Université libanaise Internationale en Mauritanie
Faculté des Sciences
Module : Recherche Opérationnelle
Année : 2025–2026

‘

Rapport RO

Réalisé par :

Randi 12430009
El Kebir 12430101
Ibtissam 12430037

Encadré par : Dr. EL BENANY Med Mahmoud

Plan

1. Introduction
2. Optimisation du transport
3. Problème des véhicules (VRP) & Optimisation combinatoire et heuristiques
4. Cas réel : routage des camions d'eau
5. Conclusion générale

A. Introduction générale

La recherche opérationnelle est une discipline qui vise à modéliser et à résoudre des problèmes de décision en optimisant l'utilisation des ressources disponibles. Elle est largement utilisée dans des domaines tels que le transport, la logistique et la gestion des systèmes.

Dans un premier temps, ce rapport s'intéresse à l'optimisation du transport à travers un problème de transport classique. Le problème initial est formulé à partir des données de départ, puis résolu à l'aide de méthodes de calcul. Une résolution à l'aide d'un outil informatique est également présentée afin de comparer et valider les résultats obtenus manuellement.

Dans un second temps, le rapport aborde l'optimisation combinatoire et les méthodes heuristiques, à travers le problème de tournées de véhicules (VRP). Ces notions sont ensuite appliquées à un cas réel de routage des camions d'eau, permettant de modéliser une situation concrète et de proposer une solution respectant les contraintes opérationnelles.

Ce travail met ainsi en évidence l'apport de la recherche opérationnelle dans la résolution de problèmes réels de transport et de logistique.

B. OPTIMISATION DU TRANSPORT

A. <u>Problème initial :</u> Le problème de transport consiste à déterminer comment répartir des marchandises depuis plusieurs sources vers plusieurs destinations, de manière à minimiser le coût total de transport, tout en respectant les contraintes d'offre et de demande.

Dans ce travail, l'objectif est de trouver la solution optimale permettant de satisfaire toutes les demandes à partir des ressources disponibles, au coût total minimal.

B. Données de départ : Les coûts Le problème étudié comporte deux sources et trois destinations.

Sources (offres) :
S1 = 100 unités
S2 = 150 unités

Destinations (demandes) :
D1 = 80 unités
D2 = 120 unités
D3 = 50 unités

Matrice des coûts unitaires :

	D1	D2	D3
S1	2	3	1
S2	5	4	8

C. Méthode de calcul : Afin de comprendre la logique du problème de transport, plusieurs méthodes classiques peuvent être utilisées.

- ✓ Coin nord-ouest : est une méthode classique permettant d'obtenir une solution initiale faisable. Elle consiste à commencer les allocations à partir de la cellule située en haut à gauche du tableau, sans tenir compte des coûts, et à satisfaire progressivement les contraintes d'offre et de demande.

Règle d'allocation :

$$x_{ij} = \min(\text{offre restante},\ \text{demande restante})$$

Cette méthode ne tient pas compte des coûts de transport.

Le tableau ci-dessous représente la solution initiale obtenue par la méthode du coin nord-ouest.
Le coût total = 1020

	D1	D2	D3	Offre
S1	80	20	0	100
S2	0	100	50	150
Demande	80	120	50	

- ✓ Méthode MODI (Modified Distribution Method) : est utilisée pour vérifier l'optimalité d'une solution de transport. Elle repose sur le calcul des potentiels associés aux lignes et aux colonnes du tableau afin de déterminer si une amélioration du coût est possible.

  Formule :

$$\Delta_{ij} = C_{ij} - (u_i + v_j)$$

Si tous les $\Delta_{ij} \geq 0$, la solution est optimale.

Sinon, la solution peut être améliorée.

Le tableau ci-dessous représente la solution initiale obtenue par MODI.
Le coût total = 900.

	D1	D2	D3	Offre
S1	80	0	20	100
S2	0	120	30	150
Demande	80	120	50	

- ✓ La méthode du moindre coût : consiste à affecter en priorité les quantités aux cellules ayant le coût unitaire le plus faible, afin de réduire le coût total dès la solution initiale.
  Principe :
  - choisir la cellule avec le coût minimal
  - affecter la quantité maximale possible
  - mettre à jour les offres et demandes

D. Résolution à l'aide d'un outil informatique :

Bien que les méthodes précédentes permettent de comprendre la logique du problème, le calcul manuel peut devenir long et complexe. Pour obtenir directement la solution optimale, le problème de transport a été modélisé sous forme de programmation linéaire et résolu à l'aide du langage Python, en utilisant la bibliothèque PuLP.

Cette bibliothèque permet de définir les variables de décision, la fonction objectif et les contraintes, puis de résoudre automatiquement le modèle à l'aide d'un solveur.

