Centrale
CASABLANCA

Compte rendu

Truck Split Delivery Vehicle Routing Problem (SD-VRP)

Réalisé par :
ACHBANI Ismail
HAMOUCH Ayoub
YAJJOU Rayane
KHALFA Youssef

Encadré par :
Abdessamad Ait el cadi

05 janvier 2025

Table des matières

Avant propos

1.1 Remerciements

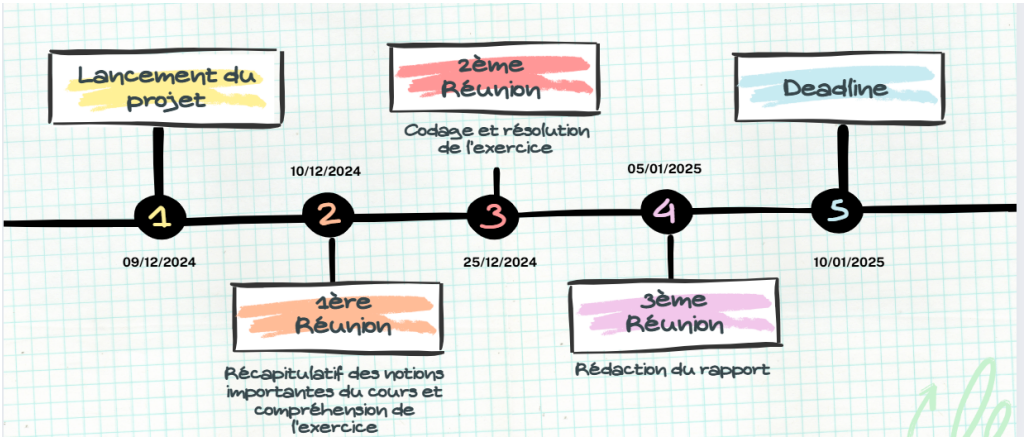
Nous souhaitons exprimer notre profonde gratitude à notre professeur d'Optimisation combinatoire, **Ait el cadi Abdessamad**, pour son accompagnement et ses conseils éclairés tout au long de ce semestre. Nous tenons à souligner la clarté et la richesse de ses enseignements, qui ont su rendre des concepts complexes accessibles et stimulants. Ce projet pratique a été une occasion précieuse de mettre en application les notions apprises, consolidant ainsi nos compétences et approfondissant notre compréhension de l'optimisation combinatoire.

1.2 Résumé

Dans ce projet, nous appliquons des techniques d'optimisation combinatoire pour résoudre le Split Delivery Vehicle Routing Problem (SD-VRP), une variante du problème classique de tournée de véhicules à capacité limitée (CVRP). Nous développons et implémentons plusieurs approches, notamment des solveurs exacts et des (méta-)heuristiques, afin de minimiser le coût total des livraisons tout en respectant les contraintes de capacité des véhicules et en satisfaisant entièrement les demandes des clients. Ce travail met en pratique les concepts théoriques abordés dans le cours et illustre l'efficacité des différentes approches dans un contexte logistique complexe.

1.3 Organisation du travail

Pour illustrer clairement l'organisation et la progression de notre travail au fil du projet, voici un aperçu chronologique des trois réunions que nous avons dûment réalisées :


Lancement du projet
09/12/2024
1
10/12/2024
2
1ère Réunion
Récapitulatif des notions importantes du cours et compréhension de l'exercice
2ème Réunion
Codage et résolution de l'exercice
25/12/2024
3
05/01/2025
4
3ème Réunion
Rédaction du rapport
Deadline
10/01/2025
5

Présentation du projet

Définition du problème

Paramètres

- n : Nombre de clients.
- Q : Capacité maximale de chaque véhicule.
- q_i : Demande du client i ($i = 1, \ldots, n$).
- (x_i, y_i) : Coordonnées des clients et du dépôt.
- d_{ij} : Distance entre les nœuds i et j, calculée par la formule euclidienne arrondie :

$$d_{ij} = \lfloor \sqrt{(x_j - x_i)^2 + (y_j - y_i)^2} + 0.5 \rfloor$$

Variables de décision

- x_{ij}^k : Quantité transportée par le véhicule k entre le nœud i et le nœud j.
- y_{ij}^k : Variable binaire, égale à 1 si le véhicule k utilise l'arc entre i et j, 0 sinon.

Fonction objectif

Minimiser le coût total des trajets :

$$\text{Minimiser} \sum_{i=0}^{n} \sum_{j=0}^{n} d_{ij} \cdot y_{ij}^k$$

M entier supérieur ou égal à 1.

