1 Résolution détaillée du cas 0 avec le solveur

1.1 Implémentation du code

Dans cette partie, on a choisi CBC (Coin-or branch and cut) comme solveur de résolution vu qu'il est open-source et permet de resoudre les problèmes de programmation entière mixte (MILP) comme dans le sujet de ce projet . Commençant par l'importation de la bibliothèque Pulp .

```
3 10
4 5 6
0 0
2 3
4 5
6 7
```

Figure 1 – fichier instance du cas 0

```
!pip install pulp
```

Avant de commencer la résolution du problème d'optimisation pour le cas 0, il est essentiel de charger les données à partir du fichier d'entrée.Celui ci contient des informations clés sur les clients, la capacité des véhicules, ainsi que les coordonnées géographiques des nœuds (clients et dépôt) afin de comprendre la structure du fichier instances pour le traiter par la suite .

```
from google.colab import files

# Fonction pour t l verser et charger un fichier d'entr e
def load_instance():
    print("Veuillez t l verser votre fichier d'entr e (exemple : Case0.txt) :")
    uploaded = files.upload()  # Cette commande ouvre une fen tre pour t l verser un fichier

    if not uploaded:
        print("Aucun fichier n'a   t   t l vers . R essayez.")
        return None

```

```
    filename = list(uploaded.keys())[0]  # R cup re le nom du fichier t l vers

    # Lecture des donn es du fichier
    try:
        with open(filename, 'r') as file:
            lines = file.readlines()
            n, Q = map(int, lines[0].split())  # Nombre de clients et capacit  maximale des v hicules
            demands = list(map(int, lines[1].split()))  # Demandes des clients
            coords = [tuple(map(int, line.split())) for line in lines[2:]]  # Coordonn es des n uds
        print("Fichier t l vers  avec succ s.")
        return n, Q, demands, coords
    except Exception as e:
        print(f"Erreur lors de la lecture du fichier : {e}")
        return None

# Tester la fonction
result = load_instance()
if result:
    n, Q, demands, coords = result
    print("Nombre de clients :", n)
    print("Capacit  des v hicules :", Q)
    print("Demandes des clients :", demands)
    print("Coordonn es des n uds :", coords)
else:
    print("Impossible de charger les donn es.")
```

Choisir des fichiers Case0.txt

- **Case0.txt**(text/plain) - 31 bytes, last modified: 21/12/2024 - 100% done

```
Saving Case0.txt to Case0.txt
Fichier téléversé avec succès.
Nombre de clients : 3
Capacité des véhicules : 10
Demandes des clients : [4, 5, 6]
Coordonnées des nœuds : [(0, 0), (2, 3), (4, 5), (6, 7)]
```

Figure 2 – Résultats du code

Puis,on implémente une modélisation détaillée du SD-VRP en respectant les contraintes du problème. Les résultats incluent les routes optimales, les

charges des véhicules, et le coût total.

Calcul des distances euclidiennes La fonction `euclidean_distance(i, j)` calcule la distance euclidienne arrondie entre deux nœuds i et j, ce qui est utilisé dans la fonction objectif pour minimiser les coûts.

```
import math

def euclidean_distance(i, j):
    dx = coords[j][0] - coords[i][0]
    dy = coords[j][1] - coords[i][1]
    return int(math.sqrt(dx**2 + dy**2) + 0.5)
```

Nombre maximal de véhicules requis

Le nombre maximal de véhicules requis est calculé comme la somme totale des demandes des clients divisée par la capacité d'un véhicule, arrondie à l'entier supérieur.

```
max_vehicles = math.ceil(sum(demands) / Q)
```

Création du modèle d'optimisation

Le modèle d'optimisation linéaire a été créé avec `PuLP`, C'est un probelemedproblème e minimisation vu qu'on souhaite trouver les trajets optimaux afin de minimiser les coûts liés aux distances .

```
import pulp

model = pulp.LpProblem("SD-VRP", pulp.LpMinimize)
```

Variables de décision Deux types de variables de décision sont définies :

- `x[i, j, k]` : Indique si le véhicule k effectue le trajet de i à j.
- `y[i, k]` : Quantité livrée par le véhicule k au client i.

```
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,
   max_vehicles + 4)),
                          cat='Binary')

y = pulp.LpVariable.dicts("y",
                          ((i, k) for i in range(1, n + 1)
```

```
                                        for k in range(1,
    max_vehicles + 4)),
                            lowBound=0)
```

