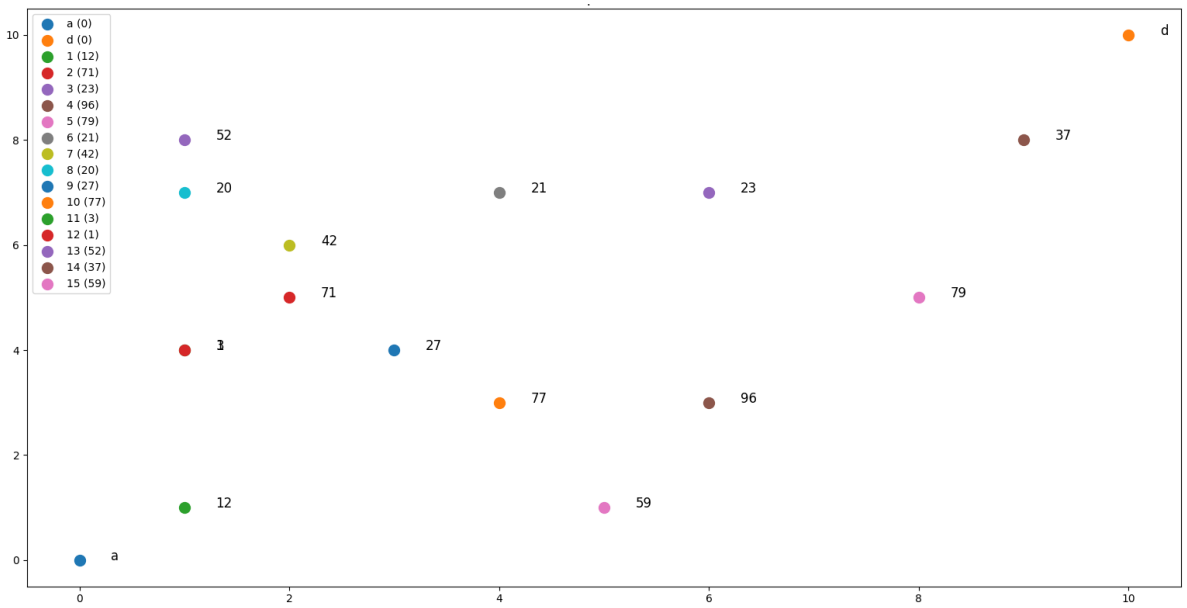


Figure 1 - Modélisation des clients avec leur gain associé.

III.1.4. Programmation du problème avec XpressNP

Xpress est un solveur de programmation mathématique qui applique des techniques avancées pour trouver des solutions optimales dans des problèmes complexes de *programmation linéaire* (PL), comme celui que nous avons formulé.

Xpress utilise les étapes suivantes pour résoudre ce problème :

— **Modélisation du problème** : Nous avons formulé le problème comme un problème de tournée de véhicules où chaque véhicule doit parcourir un ensemble de clients, avec des contraintes sur la distance maximale qu'il peut parcourir et la capacité qu'il peut transporter.

— **Branch-and-Bound et Branch-and-Cut** : Xpress utilise ces algorithmes pour explorer les différentes solutions possibles en divisant le problème en sous-problèmes. Les solutions qui ne sont pas prometteuses sont rapidement écartées à l'aide de la méthode de coupe, ce qui réduit considérablement l'espace de recherche.

— **Heuristiques** : Afin de trouver des solutions faisables rapidement, Xpress applique des heuristiques telles que les heuristiques gloutonnes et de proximité. Ces heuristiques permettent de générer des solutions initiales qui peuvent ensuite être améliorées.

Dans notre cas, Xpress permet de résoudre efficacement un problème combinatoire complexe en utilisant une approche hybride d'optimisation, combinant la recherche exacte et des techniques heuristiques. Cela garantit à la fois la faisabilité de la solution et une optimisation rapide des objectifs.