Contraintes

1. **Satisfaction des demandes** :

$$\sum_{k=1}^{M} \sum_{j=0}^{n} x_{ij}^{k} = q_i, \quad \forall i \in \{1, \ldots, n\}$$

2. **Capacité des véhicules** :

$$\sum_{j=0}^{n} x_{ij}^{k} \leq Q \cdot y_{ij}^{k}, \quad \forall i, k$$

3. **Flux d'entrée et de sortie pour chaque nœud** :

$$\sum_{j=1}^{n} y_{ij}^{k} = \sum_{j=1}^{n} y_{ji}^{k}, \quad \forall i \in \{1, \ldots, n\}, \forall k$$

4. **Conservation du flux (retour au dépôt)** :

$$\sum_{j=1}^{n} y_{0j}^{k} = \sum_{j=1}^{n} y_{j0}^{k} = 1, \quad \forall k$$

5. **Non-fragmentation des arcs** :

$$y_{ij}^{k} \in \{0, 1\}, \quad x_{ij}^{k} \geq 0, \quad \forall i, j, k$$

Étapes réalisées

3.1 Prétraitement des données

Le prétraitement des données constitue une étape essentielle pour assurer une utilisation optimale des informations issues d'un fichier brut. Dans notre cas, les données sont fournies sous la forme d'un fichier au format .txt, et il est nécessaire de les convertir en un format structuré afin de permettre une analyse ultérieure.

Étapes du prétraitement

Les étapes du prétraitement sont les suivantes :

1. **Lecture des données** : Lecture du fichier `Case0.txt` ligne par ligne.
2. **Extraction des informations principales** :
   - La première ligne indique le nombre de clients n ainsi que la capacité Q de chaque véhicule.
   - La deuxième ligne contient les demandes associées à chaque client.
   - Les lignes suivantes fournissent les coordonnées des clients ainsi que celles des points de dépôt.
3. **Structuration des données** : Transformation des données extraites en structures exploitables, telles que des listes ou des tableaux `NumPy`.
4. **Calcul des distances** : Construction de la matrice des distances entre tous les points en utilisant la formule de la distance euclidienne.

$$d_{ij} = \left\lfloor \sqrt{(x_j - x_i)^2 + (y_j - y_i)^2} + 0.5 \right\rfloor$$

où d_{ij} représente la distance entre le point i et le point j.

Problème identifié

Lors de la lecture des données brutes, un problème fréquent a été constaté dans la majorité des 32 cas testés. Il s'agissait d'erreurs causées par des espaces superflus ou des formats irréguliers dans le fichier, ce qui rendait la segmentation des données en colonnes incohérente.

Solution adoptée

Pour résoudre ce problème, nous avons utilisé la méthode `strip()` pour éliminer les espaces inutiles et `split()` pour découper les lignes de manière efficace et fiable.

Listing 3.1 – Code de prétraitement des données

```
import numpy as np

# Chemin du fichier
file_path = '/content/Case0.txt'

# Lecture du fichier
with open(file_path, 'r') as file:
    lines = file.readlines()

# Extraction des informations de la premi re ligne
first_line = lines[0].strip().split()
n = int(first_line[0])  # Nombre de clients
Q = int(first_line[1])  # Capacit  de chaque v hicule

# Extraction des informations de la deuxi me ligne
second_line = lines[1].strip().split()
ci = list(map(int, second_line))  # Demandes de chaque client

# Impression des r sultats
print(f"Nombre de clients (n) : {n}")
print(f"Capacit  de chaque v hicule (Q) : {Q}")
print(f"Demandes de chaque client (ci) : {ci}")

# Extraction de la matrice des coordonn es    partir de la
    troisi me ligne
coordinates = []
for line in lines[2:]:
    coordinates.append(list(map(int, line.strip().split())))

# Conversion en tableau NumPy pour des manipulations faciles
coordinates = np.array(coordinates)

# Calcul de la matrice des distances
num_nodes = len(coordinates)
distance_matrix = np.zeros((num_nodes, num_nodes))

for i in range(num_nodes):
    for j in range(num_nodes):
        xi, yi = coordinates[i]
        xj, yj = coordinates[j]
        distance_matrix[i, j] = round(((xj - xi) ** 2 + (yj - yi)
            ** 2) ** 0.5 + 0.5, 2)

# Impression de la matrice des distances
print("\nMatrice des distances :")
print(distance_matrix)
```

Après le prétraitement, les données sont prêtes à être exploitées dans des algorithmes d'optimisation ou de modélisation. La matrice des distances, calculée à l'aide de la formule de la distance euclidienne, servira de point de départ pour les traitements futurs.

3.2 Modélisation et résolution avec un solveur d'optimisation

L'objectif du SD-VRP

L'objectif principal du problème de routage avec livraisons fractionnées (Split Delivery Vehicle Routing Problem, SD-VRP) est de minimiser initialement le nombre de véhicules nécessaires pour satisfaire la demande des clients, puis de réduire la distance totale parcourue par ces véhicules tout en respectant leurs contraintes de capacité.

3.2.1 Présentation des solveurs d'optimisation

Les solveurs d'optimisation sont des outils performants permettant de résoudre des problèmes complexes en trouvant une solution optimale pour un ensemble de contraintes et une fonction objectif. Parmi les solveurs existants, on trouve :

- **Gurobi** : réputé pour sa rapidité et son efficacité dans la résolution de grands problèmes linéaires et non linéaires.
- **CPLEX** : particulièrement performant pour les problèmes combinatoires.
- **CBC (Coin-or Branch and Cut)** : une alternative open-source pour les problèmes d'optimisation linéaire et mixte en nombres entiers.