Fonction objectif

La fonction objectif cherche à minimiser la somme des distances parcourues par tous les véhicules. Elle multiplie la distance entre i et j par la variable `x[i, j, k]`.

```
model += pulp.lpSum(euclidean_distance(i, j) * x[i, j, k]
                    for i in range(n + 1) for j in range(n +
    1) if i != j
                    for k in range(1, max_vehicles + 4))
```

Les contraintes :

1. Chaque client doit recevoir exactement sa demande totale, en depit du nombre de véhicules qui livrent.

```
for i in range(1, n + 1):
    model += pulp.lpSum(y[i, k] for k in range(1,
    max_vehicles + 4)) == demands[i - 1]
```

2. La quantité totale livrée par un véhicule ne doit pas dépasser sa capacité Q.

```
for k in range(1, max_vehicles + 4):
    model += pulp.lpSum(y[i, k] for i in range(1, n + 1))
    <= Q
```

3. Le nombre de trajets entrants dans un nœud doit être égal au nombre de trajets sortants .

```
for k in range(1, max_vehicles + 4):
    for i in range(n + 1):
        model += pulp.lpSum(x[i, j, k] for j in range(n +
    1) if i != j) == pulp.lpSum(x[j, i, k] for j in range
    (n + 1) if i != j)
```

4. Chaque véhicule commence sa tournée au dépôt, avec au plus un départ par véhicule.

```
for k in range(1, max_vehicles + 4):
    model += pulp.lpSum(x[0, j, k] for j in range(1, n +
    1)) <= 1
```

Résolution du modèle

Ensuite on exécute le code afin de afficher les résultats

```
status = model.solve(pulp.PULP_CBC_CMD(msg=True, timeLimit=1800, threads=4))
```

Affichage de la solution

```
if pulp.LpStatus[status] == 'Optimal':
    routes = []
    truck_loads = []

    for k in range(1, max_vehicles + 4):
        route = [0]
        load = 0
        delivered = {i: 0 for i in range(1, n + 1)}
        for i in range(1, n + 1):
            if pulp.value(y[i, k]) > 0:
                delivered[i] = pulp.value(y[i, k])
                route.append(i)
                load += delivered[i]
        route.append(0)
        if len(route) > 2:
            routes.append((route, delivered, load))
            truck_loads.append(load)

    recalculated_cost = 0
    for route, _, _ in routes:
        for i in range(len(route) - 1):
            recalculated_cost += euclidean_distance(route[i], route[i + 1])

    print("Routes:")
    for k, (route, delivered, load) in enumerate(routes, start=1):
        route_str = " -> ".join(
            f"{node} ({round(delivered[node], 2)})" if node > 0 else str(node) for node in route
        )
        print(f"Route {k}: {route_str}")

    print(f"Total cost (recalculated): {recalculated_cost}")
    print(f"Number of deliveries: {sum(len(route) - 2 for route, _, _ in routes)}")
    print(f"Truck loads: {' '.join(map(str, truck_loads))}")
else:
```

```
54      print(f"No optimal solution found. Solver status: {pulp.
    LpStatus[status]}")
```

```
Routes:
Route 1: 0 -> 3 (6.0) -> 0
Route 2: 0 -> 1 (4.0) -> 2 (5.0) -> 0
Total cost (recalculated): 31
Number of deliveries: 3
Truck loads: 6.0 9.0
```

Figure 3 – Résultats par solveur

1.2 Vérification à la main des résultats obtenu

Calcul des distances entre les nœuds

On remarque d'abord que tout les contraintes sont respectés dans la solutions obtenue .Pour s'assurer que c'est bien le minimum , commençons par calculer les distances euclidiens entre chaque deux points liées dans les routes .

Distance $d_{0,1}$ (dépôt $\rightarrow$ client 1) :

$$d_{0,1} = \left\lfloor \sqrt{(2-0)^2 + (3-0)^2} + 0.5 \right\rfloor = \left\lfloor \sqrt{4+9} + 0.5 \right\rfloor = \lfloor 3.61 + 0.5 \rfloor = 4$$

Distance $d_{0,2}$ (dépôt $\rightarrow$ client 2) :

$$d_{0,2} = \left\lfloor \sqrt{(4-0)^2 + (5-0)^2} + 0.5 \right\rfloor = \left\lfloor \sqrt{16+25} + 0.5 \right\rfloor = \lfloor 6.4 + 0.5 \rfloor = 6$$