III.1.5. Définition du modèle avec XpressNP

On doit d'abord finir notre modélisation en définissant **les gains**, **coût**, ainsi que nos **paramètres** (Nombre de véhicule, Distance maximum, nombre de Sommet) :

```python
#Calculer les distances entre les points
def distance(p1, p2):
    return np.sqrt((p1[0] - p2[0])**2 + (p1[1] - p2[1])**2)

# Calculer la matrice des distances
C = {}
for i in points:
    for j in points:
        if i != j:
            C[(i, j)] = distance(points[i], points[j])

# Définir les gains associés aux clients
P = {i: points[i][2] for i in range(1, n_clients + 1)}

# Paramètres du problème
L = 15  # Distance limite par véhicule
#(La diagonale est en 14, on leur laisse un peu de marge)
# Nombre de véhicules
m = 4
# Nombre de sommets (clients + points de départ et d'arrivée)
n = len(points)

```

Matrice des distances généré :

points	*a*	*d*	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	0	14.14	1.41	5.38	9.22	6.71	9.43	8.06	6.32	7.07	5.00	5.00	4.12	4.12	8.06	12.04	5.10
d	14.14	0	12.73	9.43	5.00	8.06	5.39	6.71	8.94	9.49	9.22	9.22	10.82	10.82	9.22	2.24	10.30
1	1.41	12.73	0	4.12	7.81	5.39	8.06	6.71	5.10	6.00	3.61	3.61	3.00	3.00	7.00	10.63	4.00
2	5.38	9.43	4.12	0	4.47	4.47	6.00	2.83	1.00	2.24	1.41	2.83	1.41	1.41	3.16	7.62	5.00
3	9.22	5.00	7.81	4.47	0	4.00	2.83	2.00	4.12	5.00	4.24	4.47	5.83	5.83	5.10	3.16	6.08
4	6.71	8.06	5.39	4.47	4.00	0	2.83	4.47	5.00	6.40	3.16	2.00	5.10	5.10	7.07	5.83	2.24
5	9.43	5.39	8.06	6.00	2.83	2.83	0	4.47	6.08	7.28	5.10	4.47	7.07	7.07	7.62	3.16	5.00
6	8.06	6.71	6.71	2.83	2.00	4.47	4.47	0	2.24	3.00	3.16	4.00	4.24	4.24	3.16	5.10	6.08
7	6.32	8.94	5.10	1.00	4.12	5.00	6.08	2.24	0	1.41	2.24	3.61	2.24	2.24	2.24	7.28	5.83
8	7.07	9.49	6.00	2.24	5.00	6.40	7.28	3.00	1.41	0	3.61	5.00	3.00	3.00	1.00	8.06	7.21
9	5.00	9.22	3.61	1.41	4.24	3.16	5.10	3.16	2.24	3.61	0	1.41	2.00	2.00	4.47	7.21	3.61
10	5.00	9.22	3.61	2.83	4.47	2.00	4.47	4.00	3.61	5.00	1.41	0	3.16	3.16	5.83	7.07	2.24
11	4.12	10.82	3.00	1.41	5.83	5.10	7.07	4.24	2.24	3.00	2.00	3.16	0	0.00	4.00	8.94	5.00
12	4.12	10.82	3.00	1.41	5.83	5.10	7.07	4.24	2.24	3.00	2.00	3.16	0.00	0	4.00	8.94	5.00
13	8.06	9.22	7.00	3.16	5.10	7.07	7.62	3.16	2.24	1.00	4.47	5.83	4.00	4.00	0	8.00	8.06
14	12.04	2.24	10.63	7.62	3.16	5.83	3.16	5.10	7.28	8.06	7.21	7.07	8.94	8.94	8.00	0	8.06
15	5.10	10.30	4.00	5.00	6.08	2.24	5.00	6.08	5.83	7.21	3.61	2.24	5.00	5.00	8.06	8.06	0

III.1.6. Programmation linéaire avec Xpress

La première étape consiste à définir le modèle d'optimisation avec Xpress.
Nous définissons les variables de décision :

— x_{ij}^k
— u_i^k

Ces variables sont ensuite ajoutées au modèle.