Choix du solveur pour notre cas

Dans notre cas, nous avons choisi d'utiliser Python avec la bibliothèque `PuLP` et son solveur `PULP_CBC_CMD`. Ce choix se justifie par les avantages suivants :

- **Accessibilité** : `PuLP` est facile à utiliser et entièrement intégré à Python.
- **Flexibilité** : il permet de modéliser et résoudre des problèmes d'optimisation variés.
- **Open-source** : le solveur `CBC` est gratuit et performant pour des problèmes de taille moyenne.

Approche du solveur CBC

Le solveur `CBC` repose sur des méthodes de programmation linéaire, combinées à des techniques de *branch-and-bound* et *branch-and-cut* pour gérer les contraintes d'intégralité. Ces approches permettent de trouver une solution optimale dans des temps raisonnables.

3.2.2 Solveur utilisé

Dans le cadre de ce projet, nous avons utilisé le solveur `CBC` (Coin-or Branch and Cut), un solveur open-source conçu pour résoudre des problèmes de programmation linéaire en nombres entiers mixtes (MILP).

Présentation du solveur CBC

Le solveur `CBC` repose sur des techniques avancées de recherche combinatoire, telles que la méthode *Branch and Cut*, qui combine deux approches principales :

— **Méthode de branchement (Branching)** : Cette méthode divise le problème initial en plusieurs sous-problèmes plus petits en "branchant" sur des variables entières. Par exemple, si une variable x est une variable entière, elle est contrainte à des valeurs comme $x \leq k$ et $x > k$ pour certaines valeurs k.

— **Méthode de coupe (Cutting)** : Des coupes linéaires sont ajoutées pour réduire l'espace de recherche et exclure des solutions non optimales, tout en maintenant la faisabilité du problème initial.

Le solveur `CBC` est particulièrement performant pour des problèmes de petite à moyenne taille. Il a été choisi dans ce projet en raison de sa flexibilité, de son efficacité et de son intégration facile avec des bibliothèques de programmation comme `PuLP`.

Méthode Branch and Cut

Le principe du *Branch and Cut* repose sur une combinaison de :

— **Branch and Bound** : Une approche arborescente qui cherche à réduire l'espace de recherche en calculant des bornes inférieures et supérieures pour chaque sous-problème.

— **Coupes de Gomory et autres coupes** : Ces coupes sont des contraintes supplémentaires, générées automatiquement, qui permettent d'exclure les solutions fractionnaires ou sous-optimales.

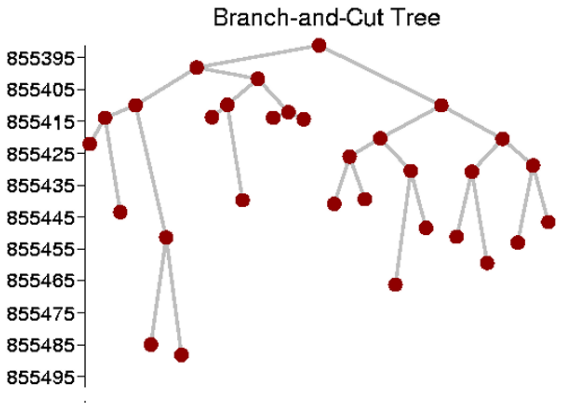


FIGURE 3.1 – Branch and Cut

Dans notre cas, la méthode *Branch and Cut* fonctionne en itérant sur l'ensemble des variables de décision du problème, telles que x_{ijk} et y_{ik}. La méthode suit les étapes suivantes :

1. Génération d'une solution optimale continue en relaxant les contraintes d'intégrité (problème linéaire).
2. Branchement sur des variables entières qui ne satisfont pas les contraintes.
3. Ajout de coupes pour réduire l'espace de recherche, en utilisant par exemple les contraintes de capacité et de flux.
4. Répétition du processus jusqu'à convergence vers une solution optimale entière.

Pourquoi utiliser CBC et le Branch and Cut pour ce problème ?

Les problèmes de routage de véhicules (VRP) sont connus pour leur complexité combinatoire. Le solveur `CBC`, avec sa méthode *Branch and Cut*, est bien adapté pour ces types de problèmes en raison de :

— **Sa capacité à gérer des contraintes complexes** liées au VRP.

— **L'efficacité de ses coupes** dans la réduction de l'espace de recherche.

— **La possibilité de fournir des solutions optimales ou quasi-optimales** dans un temps raisonnable.

En conclusion, l'utilisation de `CBC` comme solveur principal, en s'appuyant sur la méthode *Branch and Cut*, nous a permis de résoudre efficacement notre problème de routage de véhicules avec livraisons fractionnées.

3.2.3 Modélisation algorithmique

Pour résoudre le SD-VRP (Split Delivery Vehicle Routing Problem), nous avons procédé comme suit :

Étape 1 : Calcul du nombre minimal de véhicules nécessaires

Avant d'optimiser la distance totale parcourue, nous déterminons le nombre minimal de véhicules nécessaires pour satisfaire la demande des clients. Ce calcul repose sur la formule suivante : $\text{M\_min} = \left\lfloor \frac{\sum_{i=1}^{n} c_i}{Q} \right\rfloor$

où ci représente la demande du client i et Q est la capacité d'un véhicule.