Distance $d_{0,3}$ (dépôt $\rightarrow$ client 3) :

$$d_{0,3} = \left\lfloor \sqrt{(6-0)^2 + (7-0)^2} + 0.5 \right\rfloor = \left\lfloor \sqrt{36+49} + 0.5 \right\rfloor = \lfloor 8.6 + 0.5 \rfloor = 9$$

Distance $d_{1,2}$ (client 1 $\rightarrow$ client 2) :

$$d_{1,2} = \left\lfloor \sqrt{(4-2)^2 + (5-3)^2} + 0.5 \right\rfloor = \left\lfloor \sqrt{4+4} + 0.5 \right\rfloor = \lfloor 2.83 + 0.5 \rfloor = 3$$

Distance $d_{1,3}$ (client 1 $\rightarrow$ client 3) :

$$d_{1,3} = \left\lfloor \sqrt{(6-2)^2 + (7-3)^2} + 0.5 \right\rfloor = \left\lfloor \sqrt{16+16} + 0.5 \right\rfloor = \lfloor 5.66 + 0.5 \rfloor = 6$$

Distance $d_{2,3}$ (client 2 $\rightarrow$ client 3) :

$$d_{2,3} = \left\lfloor \sqrt{(6-4)^2 + (7-5)^2} + 0.5 \right\rfloor = \left\lfloor \sqrt{4+4} + 0.5 \right\rfloor = \lfloor 2.83 + 0.5 \rfloor = 3$$

Vérification des routes optimales

Nous avons obtenu lors de l'implemenattaiondimplementation eux solutions de routes différentes.
Solution 1 :

$$0 \rightarrow 1(4) \rightarrow 2(1) \rightarrow 0$$

$$0 \rightarrow 2(4) \rightarrow 3(6) \rightarrow 0$$

Solution 2 :

$$0 \rightarrow 3(6) \rightarrow 0$$

$$0 \rightarrow 1(4) \rightarrow 2(5) \rightarrow 0$$

Pour la solution 1

Route 1 :

- $d_{0,1} = 4$ (calculé précédemment) - $d_{1,2} = 3$ (calculé précédemment) - $d_{2,0} = 6$ (calculé précédemment)

$$\text{Coût de la Route 1} = d_{0,1} + d_{1,2} + d_{2,0} = 4 + 3 + 6 = 13$$

Route 2 : $0 \to 2 \to 3 \to 0$

- $d_{0,2} = 6$ (calculé précédemment) - $d_{2,3} = 3$ (calculé précédemment) - $d_{3,0} = 9$ (calculé précédemment)

$$\text{Coût de la Route 2} = d_{0,2} + d_{2,3} + d_{3,0} = 6 + 3 + 9 = 18$$

Calcul du coût total

$$\text{Coût total} = \text{Coût Route 1} + \text{Coût Route 2} = 13 + 18 = 31$$

Pour la solution 2

Coût de la Route 1 : $0 \to 1 \to 2 \to 0$

$$d_{0,1} = 4, \quad d_{1,2} = 3, \quad d_{2,0} = 6$$

$$\text{Total (Route 1)} = d_{0,1} + d_{1,2} + d_{2,0} = 4 + 3 + 6 = 13$$

Coût de la Route 2 : $0 \to 3 \to 0$

$$d_{0,3} = 9, \quad d_{3,0} = 9$$

$$\text{Total (Route 2)} = d_{0,3} + d_{3,0} = 9 + 9 = 18$$

Coût total :

$$\text{Coût total} = \text{Coût Route 1} + \text{Coût Route 2} = 13 + 18 = 31$$

On remarque que les coûts totales sont égaux aux coûts totales trouvées par le solveur , ce qui prouve qu'on peut avoir différents chemin possibles mais avec le même nombres de routes minimales .

2 Résolution détaillée du cas 0 avec l'heuristique et la méta-heuristique

2.1 Principe de l'heuristique de Solomon

La méthode d'insertion séquentielle de Solomon est une technique de construction employée pour résoudre le problème de routage des véhicules . Elle opère en élaborant les voies successivement, en introduisant progressivement les clients conformément à un processus fondé sur des critères précisément déterminés. Une fois la route initialisée, les autres clients non visités sont insérés dans la route existante en tenant compte de leur meilleure position d'insertion. Cette position est déterminée à l'aide de deux critères principaux.