L'objectif du modèle est de maximiser les profits, ce qui est formulé à l'aide de la fonction `setObjective`.

```
# Définition des Variables et de notre objectif
model = xp.problem() # instance d'un solveur Xpress

# Variables de décision
x = {(k, i, j): xp.var(vartype=xp.binary)
    for k in range(m)
    for i in points
    for j in points
    if i != j
}
u = {(k, i): xp.var(vartype=xp.binary)
    for k in range(m)
    for i in points
}

# Ajout des variables au modèle
model.addVariable(x)
model.addVariable(u)

# Objectif : Maximiser les profits
model.setObjective(
    xp.Sum(P[i] * u[k, i]
        for k in range(m)
        for i in points
        if i != 'a' and i != 'd'
    ),
    sense=xp.maximize
)
```

III.1.7. Définition des contraintes avec Xpress

Nous pouvons ensuite définir avec Xpress nos **contraintes** vue précédament

```
# Contraintes

# Limite de distance (temps) par véhicule
for k in range(m):
    model.addConstraint(
        xp.Sum(C[i, j] * x[k, i, j]
            for i in points
            for j in points
            if i != j
        ) <= L  # Pas plus de L par véhicule
    )

# Lier les variables u[k, i] et x[k, i, j] et ainsi
obligé qu'il y ai 2 ARC sur chaque sommet
for k in range(m):
    for i in points:
        if i not in ['a', 'd']:
            model.addConstraint(
                (2*u[k, i]
                ==
                (xp.Sum(x[k, j, i] for j in points if j != i) +
                xp.Sum(x[k,i,j] for j in points if j != i)))
            )

#Commence par a
for k in range(m):
    model.addConstraint(
        xp.Sum(x[k, 'a', j] for j in points if j!='a') == 1
    )
    model.addConstraint(
        xp.Sum(u[k, 'a'] ) == 1
    )
    #Pas de sous tour avec a
    model.addConstraint(
         xp.Sum(x[k, j, 'a'] for j in points if j!='a') == 0
    )


#Termine par d
for k in range(m):
```

```
    model.addConstraint(
        xp.Sum(x[k, j, 'd'] for j in points if j!='d') == 1
    )
    model.addConstraint(
        xp.Sum(u[k, 'd'] ) == 1
    )
    #Pas de sous tour avec d
    model.addConstraint(
         xp.Sum(x[k, 'd', j] for j in points if j!='d') == 0
    )

# Chaque sommet est visité une seul fois
for i in points:
    if i not in ['a', 'd']:
        model.addConstraint(
            xp.Sum(u[k, i] for k in range(m)) <= 1
        )

# Ajout des contraintes d'élimination des sous-tours
for k in range(m):
    for i, j in itertools.combinations([i for i in points], 2):
        # Contrainte pour empêcher le sous-tour : si on va de i à j,
        # alors on ne peut pas revenir de j à i dans le même trajet
        model.addConstraint(
            x[k, i, j] + x[k, j, i] <= 1
            # Si x[k, i, j] = 1, alors x[k, j, i] doit être 0
        )

```

III.1.8. Génération de la solution

Finalement on génère la Solution avec Xpress avec **model.solve()**
Voici dans notre example le résultat de la solution optimal trouvé et tracé :