Listing 3.2 – Calcul du nombre minimal de véhicules nécessaires

```
def calculate_min_vehicles(ci, Q):
    total_demand = sum(ci) # Somme totale des demandes des clients
    min_vehicles = (total_demand + Q- 1) // Q # Calcul arrondi
        vers le haut
    return min_vehicles
```

Étape 2 : Définition du problème avec PuLP

Nous formulons le problème comme suit :

$$\text{Minimiser} \quad \sum_{k=1}^{M} \sum_{i=0}^{n} \sum_{j=0}^{n} d_{ij} x_{ijk}$$

où d_{ij} est la distance entre les nœuds i et j, x_{ijk} est une variable binaire indiquant si le véhicule k parcourt l'arc (i, j), et M est le nombre de véhicules.

Listing 3.3 – Définition du problème avec PuLP

```
# Cr er le probl me
prob = pulp.LpProblem("SD-VRP", pulp.LpMinimize)
```

Étape 3 : Déclaration des variables de décision

Les variables de décision sont définies comme suit :

$$x_{ijk} = \begin{cases} 1 & \text{si le véhicule } k \text{ parcourt l'arc } (i, j) \\ 0 & \text{sinon} \end{cases}$$

et $y_{ik} \geq 0$ (quantité livrée par le véhicule k au client i).

Listing 3.4 – Déclaration des variables de décision

```
# Variables de d cision
x = pulp.LpVariable.dicts("x", ((i, j, k) for i in range(n+1) for j
    in range(n+1) for k in range(1, M+1)), cat='Binary')
y = pulp.LpVariable.dicts("y", ((i, k) for i in range(1, n+1) for k
    in range(1, M+1)), lowBound=0)
```

Étape 4 : Définition de la fonction objectif

$$\text{Minimiser} \quad \sum_{k=1}^{M} \sum_{i=0}^{n} \sum_{j=0}^{n} d_{ij} x_{ijk}$$

Listing 3.5 – Définition de la fonction objectif

```
# Objectif : minimiser la somme des distances parcourues
prob += pulp.lpSum(distance_matrix[i][j] * x[i, j, k] for i in
   range(n+1) for j in range(n+1) for k in range(1, M+1))
```

Étape 5 : Ajout des contraintes

Satisfaction de la demande

$$\sum_{k=1}^{M} y_{ik} = c_i, \quad \forall i \in \{1, \ldots, n\}$$

Listing 3.6 – Contraintes de satisfaction de la demande

```
for i in range(1, n+1):
    prob += pulp.lpSum(y[i, k] for k in range(1, M+1)) == ci[i-1]
```

Capacité des véhicules

$$\sum_{i=1}^{n} y_{ik} \leq Q, \quad \forall k \in \{1, \ldots, M\}$$

Listing 3.7 – Contraintes de capacité des véhicules

```
for k in range(1, M+1):
    prob += pulp.lpSum(y[i, k] for i in range(1, n+1)) <= Q
```

Conservation du flux

$$\sum_{j=0}^{n} x_{ijk} = \sum_{j=0}^{n} x_{jik}, \quad \forall i \in \{1, \ldots, n\}, \forall k$$

Listing 3.8 – Contraintes de conservation du flux

```
for i in range(1, n+1):
    for k in range(1, M+1):
        prob += pulp.lpSum(x[i, j, k] for j in range(n+1)) == pulp.
            lpSum(x[j, i, k] for j in range(n+1))
```

Flux depuis et vers le dépôt

$$\sum_{j=1}^{n} x_{0jk} = 1 \quad \text{et} \quad \sum_{j=1}^{n} x_{j0k} = 1, \quad \forall k \in \{1, \ldots, M\}$$

Listing 3.9 – Contraintes de flux depuis et vers le dépôt

```
for k in range(1, M+1):
    prob += pulp.lpSum(x[0, j, k] for j in range(1, n+1)) == 1
    prob += pulp.lpSum(x[j, 0, k] for j in range(1, n+1)) == 1
```

Compatibilité des trajets et livraisons

$$y_{ik} \leq Q \cdot \sum_{j=0}^{n} x_{ijk}, \quad \forall i \in \{1, \ldots, n\}, \forall k$$

Listing 3.10 – Contraintes de compatibilité des trajets et livraisons

```
for i in range(1, n+1):
    for k in range(1, M+1):
        prob += y[i, k] <= Q * pulp.lpSum(x[i, j, k] for j in range
            (n+1))
```

Étape 6 : Résolution et analyse

Le problème est résolu à l'aide du solveur :