Le premier critère, noté $c_1(i, u, j)$, évalue le coût d'insertion d'un client u entre deux clients i et j dans une route partiellement construite. Ce coût combine deux composantes :

$$c_1(i, u, j) = \alpha_1 c_{11}(i, u, j) + \alpha_2 c_{12}(i, u, j)$$

où :

- $c_{11}(i, u, j)$ représente le coût additionnel lié au parcours entre i, u, et j,
- $c_{12}(i, u, j)$ mesure l'impact sur les contraintes de la route, comme la demande.

Les coefficients α_1 et α_2 pondèrent ces deux composantes et ajustent leur importance relative.

Le deuxième critère, noté $c_2(i, u, j)$, mesure la balance entre le coût d'insertion du client u et le coût de création d'une nouvelle route. Il est calculé comme :

$$c_2(i, u, j) = \lambda t_{0u} - c_1(i, u, j)$$

où t_{0u} est la distance entre le dépôt et u, et λ ajuste l'importance de cette distance.

Le client u^* ayant la meilleure valeur pour le critère $c_2(i, u, j)$ est sélectionné et inséré à la position optimale dans la route. Si aucun client restant ne peut être inséré dans une route existante, une nouvelle route est initialisée avec un client non visité, et le processus est répété.

Cette approche permet de construire progressivement une solution au VRP, en minimisant les coûts d'insertion et en respectant les contraintes de

capacité des véhicules. L'heuristique est reconnue pour sa simplicité et sa capacité à générer rapidement une solution initiale. Cependant, la qualité des résultats dépend fortement des critères d'initialisation et d'insertion, et elle peut produire des solutions sous-optimales si elle n'est pas suivie d'une phase d'amélioration [**b**]

2.2 Code

La première fonction du code, `load_instance`, est responsable de la lecture et de l'extraction des données nécessaires depuis un fichier d'instance. Cette fonction ouvre le fichier spécifié par le chemin `file_path` et lit toutes ses lignes. La première ligne du fichier contient deux valeurs entières : le nombre de clients (`num_clients`) et la capacité des véhicules (`vehicle_capacity`). La deuxième ligne liste les demandes de chaque client, stockées dans la liste `demands`. Les lignes suivantes contiennent les coordonnées des nœuds, incluant le dépôt et les clients. Chaque ligne de coordonnées est analysée pour extraire les valeurs `x` et `y`, qui sont ensuite ajoutées sous forme de tuples à la liste `coordinates`. Les lignes vides ou mal formatées sont ignorées, garantissant ainsi que seules les données correctes sont traitées. Enfin, la fonction retourne le nombre de clients, la capacité des véhicules, les demandes des clients et les coordonnées des nœuds.

```
def load_instance(file_path):
    with open(file_path, 'r') as f:
        lines = f.readlines()

    # Premi re ligne : Nombre de clients et capacit  des v hicules
    num_clients, vehicle_capacity = map(int, lines[0].split())

    # Deuxi me ligne : Demandes des clients
    demands = list(map(int, lines[1].split()))

    # Coordonn es des n uds (d p t + clients)
    coordinates = []
    for line in lines[2:]:
        if line.strip():  # Ignorer les lignes vides
            parts = line.split()
            if len(parts) == 2:  # V rifie que la ligne contient exactement deux valeurs
                x, y = map(int, parts)
```

```
                coordinates.append((x, y))
            else:
                print(f"Ligne ignor e ou mal format e : {line}")

    return num_clients, vehicle_capacity, demands, coordinates
```

Listing 1 – Fonction `load_instance`

La fonction `calculate_distance_matrix` calcule une matrice des distances euclidiennes entre tous les nœuds, y compris le dépôt et les clients. Elle commence par déterminer le nombre total de nœuds (`num_nodes`) à partir de la liste des coordonnées. Une matrice carrée de taille `num_nodes x num_nodes` est initialisée avec des zéros. Ensuite, pour chaque paire de nœuds (i, j), si les indices i et j sont différents, la distance euclidienne est calculée en utilisant la formule :