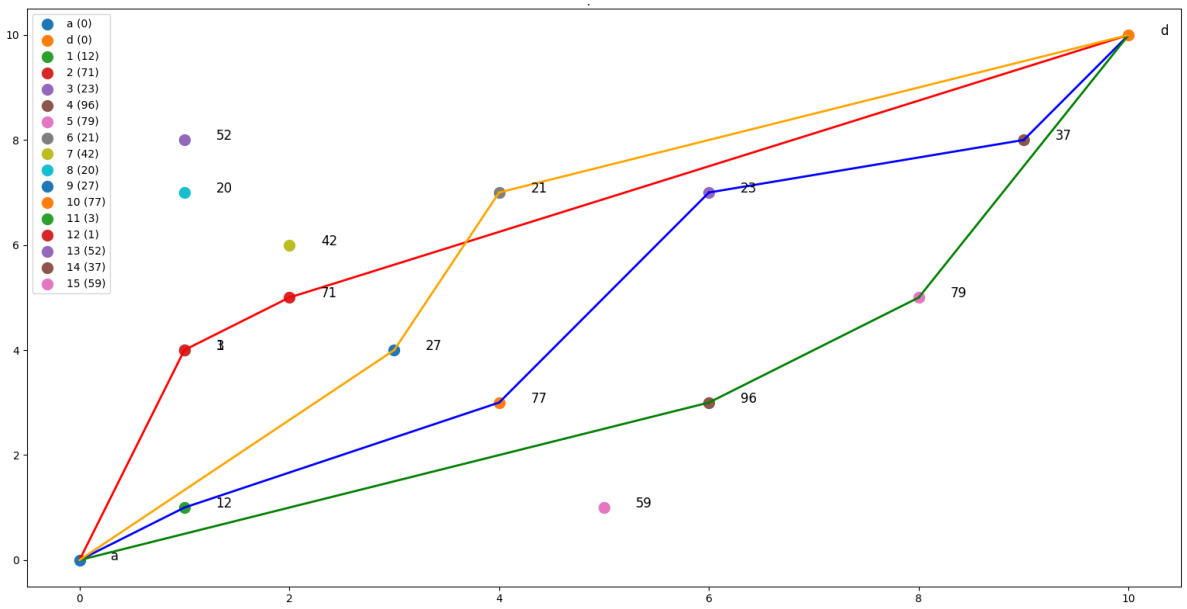


Figure 2 - Solution optimale via Xpress.

Véhicule 1 :
- Traverse de a à 1 (Coût : 1.41)
- Traverse de 1 à 10 (Coût : 3.60)
- Traverse de 10 à 3 (Coût : 4.47)
- Traverse de 3 à 14 (Coût : 3.16)
- Traverse de 14 à d (Coût : 2.23)

Résumé Véhicule 1 :
- Coût total : 14.89
- Gain total : 149

Véhicule 2 :
- Traverse de a à 4 (Coût : 7.70)
- Traverse de 4 à 5 (Coût : 2.82)
- Traverse de 5 à d (Coût : 5.38)

Résumé Véhicule 2 :
- Coût total : 14.92
- Gain total : 175

Véhicule 3 :
- Traverse de a à 12 (Coût : 4.12)
- Traverse de 12 à 11 (Coût : 0)
- Traverse de 11 à 2 (Coût : 1.41)

— Traverse de 2 à d (Coût : 9.43)

Résumé Véhicule 3 :

— Coût total : 14.97
— Gain total : 75

Véhicule 4 :

— Traverse de a à 9 (Coût : 5.00)
— Traverse de 9 à 6 (Coût : 3.16)
— Traverse de 6 à d (Coût : 6.70)

Résumé Véhicule 4 :

— Coût total : 14.87
— Gain total : 48

Résumé global :

— Coût total de l'ensemble de la flotte : 59.6538
— Gain total de l'ensemble de la flotte : 447

Voici quelques points clés à noter :

— **Respect de la contrainte de distance limite :** Pour tous les véhicules, les coûts totaux sont proches de la limite de 15, ce qui indique que la contrainte de distance a été bien gérée. Cela signifie que les véhicules n'ont pas effectué de trajets trop longs, respectant ainsi la capacité de parcours définie par le modèle.
— **Temps de résolution :** Le modèle a été résolu en 3.82 secondes, ce qui témoigne de l'efficacité du solveur Xpress pour ce type de problème.