Listing 3.11 – Résolution du problème avec PuLP

```
solver = pulp.PULP_CBC_CMD(timeLimit=900)
prob.solve(solver)

if pulp.LpStatus[prob.status] == 'Optimal':
    total_cost = pulp.value(prob.objective)
    print(f"Cost total minimal : {total_cost}")
```

```
Nombre minimal de véhicules nécessaires (M) : 6
Coût total minimal: 12009.0
Route 1: 0 -> 4 (50.0) -> 7 (50.0) -> 0
Route 2: 0 -> 1 (50.0) -> 8 (50.0) -> 0
Route 3: 0 -> 4 (40.0) -> 5 (60.0) -> 0
Route 4: 0 -> 1 (10.0) -> 6 (90.0) -> 0
Route 5: 0 -> 2 (90.0) -> 7 (10.0) -> 0
Route 6: 0 -> 3 (60.0) -> 8 (40.0) -> 0
Total cost: 12009.0
Number of deliveries: 12
Trucks loads: 100.0 100.0 100.0 100.0 100.0 100.0
```

FIGURE 3.2 – Exemple de solution : cas 1

3.3 Implémentation d'une méta-heuristique pour résoudre le problème

3.3.1 Méthodologie

Recherche Tabou

La recherche tabou est une méta-heuristique qui améliore une solution initiale en explorant son voisinage, tout en empêchant les retours en arrière grâce à une liste taboue. Cela aide à éviter les minima locaux.

Recherche Clarke et Wright

Cette méthode gloutonne commence avec une tournée individuelle pour chaque client. Elle fusionne les tournées de manière itérative en maximisant les économies de distance, sous contrainte de capacité.

limitations

La méthode Clarke et Wright ne prend pas en charge les livraisons fractionnées (split delivery), car elle suppose que chaque client est servi par un seul véhicule et qu'une demande entière est satisfaite en une seule tournée. Cela limite la flexibilité dans la création de routes optimales pour des problèmes comme le Split Delivery Vehicle Routing Problem (SDVRP), où les demandes peuvent être partagées entre plusieurs véhicules pour réduire les coûts totaux.

Variable Neighborhood search (VNS)

Son principe repose sur la recherche dans des espaces de solutions multiples (voisinages) en alternant entre des phases de perturbation et de recherche locale. Cette alternance permet d'explorer l'espace des solutions de manière diversifiée tout en exploitant les optima locaux dans chaque voisinage.

1. On génére une solution initiale en fractionnant les livraisons en fonction de la capacité des véhicules et des demandes des clients.
2. On met à jour la solution actuelle, càd on améliore les routes ou bien on en fractionnant les livraisons. Comme ça on explore le voisinage suivant.
3. Après avoir atteint un optimum local, une nouvelle solution initiale est générée pour explorer d'autres régions de l'espace des solutions.
4. si aucune amélioration significative n'est trouvée l'algorithme s'arrête.

Le VNS combine exploration globale et exploitation locale, ce qui assure la diversification des solutions tout en convergeant vers des solutions optimales ou quasi-optimales.

Recherche locale

Pour une solution(route) $r \in \mathcal{R}$, son voisinage est noté $N_k(R)$, où k désigne le niveau de voisinage. Une recherche locale explore $N_k(R)$ pour trouver une meilleure solution :

$$R' = \arg \min_{x \in N_k(R)} f(x).$$

Si $f(R') < f(R)$, alors R est remplacée par R'.

Fonction de coût

Dans le cadre du SDVRP (Split Delivery Vehicle Routing Problem), la fonction de coût total $f(R)$ est définie d'une manière plus explicite pour la fonction définie précédemment) comme suit :

$$f(R) = \sum_{\text{route } r \in R} \left(\text{Coût}_{\text{dépôt} \to \text{premier client}} + \sum_{i \in r} \text{Coût}_{i \to i+1} + \text{Coût}_{\text{dernier client} \to \text{dépôt}} \right),$$

avec dépôt toujours de coordonnées (0,0).

Optimisation multi-voisinage

L'alternance entre voisinages permet au VNS de surmonter les limitations des optima locaux et de combiner efficacement :

- **Exploration globale :** Diversification des solutions en explorant différents voisinages.
- **Exploitation locale :** Amélioration des solutions dans un voisinage donné.

Nous avons trouvé que le VNS assure une exploration systématique et flexible de l'espace des solutions ce qui le rend adapté au problème combinatoire de SDVRP.

3.3.2 Analyse des résultats

Premieres approches

L'implémentation de méta-heuristique **Tabou** et **Clarke et Wright** a résolu le problème décrit, sauf que les solutions n'étaient pas à la hauteur de nos attentes ni au critère évolutif pour le problème de tournées de véhicules à capacité limitée (CVRP) basé sur le partage des livraisons en plusieurs routes. En effet, il tend, dans la plupart des cas, à servir les clients chacun à part. Par conséquent, l'opération était coûteuse, càd consomme des distances inutiles.

exemples :

pour **cas 0**
Route 1 : 0 – 1 (4) – 2 (5) – 0
Route 2 : 0 – 3 (6) – 0
Total cost : 31
Number of deliveries : 3
Loads : 9 6
= Aucun problème pour ce cas puisque la solution coïncide avec la solution optimale.

pour **cas 1**
Route 1 : 0 – 1 (60) – 0
Route 2 : 0 – 2 (90) – 0
Route 3 : 0 – 3 (60) – 0
Route 4 : 0 – 4 (90) – 0
Route 5 : 0 – 5 (60) – 0
Route 6 : 0 – 6 (90) – 0
Route 7 : 0 – 7 (60) – 0
Route 8 : 0 – 8 (90) – 0
Total cost : 24000
Number of deliveries : 8
Loads : 60 90 60 90 60 90 60 90
= Chaque route sert un et un seul client.