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

Cette distance est ensuite arrondie à l'entier le plus proche en ajoutant 0.5 avant de la convertir en entier, conformément aux spécifications. La distance calculée est stockée dans la matrice à la position `[i][j]`. Une fois toutes les distances calculées, la matrice est renvoyée, fournissant ainsi une représentation complète des distances entre chaque paire de nœuds.

```
def calculate_distance_matrix(coordinates):
    num_nodes = len(coordinates)
    distance_matrix = np.zeros((num_nodes, num_nodes), dtype=int)

    for i in range(num_nodes):
        for j in range(num_nodes):
            if i != j:
                distance = np.sqrt((coordinates[j][0] - coordinates[i][0])**2 +
                                   (coordinates[j][1] - coordinates[i][1])**2)
                distance_matrix[i][j] = int(distance + 0.5)
    # Arrondi comme sp cifi

    return distance_matrix
```

Listing 2 – Fonction `calculate_distance_matrix`

La fonction `sequential_insertion` implémente l'heuristique d'insertion séquentielle de Solomon pour construire les routes de livraison. Elle commence par créer une liste des clients non visités, initialisée avec tous les clients (indices de 1 à `num_clients`). Une liste vide `routes` est également initialisée pour stocker les différentes routes générées. Tant qu'il reste des clients non visités, une nouvelle route est commencée avec le dépôt (0) et la capacité actuelle du véhicule est initialisée à zéro. Pour chaque route, la fonction cherche le meilleur client à insérer en évaluant le coût d'insertion de chaque client non visité dans chaque position possible de la route actuelle. Le coût d'insertion est calculé en tenant compte de la distance supplémentaire ajoutée par l'insertion du client dans la route. Si un client peut être inséré sans dépasser la capacité du véhicule, il est inséré à la position qui minimise ce coût. Une fois qu'aucun autre client ne peut être inséré dans la route actuelle (soit parce que tous les clients restants dépassent la capacité du véhicule, soit parce qu'ils ne peuvent pas être insérés sans augmenter le coût de manière significative), la route est terminée en revenant au dépôt et ajoutée à la liste des routes. Le processus se répète jusqu'à ce que tous les clients soient visités, produisant ainsi une liste complète de routes optimales selon l'heuristique d'insertion séquentielle.

```
1  def sequential_insertion(num_clients, vehicle_capacity,
       demands, distance_matrix):
2      unvisited_clients = list(range(1, num_clients + 1))  #
       Clients non visit s
3      routes = []  # Liste des routes
4
5      while unvisited_clients:
6          current_route = [0]  # D p t au d but de chaque
       route
7          current_capacity = 0
8
9          while unvisited_clients:
10             best_client = None
11             best_cost = float('inf')
12             best_position = None
13
14             # Trouver le client avec le meilleur co t d'
       insertion
15             for client in unvisited_clients:
16                 if current_capacity + demands[client - 1] <=
       vehicle_capacity:
17                     for pos in range(1, len(current_route) +
```

```
    1):
18                          cost = (
19                              distance_matrix[current_route[pos
     - 1]][client]
20                              + distance_matrix[client][
    current_route[pos % len(current_route)]]
21                              - distance_matrix[current_route[
    pos - 1]][current_route[pos % len(current_route)]]
22                          )
23                          if cost < best_cost:
24                              best_cost = cost
25                              best_client = client
26                              best_position = pos
27
28              # Si aucun client ne peut  tre  ins r  ,
    terminez la route
29              if best_client is None:
30                  break
31
32              # Ins rer le client s lectionn     la
    meilleure position
33              current_route.insert(best_position, best_client)
34              current_capacity += demands[best_client - 1]
35              unvisited_clients.remove(best_client)
36
37          # Compl ter la route en retournant au d p t
38          current_route.append(0)
39          routes.append(current_route)
40
41      return routes
```

Listing 3 – Fonction `sequential_insertion`

La fonction `calculate_cost` est utilisée pour déterminer le coût total d'une solution, c'est-à-dire la somme des distances parcourues sur toutes les routes générées. Elle prend en entrée la liste des routes et la matrice des distances. Pour chaque route dans la liste, la fonction itère sur chaque paire consécutive de nœuds (`route[i]`, `route[i + 1]`) et additionne la distance entre ces deux nœuds, telle que spécifiée dans la matrice des distances. Ce processus est répété pour toutes les routes, et le coût total accumulé est finalement retourné. Cette fonction permet d'évaluer l'efficacité de la solution obtenue par l'heuristique d'insertion séquentielle en fournissant une mesure quantitative du coût associé aux routes générées.

```
1 def calculate_cost(routes, distance_matrix):
```

```
    total_cost = 0
    for route in routes:
        for i in range(len(route) - 1):
            total_cost += distance_matrix[route[i]][route[i + 1]]
    return total_cost
```