III.1.9. Complexité de l'algorithme 1

Théorique

La complexité peut être divisée en deux parties :

— Complexité de la création de la matrice, des contraintes et variables
— Complexité de la résolution du solveur.

La création de la matrice de distance est de $O(n^2)$ pour n sommets (clients). De plus, la condition de vérification des contraintes par véhicule est de l'ordre de $O(n^2 * m)$ avec m nombre de véhicules de la flotte. L'affichage quand à lui à une complexité de $O(m*n)$.

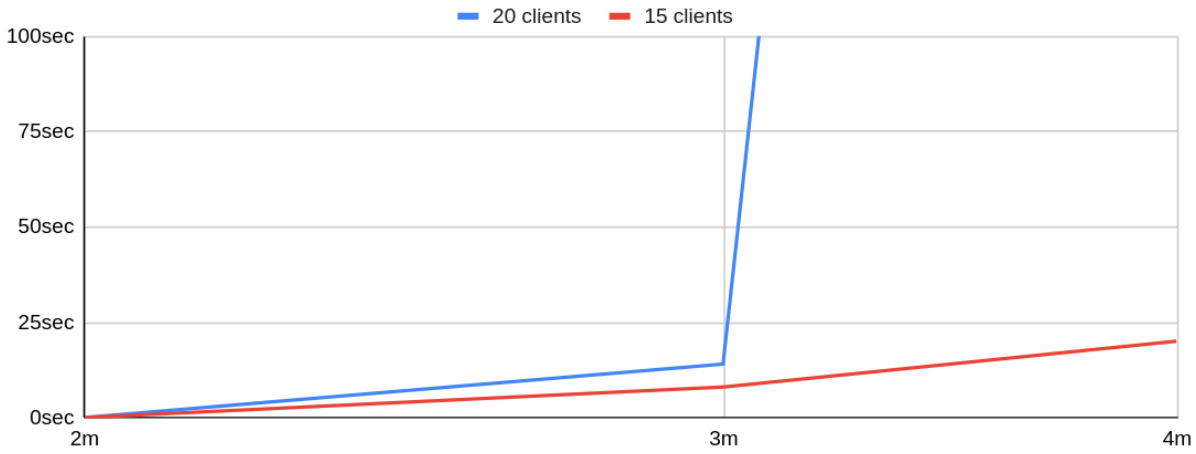
Pour la complexité du solveur, l'algorithme du simplexe a une complexité dans le pire des cas exponentielle. Dans notre situation cette dernière peut être bornée par $O(n^3)$.

On en déduit une complexité générale bornée par $O(n^3)$.

Temps d'exécution et paramètres

Pour plusieurs couples de n clients et m véhicules nous avons comparé le temps d'exécution pour évaluer le poids de chaque paramètre dans la durée d'exécution.

— Pour $n = \mathbf{15}$ et $m = \mathbf{4}$
Le temps est de **20 sec**
— Pour $n = \mathbf{20}$ et $m = \mathbf{4}$
Le temps est de **1100 sec**
— Pour $n = \mathbf{15}$ et $m = \mathbf{3}$
Le temps est de **7.75 sec**
— Pour $n = \mathbf{20}$ et $m = \mathbf{3}$
Le temps est de **14.8 sec**
— Pour $n = \mathbf{15}$ et $m = \mathbf{2}$
Le temps est de **0.57 sec**
— Pour $n = \mathbf{20}$ et $m = \mathbf{2}$
Le temps est de **0.55 sec**



Graphique 1 - Temps d'exécution par rapport au nombre de véhicule

Comme on peux le voir ci-dessus le nombre de clients et le nombre de véhicule augment de manière exponentielle le temps de recherche de l'algorithme

III.2. Algorithme 2 : Heuristique gloutonne

L'heuristique gloutonne cherche une solution optimale locale pour trouver une solution optimale globale. Elle est une implémentation plutôt naive pour ce problème du TOP. En effet, l'algorithme glouton peut trouver une solution, mais elle n'est souvent pas optimale.