Outils nécessaires

— Des fonctions autorisant le fractionnement des livraisons.

— Prendre en considération le fractionnement dans la répartition des coûts entre les différentes tournées.

Approche VNS Variable Neighborhood Search

Initialisation

créer une solution réalisable qui respecte les contraintes du problème, tout en permettant les livraisons fractionnées pour optimiser l'utilisation des véhicules.

1. Nous avons commencé par l'initialisation de structures, notamment une liste vide pour stocker les tournées, un tableau contenant les demandes restantes de chaque client, et des variables pour suivre la charge actuelle du véhicule et la tournée en cours.
   generate_split_initial_solution(nb_customers, demands, capacity)
2. On parcoure séquentiellement les clients et on livre pour chacun une quantité calculée comme étant la plus petite valeur entre la demande restante et la capacité restante du véhicule.
   calculate_split_solution_cost(routes, distance_matrix, distance_depots)
3. On met à jour la tournée actuelle et la charge du véhicule
4. Lorsque la capacité maximale du véhicule est atteinte ou que tous les clients ont été visités dans une itération, la tournée est ajoutée à la liste des tournées, le véhicule retourne au dépôt, et on commence une nouvelle tournée.
5. répétition jusqu'à satisfaire les demandes.

Listing 3.12 – Code de prétraitement des données

```
def generate_split_initial_solution(nb_customers, demands, capacity):
    routes = []
    remaining_demands = demands[:]
    while any(remaining_demands):
        route = []
        current_load = 0
        for customer in range(nb_customers):
            if remaining_demands[customer] > 0:
                deliverable = min(remaining_demands[customer], capacity - current_load)
                if deliverable > 0:
                    route.append((customer, deliverable))
                    remaining_demands[customer] -= deliverable
                    current_load += deliverable
                    if current_load == capacity:
                        break
        routes.append(route)
    return routes

def calculate_split_solution_cost(routes, distance_matrix, distance_depots):
    total_cost = 0
    for route in routes:
        if not route:
            continue
        total_cost += distance_depots[route[0][0]]
        for i in range(len(route) - 1):
            total_cost += distance_matrix[route[i][0]][route[i + 1][0]]
        total_cost += distance_depots[route[-1][0]]
    return total_cost

def generate_neighborhood(routes, nb_customers, demands, capacity):
    neighbors = []
    for i, route in enumerate(routes):
        for j, (customer, quantity) in enumerate(route):
            if quantity > 1:
                new_routes = [list(r) for r in routes]
                split_quantity = quantity // 2
                new_routes[i][j] = (customer, quantity - split_quantity)
                for k, other_route in enumerate(new_routes):
                    if k != i and sum(q for _, q in other_route) + split_quantity <= capacity:
                        new_routes[k].append((customer, split_quantity))
                        neighbors.append(new_routes)
                        break
    return neighbors

def local_search(routes, distance_matrix, distance_depots, demands,
```

```
    capacity):
46      current_solution = routes
47      best_solution = routes
48      best_cost = calculate_split_solution_cost(routes,
            distance_matrix, distance_depots)
49
50      improving = True
51      while improving:
52          improving = False
53          neighborhood = generate_neighborhood(current_solution, len(
                demands), demands, capacity)
54          for neighbor in neighborhood:
55              cost = calculate_split_solution_cost(neighbor,
                    distance_matrix, distance_depots)
56              if cost < best_cost:
57                  best_solution = neighbor
58                  best_cost = cost
59                  improving = True
60          current_solution = best_solution
61
62      return best_solution
63
64  def variable_neighborhood_search(nb_customers, demands, capacity,
        distance_matrix, distance_depots, max_iterations=100):
65      current_solution = generate_split_initial_solution(nb_customers
            , demands, capacity)
66      best_solution = current_solution
67      best_cost = calculate_split_solution_cost(current_solution,
            distance_matrix, distance_depots)
68
69      for iteration in range(max_iterations):
70          local_optimum = local_search(current_solution,
                distance_matrix, distance_depots, demands, capacity)
71          local_cost = calculate_split_solution_cost(local_optimum,
                distance_matrix, distance_depots)
72
73          if local_cost < best_cost:
74              best_solution = local_optimum
75              best_cost = local_cost
76
77          current_solution = generate_split_initial_solution(
                nb_customers, demands, capacity)
78
79      return best_solution, best_cost
```

Description des fonctions

- **generate_split_initial_solution(nb_customers, demands, capacity)** génère une solution initiale réalisable pour le problème
- **calculate_split_solution_cost(routes, distance_matrix, distance_depots)** S'assure d'évaluer avec précision les coûts associés aux routes, y compris ceux liés aux livraisons fractionnées
- **generate_neighborhood(routes, nb_customers, demands, capacity)** Effectue des opérations comme déplacer une partie ou la totalité de la demande d'un client vers une autre route, échanger des livraisons entre clients ou réorganiser les routes, tout en respectant les contraintes de capacité