Listing 4 – Fonction `calculate_cost`

La fonction principale `solve_instance` coordonne l'ensemble du processus de résolution d'une instance donnée du problème de tournées de véhicules. Elle commence par charger les données de l'instance en appelant la fonction `load_instance` avec le chemin du fichier d'instance. Ensuite, elle calcule la matrice des distances en utilisant la fonction `calculate_distance_matrix` basée sur les coordonnées des nœuds. Après avoir préparé les données nécessaires, la fonction applique l'heuristique d'insertion séquentielle en appelant `sequential_insertion` pour générer les routes optimales. Une fois les routes construites, elle calcule le coût total de ces routes en utilisant la fonction `calculate_cost`. Enfin, la fonction retourne les routes générées ainsi que le coût total associé, fournissant ainsi une solution complète à l'instance du VRP.

```
def solve_instance(file_path):
    # Charger les donn es
    num_clients, vehicle_capacity, demands, coordinates = load_instance(file_path)

    # Calculer la matrice des distances
    distance_matrix = calculate_distance_matrix(coordinates)

    # G n rer une solution avec l'heuristique d'insertion s quentielle
    routes = sequential_insertion(num_clients, vehicle_capacity, demands, distance_matrix)

    # Calculer le co t total des routes
    cost = calculate_cost(routes, distance_matrix)

    # Retourner les routes et leur co t
    return routes, cost
```

Listing 5 – Fonction `solve_instance`

Le dernier bloc de code montre comment utiliser les fonctions précédentes pour résoudre une instance spécifique du VRP. Il vérifie si le script est exécuté

en tant que programme principal en utilisant `if __name__ == "__main__":`. Ensuite, il définit le chemin vers le fichier d'instance (`Case0.txt`), qui doit être remplacé par le chemin réel du fichier contenant les données de l'instance à résoudre. La fonction `solve_instance` est ensuite appelée avec ce fichier, et les résultats (routes et coût total) sont stockés dans les variables `solution` et `cost`. Enfin, ces résultats sont affichés à l'écran à l'aide des commandes `print`, permettant ainsi à l'utilisateur de voir les routes optimales générées par l'heuristique ainsi que le coût total associé.

```
if __name__ == "__main__":
    # Chemin vers le fichier d'instance
    instance_file = "/content/Case0.txt"  # Remplacez par le
   chemin de votre fichier

    # R soudre l'instance
    solution, cost = solve_instance(instance_file)

    # Afficher les r sultats
    print("Solution :", solution)
    print("Co t total :", cost)
```

Listing 6 – Bloc principal

```
Solution : [[0, 2, 1, 0], [0, 3, 0]]
Coût total : 31
```

Figure 4 – Résultats Heuristique

2.3 Principe de la méta-heuristique de Tabou

La méta-heuristique de tabou (ou Tabu Search, introduite par Fred Glover en 1986) est une méthode d'optimisation combinatoire qui améliore progressivement une solution en explorant son voisinage tout en évitant les cycles et les pièges des minima locaux. Elle repose sur l'utilisation d'une liste tabou pour interdire temporairement certaines solutions déjà explorées.

2.4 Code

Cette fonction lit les données depuis un fichier d'instance. La première ligne extrait le nombre de clients et la capacité des véhicules . La deuxième ligne récupère les demandes des clients . Les lignes restantes contiennent les coordonnées des nœuds (dépôt et clients). Ces données sont nettoyées et vérifiées avant d'être renvoyées.

```
def load_instance(file_path):
    with open(file_path, 'r') as f:
        lines = f.readlines()

    num_clients, vehicle_capacity = map(int, lines[0].split()
    )
    demands = list(map(int, lines[1].split()))
    coordinates = []
    for line in lines[2:]:
        if line.strip():
            parts = line.split()
            if len(parts) == 2:
                x, y = map(int, parts)
                coordinates.append((x, y))
    return num_clients, vehicle_capacity, demands,
    coordinates
```

Listing 7 – Fonction `load_instance`

Cette fonction calcule les distances euclidiennes entre tous les nœuds en utilisant leurs coordonnées. Les distances sont arrondies à l'entier le plus proche.

```
def calculate_distance_matrix(coordinates):
    num_nodes = len(coordinates)
    distance_matrix = np.zeros((num_nodes, num_nodes), dtype=
    int)
    for i in range(num_nodes):
        for j in range(num_nodes):
            if i != j:
                distance = np.sqrt((coordinates[j][0] -
    coordinates[i][0])**2 +
                                   (coordinates[j][1] -
    coordinates[i][1])**2)
                distance_matrix[i][j] = int(distance + 0.5)
    return distance_matrix
```