III.2.1. Algorithme glouton

Algorithm 1 Algorithme glouton pour le TOP

Input: F, G, L
Output: Le trajet optimal, le profit associé, et le graphe modifié G

```
trajet ← []
NonVisitedClient ← G.nodes()
for i ← 1 to F do
    trajet ← [A]
    totalCost ← 0
    totalProfit ← 0
    while NonVisitedClient is not null do
        bestC ← null
        bestP ← 0
        for c in trajet.voisin do
            if c in NonVisitedClient and c ≠ D and c ≠ A then
                p ← c.profit()
                cost ← c.cost() + d.cost()  if totalCost + cost ≤ L then
                    if p > bestP then
                        bestP ← p
                        bestC ← c
                    end
                end
            end
        end
        if bestC not null then
            trajet ← trajet.append(bestC)
            totalCost ← totalCost + c.cost()
            totalProfit ← totalProfit + c.profit()
            NonVisitedClient.remove(c)
        end
    end
end
return trajet, profit, G
```

III.2.2. Code Python

Le code Python est le suivant :

```python
def gluttony(flotte, G, profit_dict, L):
    # Stocker les profits pour chaque véhicule
    profit = {}
    # Stocker les trajets pour chaque véhicule
    trajets = {i: [] for i in flotte}
    # Liste des clients non visités
    NonVisitedClient = list(G.nodes).copy()
    getAllProfit = 0
    for i in flotte:
        j = 0 # Index pour répérer le temps max de trajet pour i.
        print(f"Flotte {i}")
        profit[i] = 0
        trajet = ['A']  # Commence toujours par 'A'
        totalCost = 0
        totalProfit = 0

        # Tant qu'il reste des clients à visiter
        while NonVisitedClient:
            bestC = None
            bestP = 0
            # Parcourt les voisins du dernier nœud
            for c in list(G.neighbors(trajet[-1])):
                if c in NonVisitedClient and c != 'D' and c != 'A':
            # Un client n'est visitable qu'une fois, d
            # de même on ne repasse pas par A
                    p = profit_dict[c]
                     # Fonction networkX du plus court chemin
                     # Dijkstra
                    distance_to_D = nx.shortest_path_length(G,
                    source=c, target='D', weight='coût')
                    cost = G[trajet[-1]][c]['coût'] + distance_to_D
            # Le coût est la distance vers le voisin
            # Si le coût empêche d'arriver jusqu'a D,
            # on élimine cette possibilité
            # Vérifie les contraintes
                    if totalCost + cost <= L[j] and p > bestP:
                        bestC = c
                        bestP = p

            if bestC is not None:  # Si un client valide est trouvé
```

```
                trajet.append(bestC)
                totalCost += G[trajet[-2]][bestC]['coût']
                totalProfit += profit_dict[bestC]
                NonVisitedClient.remove(bestC)
            else:
            # Aucun client viable, termine le trajet pour ce véhicule
                break

        # Ajoute 'D' (destination finale) si possible
        if totalCost + nx.shortest_path_length(G,
        source=trajet[-1], target='D', weight='coût') <= L[j]:
            trajet.append('D')

        trajets[i] = trajet  # Enregistre le trajet pour ce véhicule
        profit[i] = totalProfit
        # Enregistre le profit total pour ce véhicule
        getAllProfit += totalProfit

        j += 1

    return profit, trajets, getAllProfit
```