Interprétations des résultats

cas	cas 0	cas 1	cas 25
coût total	37	30726	1145
Nombre des livraisons	4	12	36
Charges	10 5	100 100 100 100 100 100 100	7800 8000 8000 5570

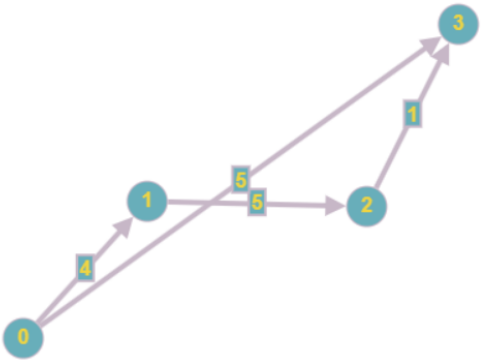


FIGURE 3.3 – Graphe de l'algorithme VNS appliqué au cas 0

```
Route 1: 0 - 1 (4) - 2 (5) - 3 (1) - 0
Route 2: 0 - 3 (5) - 0
Total cost: 37
Number of deliveries: 4
Loads: 10 5
```

FIGURE 3.4 – solution cas 0

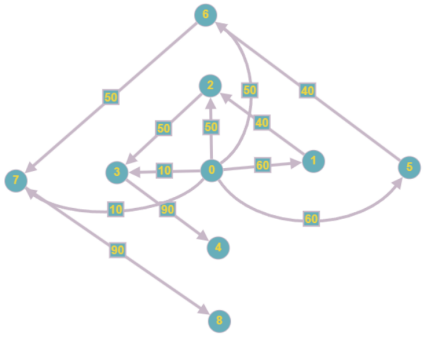


FIGURE 3.3 – Graphe de l'algorithme VNS appliqué au cas 1

NB : les noeuds (2 jusqu'à 8) reviennent à 0

```
Route 1: 0 - 1 (60) - 2 (40) - 0
Route 2: 0 - 2 (50) - 3 (50) - 0
Route 3: 0 - 3 (10) - 4 (90) - 0
Route 4: 0 - 5 (60) - 6 (40) - 0
Route 5: 0 - 6 (50) - 7 (50) - 0
Route 6: 0 - 7 (10) - 8 (90) - 0
Total cost: 30726
Number of deliveries: 12
Loads: 100 100 100 100 100 100
```

FIGURE 3.4 – solution cas 1

Explication de la solution "Case0"

— Les deux premieres lignes representent chacune l'itinéraire de point de depôt vers les clients trouvés dans la solution. optimale.
Disons un camion qui va partir de 0 (départ) chargé avec 10 unités (articles). Il va commencer à livrer 4 unités au client 1 et à partir de cette position il poursuit vers le client 2 pour livrer 5 unités puis livre finalement le reste de son charge (10-4-5=1) au client 3.
Le client 3 n'est pas entièrement satisfait (il reste 5).Un deuxième camion va sortir de 0 et le livre cette quantité.

— Le coût total est la somme des distances parcourues de depôt vers le premier client, entre les clients et vers le depôt.(Distance définie dans la partie de définition)

— nombre des livraisons =4 , est tout simplement le nombre des clients servis **par touts les camions !**.

Cas des grandes instances

pour le cas 32

Résultats

Résultat obtenu pour le cas 0

Données : Le cas 0 est défini par les données suivantes :

- **Nombre de clients :** 3.
- **Capacité des véhicules :** 10 unités.
- **Demande des clients :** $[4, 5, 6]$.
- **Coordonnées des nœuds :**
  - Dépôt : $(0, 0)$.
  - Clients : $(2, 3), (4, 5), (6, 7)$.

Résultats du solveur : En appliquant le solveur **PULP-CBC** à ces données, les résultats suivants ont été obtenus :

- **Coût total :** 31 unités de distance.
- **Routes :**
  - Route 1 : $0 \to 1\,(4.0) \to 3\,(6.0) \to 0$.
  - Route 2 : $0 \to 2\,(5.0) \to 0$.
- **Nombre total de livraisons :** 3.
- **Charges des camions :** 10 unités pour le camion 1, 5 unités pour le camion 2.

Discussion :

- La solution est réalisable et respecte toutes les contraintes :
  - Chaque client reçoit sa demande complète.
  - Les capacités des véhicules ne sont pas dépassées.
  - Le flux entrant et sortant est équilibré.
- Le solveur utilise un algorithme exact (*Branch and Cut*), garantissant une solution optimale pour ce cas.

Résultats pour les autres cas

Problèmes rencontrés : Pour les autres instances, le solveur n'a pas pu produire de solution réalisable dans les limites de temps allouées (1000 secondes). Cela est dû aux facteurs suivants :

— **Complexité combinatoire :** Le problème devient rapidement intraitable à mesure que le nombre de clients et de véhicules augmente.

— **Limites du solveur :** Bien que performant pour les petites instances, **PULP-CBC** peine à gérer les grandes instances. Les approches exactes nécessitent des temps de calcul exponentiels.