Listing 8 – Fonction `calculate_distance_matrix`

Cette fonction construit une solution initiale en assignant des clients à des véhicules tant que leur demande respecte la capacité maximale. Une route est terminée lorsque la capacité est atteinte.

```
def generate_initial_solution(num_clients, vehicle_capacity, demands):
    routes = []
    current_route = []
    current_capacity = 0
    for client in range(1, num_clients + 1):
        if current_capacity + demands[client - 1] > vehicle_capacity:
            routes.append([0] + current_route + [0])
            current_route = []
            current_capacity = 0
        current_route.append(client)
        current_capacity += demands[client - 1]
    if current_route:
        routes.append([0] + current_route + [0])
    return routes
```

Listing 9 – Fonction `generate_initial_solution`

Cette fonction calcule le coût total d'une solution en sommant les distances entre les nœuds pour chaque route.

```
def calculate_cost(routes, distance_matrix):
    total_cost = 0
    for route in routes:
        for i in range(len(route) - 1):
            total_cost += distance_matrix[route[i]][route[i + 1]]
    return total_cost
```

Listing 10 – Fonction `calculate_cost`

Génération des voisins (`generate_neighbors`)

Cette fonction génère des solutions voisines en permutant deux clients dans une même route.

```
def generate_neighbors(solution):
    neighbors = []
    for route in solution:
        for i in range(1, len(route) - 1):
            for j in range(i + 1, len(route) - 1):
```

```
                neighbor = [r[:] for r in solution]
                neighbor_route = neighbor[solution.index(
    route)]
                neighbor_route[i], neighbor_route[j] =
    neighbor_route[j], neighbor_route[i]
                neighbors.append(neighbor)
     return neighbors
```

Listing 11 – Fonction `generate_neighbors`

L'algorithme de recherche tabou explore les solutions voisines tout en interdisant temporairement certains mouvements via une liste tabou. La meilleure solution est mise à jour lorsqu'une solution meilleure est trouvée.

```
def tabu_search(initial_solution, distance_matrix,
    tabu_tenure=5, max_iterations=100):
     current_solution = initial_solution
     current_cost = calculate_cost(current_solution,
    distance_matrix)
     best_solution = current_solution
     best_cost = current_cost

     tabu_list = []
     for _ in range(max_iterations):
         neighbors = generate_neighbors(current_solution)
         neighbor_costs = [calculate_cost(neighbor,
    distance_matrix) for neighbor in neighbors]

         best_neighbor = None
         best_neighbor_cost = float('inf')
         for neighbor, cost in zip(neighbors, neighbor_costs):
             if neighbor not in tabu_list and cost <
    best_neighbor_cost:
                 best_neighbor = neighbor
                 best_neighbor_cost = cost

         if best_neighbor is None:
             break

         current_solution = best_neighbor
         current_cost = best_neighbor_cost
         if current_cost < best_cost:
             best_solution = current_solution
             best_cost = current_cost

         tabu_list.append(current_solution)
```

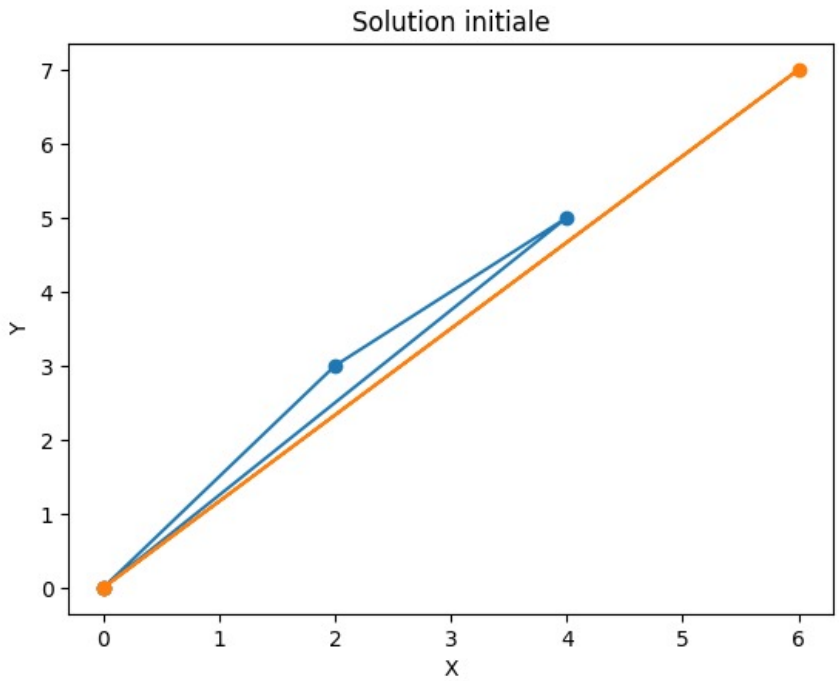
```
29            if len(tabu_list) > tabu_tenure:
30                tabu_list.pop(0)
31        return best_solution, best_cost
```