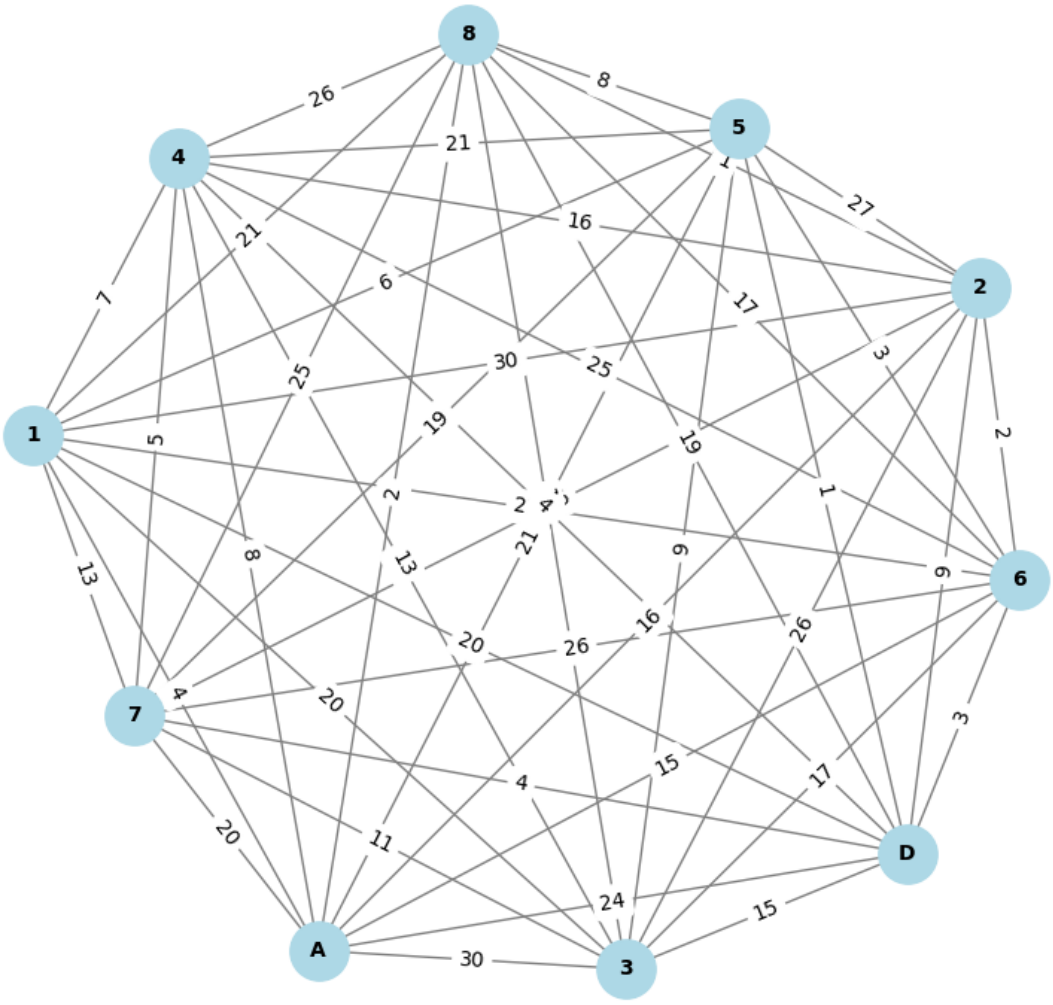


Figure 3 - Représentation du graphe complet $G(V, E)$ par NetworkX pour $V = n = 10$.

	Véhicule 1	**Véhicule 2**
Clients visités dans l'ordre	[A, 8, 2, 6, 5, D] (4 clients)	[A, 1, D] (1 client)
Limite et Coût	L = 11, Coût = 9	L = 25, Coût = 24
Profit réalisé	338	59

Tableau 1 - Résultats obtenu par la tournée pour le graphe G et $F = [1, 2]$

III.2.3. Complexité de l'algorithme 2

Théorique

La recherche du chemin est simple mais peu optimisée :
- On commence par A puis on cherche le voisin avec le meilleur profit, sachant que c'est un graphe complet on a n voisins : $O(n)$
- On calcul ensuite le coût pour y aller mais aussi le coût pour aller à l'arrivée afin d'être sûr de respecter la condition $\leq L$, on utilise l'algorithme du plus court chemin par NetworkX. Par défaut, c'est l'algorithme de Djikstra qui est utilisé soit $O((n+m)log(n))$.
- Cependant, ces opérations précédentes devrons être répétées n fois si aucun chemin n'a été trouvé.
- Puis il passera au voisin suivant.