Voici le code

```python
import pulp

# Données
supply = [100, 150]             # Offre des sources S1 et S2
demand = [80, 120, 50]          # Demande des destinations D1, D2, D3
costs = [[2, 3, 1],             # Coûts unitaires
         [5, 4, 8]]

sources = ["S1", "S2"]
destinations = ["D1", "D2", "D3"]

# Création du problème (Minimisation)
prob = pulp.LpProblem("Transport_Minimisation", pulp.LpMinimize)

# Variables de décision X[i,j] >= 0
x = pulp.LpVariable.dicts("X",
                          ((i, j) for i in range(len(sources)) for j in range(len(destinations))),
                          lowBound=0,
                          cat='Continuous')

# Fonction objectif : min Z = sum(Cij * Xij)
prob += pulp.lpSum(costs[i][j] * x[i, j] for i in range(len(sources)) for j in range(len(destinations)))

# Contraintes d'offre
for i in range(len(sources)):
    prob += pulp.lpSum(x[i, j] for j in range(len(destinations))) <= supply[i]
```

```python
# Contraintes de demande
for j in range(len(destinations)):
    prob += pulp.lpSum(x[i, j] for i in range(len(sources))) == demand[j]

# Résolution
prob.solve()

# Affichage des résultats
print("Status:", pulp.LpStatus[prob.status])
print("Solution optimale :")
for i in range(len(sources)):
    for j in range(len(destinations)):
        print(f"X[{sources[i]},{destinations[j]}] =", x[i,j].varValue)

print("Coût total minimal Z =", pulp.value(prob.objective))
```

Résultats obtenus :

	D1	D2	D3	Offre
S1	50	0	50	100
S2	30	120	0	150
Demande	80	120	50	
Coût total minimal Z = 780				

La solution obtenue respecte toutes les contraintes d'offre et de demande. Les quantités allouées sont positives et aucune contrainte n'est violée. La solution est donc faisable et optimale.

E. Lien avec le problème VRP :

Le problème de transport constitue une base du VRP. Alors que le problème de transport se concentre sur la répartition des quantités, le VRP ajoute la notion de tournée et d'ordre de visite, ce qui le rend plus complexe et plus proche des situations réelles.

3. VRP + Optimisation combinatoire et heuristiques

Organisation (thèmes 2 & 3) : Le VRP est un problème d'optimisation combinatoire (NP-difficile). Pour cette raison, nous traitons ensemble le thème 2 (VRP) et le thème 3 (optimisation combinatoire et heuristiques) : nous présentons le problème (données et contraintes), puis une méthode exacte (combinatoire) et une méthode heuristique (Clarke & Wright), avant de valider par un outil informatique.

A. Problème initial : On considère un VRP avec un dépôt et des clients à desservir. L'objectif est de déterminer des tournées partant du dépôt et y revenant, de manière à minimiser la distance totale parcourue, tout en respectant la contrainte de capacité.

B. Données de départ : Le problème étudié est défini par les données suivantes :

- Dépôt : nœud 0
- Clients : C1, C2, C3, C4, C5
- Demandes :
  C1 = 2
  C2 = 3
  C3 = 2
  C4 = 3
  C5 = 1

- Capacité du véhicule : Q = 5

  La contrainte de capacité à respecter pour chaque tournée est : $\sum q_i \leq Q$

La somme totale des demandes est égale à 11, ce qui est supérieure à la capacité du véhicule. Il est donc nécessaire d'effectuer plusieurs tournées afin de respecter la contrainte de capacité.

- Matrice des distances : La matrice des distances d(i,j) donne la distance entre le nœud de départ i et le nœud d'arrivée j. Elle est utilisée pour calculer le coût total des tournées.

i/j	1	2	3	4	5
0	4	6	5	7	3
1	4	3	4	6	2
2	6	3	2	4	5
3	5	2	2	3	4
4	7	4	3	3	6
5	3	5	4	6	1

- Coût d'une tournée :

Pour une tournée $0 \to v_1 \to \cdots \to v_k \to 0$, le coût est :

$$\text{Coût} = d(0, v_1) + \sum_{t=1}^{k-1} d(v_t, v_{t+1}) + d(v_k, 0)$$

C. <u>Méthode de calcul :</u>

- ✓ Méthode combinatoire : La méthode combinatoire explore les répartitions possibles des clients en tournées faisables (capacité respectée), puis teste les ordres de visite afin de retenir le coût total minimal.