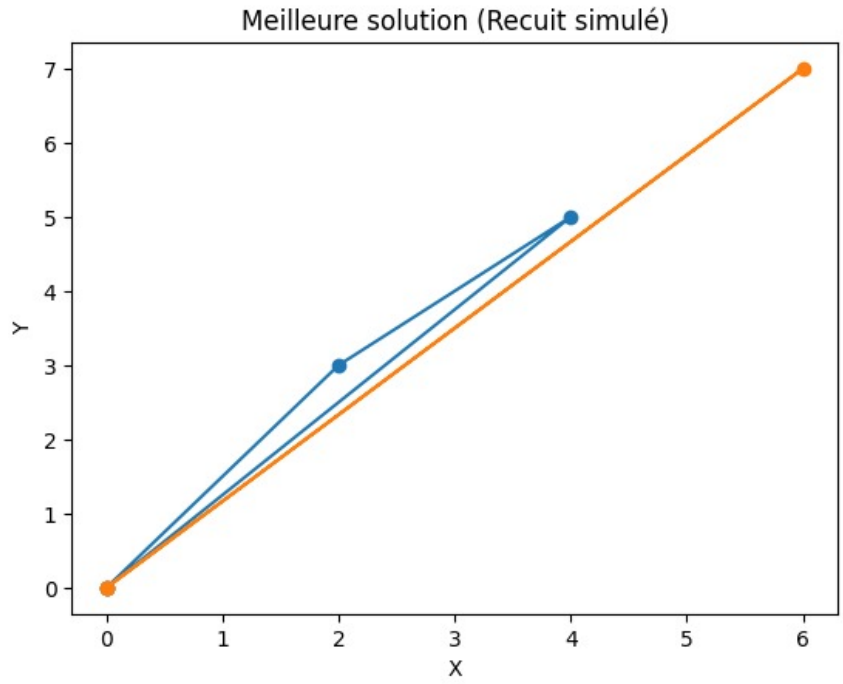
Listing 12 – Fonction `tabu_search`

Cette fonction coordonne le chargement des données, la construction de la solution initiale, et l'application de l'algorithme de recherche tabou pour trouver la meilleure solution.

```
1  def solve_instance(file_path):
2      num_clients, vehicle_capacity, demands, coordinates =
       load_instance(file_path)
3      distance_matrix = calculate_distance_matrix(coordinates)
4      initial_solution = generate_initial_solution(num_clients,
       vehicle_capacity, demands)
5      best_solution, best_cost = tabu_search(initial_solution,
       distance_matrix)
6      return best_solution, best_cost
```

Listing 13 – Fonction `solve_instance`





2.5 Conclusion

Une méta-heuristique, est une méthode plus générale .Elle est conçue pour explorer l'espace des solutions de manière efficace, tout en évitant de se bloquer dans des solutions locales.

Références

[1] Lien colab du code pour le cas 0 https://colab.research.google.com/drive/1qDF13H4vMO8m2tnIS6sZFY_DLA0zGk03?usp=sharing

[2] Optimisation de tournées de véhicules pour l'exploitation de Réseau Telecom file:///C:/Users/HP%20X360%201040%20G7/AppData/Local/Microsoft/Windows/INetCache/IE/7TZ0PLYE/malapert-06-FT[1].pdf

[3] Lien colab de l'Heuristique et la méta heuristique https://colab.research.google.com/drive/1kjZ4ZoCOUIPX_u9Rr7WKpDLPauNBHDai?usp=sharing#scrollTo=1-hljTeZx4HZ