Asymptomatiquement, on estime que m est négligeable face à n. On en déduit une complexité globale de $n * O(n^2(n+m) * log(n)) = O(n^3 log(n))$. On a une complexité exponentielle qui augmentera considérablement la durée d'éxecution pour un nombre de client important (on estime qu'a partir de $n > 100$ on commence à perdre signicativement.

Temps d'exécution et paramètres

Pour plusieurs couples de n clients et m véhicules nous avons comparé le temps d'exécution pour évaluer le poids de chaque paramètre dans la durée d'exécution.
- Pour $n = \mathbf{10}$ et $m = \mathbf{2}$
  Le temps est de $\mathbf{3}$ μs
- Pour $n = \mathbf{100}$ et $m = \mathbf{10}$
  Le temps est de $\mathbf{3}$ ms
- Pour $n = \mathbf{500}$ et $m = \mathbf{10}$
  Le temps est de $\mathbf{621}$ ms
- Pour $n = \mathbf{1000}$ et $m = \mathbf{10}$
  Le temps est de $> \mathbf{10}$ min

Nous pouvons constater que le temps de recherche est bien inférieur à l'algorithme avec XpressNP.

III.2.4. Critique

On a observé que la complexité théorique se reflète sur le temps d'exécution. Non seulement l'heuristique gloutonne ne permet pas d'obtenir une solution optimale (en particulier pour des n et m important), elle devient rapidement inutilisable pour un graphe complet avec beaucoup de sommets. Cet approche n'est donc pas envisageable dans l'industrie car il n'est pas rare d'avoir à établir un plan de livraison pour $n > 100$

personnes.

De plus, de l'implémentation réalisée, lors de la recherche, l'algorithme parcourt la flotte (une liste) par l'index 0 pour aller à n. Comme aucun client ne peut être visité plus d'une fois, le véhicule avec le chemin le plus optimale sera toujours le véhicule en première position (index 0) car c'est celui qui aura le plus de possibilités. C'est donc aussi celui où la recherche d'un meilleur trajet est le plus long. Plus nous irons loin dans la flotte, plus le temps de recherche individuel pour chaque véhicule sera écourté. Pour un nombre faible de client et un nombre important de véhicules, nous pouvons arriver à une solution absurde où le véhicule 0 parcourera tous les clients (si L assez grand), laissant les autres avec aucun client.

IV. Conclusion

Dans cette étude, nous avons comparé deux approches pour résoudre le problème de la tournée de véhicules : l'algorithme glouton et la programmation linéaire (PL) avec le solveur Xpress.

— **Algorithme glouton** Cette approche, bien que très rapide, n'a pas nécessairement conduit à des solutions optimales. En effet, l'algorithme glouton fait des choix locaux à chaque étape sans tenir compte de l'impact global de ces choix. Par conséquent, bien qu'il soit capable de fournir une solution rapidement, les solutions obtenues ne maximisent pas toujours les profits.

— **Programmation linéaire avec Xpress** En revanche, le modèle de programmation linéaire avec le solveur Xpress permet de trouver des solutions optimales, en tenant compte de l'ensemble des contraintes et en maximisant l'objectif global (les profits dans notre cas). Bien que cette approche soit plus lente que l'algorithme glouton, elle fournit des solutions qui respectent à la fois la limite de distance et maximisent les gains de manière plus équilibrée entre les véhicules.

En résumé, **l'algorithme glouton** est plus rapide pour un nombre limité de client à visiter, mais il peut mener à des solutions sous-optimales en raison de son approche locale et myope. À l'inverse, la **programmation linéaire avec Xpress**, bien que plus coûteuse en termes de temps de calcul, permet d'obtenir des solutions plus optimales et plus équilibrées.

Les deux solutions sont viables en fonction des choix de cahcun (temps de calcul ou maximiser les profits ?)