— **Structure des données :** Certaines instances présentent des demandes ou distances créant des contraintes difficiles à satisfaire.

Impact des limitations : L'incapacité du solveur à trouver une solution réalisable pour ces cas met en évidence la nécessité de méthodes alternatives, telles que :

— **Heuristiques :** Ces méthodes fournissent des solutions rapides et souvent acceptables en sacrifiant l'exactitude.

— **Méta-heuristiques :** Des algorithmes comme l'algorithme génétique ou la colonie de fourmis peuvent explorer efficacement l'espace des solutions.

— **Solveurs avancés :** Des outils commerciaux comme **Gurobi** ou **CPLEX** offrent des performances supérieures sur les grandes instances.

Recommandations :

— Développer et tester des approches hybrides combinant un solveur exact pour des sous-problèmes et des heuristiques pour gérer la complexité globale.

— Explorer l'utilisation de méta-heuristiques pour les instances moyennes et grandes, où elles surpassent les méthodes exactes en termes de temps d'exécution.

— Améliorer la gestion des contraintes dans le modèle, par exemple en utilisant des techniques de relaxation ou de reformulation.

Conclusion

Le cas 0 illustre le succès des méthodes exactes pour les petites instances, mais les échecs sur les autres cas soulignent les limites des solveurs comme **PULP-CBC**. Les méthodes heuristiques et méta-heuristiques apparaissent comme des solutions prometteuses pour les instances de taille moyenne et grande.

Nos avis

5.1 ACHBANI Ismail

Travailler sur ce projet m'a permis d'approfondir ma compréhension de l'optimisation combinatoire et de ses applications pratiques dans des problèmes logistiques complexes. La mise en œuvre des approches exactes et heuristiques a été un véritable défi technique, révélant les forces et les limites de chaque méthode. Cette expérience a renforcé mon intérêt pour la résolution de problèmes industriels réels en utilisant des techniques avancées.

5.2 HAMOUCH Ayoub

J'ai particulièrement apprécié le processus d'analyse des performances des différentes approches pour le Split Delivery Vehicle Routing Problem. Cela m'a permis de mieux comprendre les subtilités des heuristiques et des méta-heuristiques dans des contextes pratiques. Ce projet m'a aidé à développer mes compétences analytiques et à mieux saisir l'importance de l'optimisation dans les systèmes logistiques modernes.

5.3 KHALFA Youssef

Ce projet a été une opportunité enrichissante pour appliquer des concepts théoriques à un problème réel. La combinaison des solveurs exacts et des heuristiques m'a offert une perspective globale sur les méthodes d'optimisation combinatoire. J'ai particulièrement apprécié le travail en équipe, qui m'a permis de consolider mes compétences en programmation et en analyse mathématique.

5.4 YAJJOU Rayane

Participer à ce projet m'a permis d'acquérir une vision plus approfondie des défis logistiques auxquels les entreprises sont confrontées. L'utilisation des algorithmes développés, ainsi que l'interprétation des résultats, a été une expérience formatrice. Ce travail a renforcé mon aptitude à résoudre des problèmes complexes tout en développant une approche structurée et analytique.

Conclusion

Ce projet a permis d'explorer en profondeur le problème de tournée des véhicules avec livraisons fractionnées (SD-VRP) et d'appliquer des approches combinatoires pour sa résolution. Bien que les résultats soient satisfaisants, plusieurs limitations et pistes d'amélioration ont été identifiées. D'une part, l'optimisation du nombre de véhicules et la gestion des contraintes de capacité nécessitent une flexibilité accrue pour des scénarios plus réalistes. D'autre part, les performances des solveurs, comme PuLP_CBC, sont limitées face à des instances complexes, soulignant l'intérêt d'explorer des solveurs avancés (Gurobi, CPLEX) et des infrastructures matérielles plus performantes (GPU ou HPC). L'intégration de contraintes plus réalistes, telles que les fenêtres temporelles, et l'amélioration des algorithmes métaheuristiques, notamment les algorithmes génétiques et VNS(Variable Neighborhood Search), constituent également des perspectives prometteuses.

Sur le plan des enseignements, ce projet a permis de mieux comprendre les défis des problèmes logistiques complexes, notamment les spécificités du SD-VRP qui augmentent la flexibilité et la complexité par rapport au CVRP. En comparant le solveur exact et les algorithmes métaheuristiques, nous avons observé des différences marquées en termes de rapidité et d'efficacité selon la taille des instances. Le solveur s'est révélé plus adapté aux petites instances, tandis que l'algorithme génétique a montré sa pertinence pour des problèmes plus grands.

Enfin, ce projet a renforcé des compétences clés, notamment en programmation, en configuration de solveurs, et en application d'algorithmes d'optimisation sur des données réelles. Il a également offert une expérience pratique dans la gestion de projets d'optimisation, incluant la production et l'analyse de solutions, ainsi que la comparaison critique des outils disponibles. Ces acquis offrent des perspectives intéressantes pour des travaux futurs visant à étendre l'applicabilité du modèle à des contextes industriels plus complexes et à améliorer davantage son efficacité.