Etape 1 :

Le tableau suivant présente les différentes sous-tournées faisables, les permutations possibles des clients ainsi que le coût associé à chaque ordre de visite. Pour chaque tournée, le meilleur ordre est retenu en fonction du coût minimal obtenu.

sous tournee	permutation possible	meilleur ordre	cout
(1)	(1)	(1)	8
(2)	(2)	(2)	12
(3)	(3)	(3)	10
(4)	(4)	(4)	14
(5)	(5)	(5)	6
(1,2)	(1,2) (2,1)	(1,2)	13
(1,3)	(1,3) (3,1)	(1,3)	13
(1,4)	(1,4) (4,1)	(1,4)	17
(1,5)	(1,5) (5,1)	(1,5)	9
(2,3)	(2,3) (3,2)	(2,3)	13
(2,5)	(2,5) (5,2)	(2,5)	14
(3,4)	(3,4) (4,3)	(3,4)	14
(4,5)	(4,5) (5,4)	(4,5)	16
(1,3,5)	toutes permutation	(5,1,3)	11
(3,5)	(3,5)	(5,3)	9

Etape 2 :

Maintenant on va chercher les partitions qui couvrent tous les clients. Pour 5 clients, on cherche toutes les combinaisons de sous-tournées dont la demande inférieure ou égale à Q et qui couvrent tous les clients.

- Partition 1 : (1,2) ;(3,4) ; (5)
  Cout total =33
- Partition 2 : (1,3,5) ;(2) ; (4)
  Cout total =37
- Partition 3 : (1,4) ;(2,3) ; (5)
  Cout total =36
- Partition 4 : (1,5) ;(2,3) ; (4)
  Cout total =36
- Partition 5 : (1,3) ;(2,5) ; (4)
  Cout total =41

Etape 3 : Regroupement des partitions par cout total

Cout total	Partitions
33	(1,2) ; (3,4) ; (5)
36	(1,4); (2,3) ;(5)
36	(1,5); (2,3) ; (4)
37	(1,3,5) ; (2) ; (4)
41	(1,3) ; (2,5) ; (4)

Solution combinatoire optimale :

- Partition optimale : (1,2) ; (3,4) ; (5)
- Cout minimale total : 33

✓ Méthode heuristique de Clarke & Wright (Savings) :

Nous utilisons la méthode heuristique de Clarke et Wright, également appelée méthode des économies (Savings), afin de résoudre le problème de tournées de véhicules. Cette méthode permet de construire des tournées efficaces sans explorer toutes les solutions possibles, ce qui réduit le temps de calcul.
Le principe de la méthode consiste à mesurer l'économie réalisée lorsqu'on regroupe deux clients i et j dans une même tournée au lieu de les desservir séparément. Les clients sont ensuite regroupés progressivement en fonction des économies les plus élevées, tout en respectant la contrainte de capacité du véhicule.

Etape 1 calcul des savings pour 2 clients i et j :

La formule : $S(i,j) = d(0,i) + d(0,j) - d(i,j)$

i-j	Savings
1-2	7
1-3	5
1-4	5
1-5	5
2-3	9
2-4	9
2-5	4
3-4	9
3-5	4
4-5	4

Etape 2 : Fusionner les tournées :

Fusion $(2,3)$:
$S(2,3) = 9, d_2 + d_3 = 5 \leq Q \Rightarrow$ **OK**, nouvelle tournée $(2,3)$

Fusion $(2,4)$:
$S(2,4) = 9, d_{(2,3)} + d_4 = 8 > Q \Rightarrow$ **Impossible**

Fusion $(3,4)$:
$S(3,4), d_{(2,3)} + d_4 = 8 > Q \Rightarrow$ **Impossible**

Fusion $(1,2)$:
$S(1,2) = 7, d_{(2,3)} + d_1 = 7 > Q \Rightarrow$ **Impossible**

Fusion $(1,3)$:
$S(1,3) = 5, d_{(2,3)} + d_1 = 7 > Q \Rightarrow$ **Impossible**

Fusion $(1,4)$:
$S(1,4) = 5, d_1 + d_4 = 5 \leq Q \Rightarrow$ **OK**, nouvelle tournée $(1,4)$

Etape 3 : Calcul cout heuristiques :

(2,3) : 0 →2→3→0 : 13

(1,4) : 0 →1→4→0 : 17

(5) : 0 →5→0 : 6

} Cout total = 36

D. Comparaison entre la méthode combinatoire et la méthode heuristique :

La méthode combinatoire a permis d'obtenir une solution optimale avec un coût total égal à 33, en testant l'ensemble des tournées possibles.
En revanche, la méthode heuristique de Clarke et Wright a fourni une solution approchée avec un coût total égal à 36, obtenue en un temps de calcul plus réduit.

Cette comparaison montre que la méthode combinatoire garantit le meilleur coût, tandis que la méthode heuristique offre un bon compromis entre qualité de solution et rapidité de calcul, ce qui la rend plus adaptée aux problèmes réels de grande taille.

E. Résolution à l'aide d'un outil informatique :

La résolution du problème a également été effectuée à l'aide de la bibliothèque **itertools** c'est une bibliothèque Python permettant de générer efficacement des permutations et des combinaisons, utilisée pour explorer l'espace des solutions en optimisation combinatoire.

Voici le code

```python
from itertools import permutations, combinations

# --- Données ---
clients = [1,2,3,4,5]
demands = {1:2,2:3,3:2,4:3,5:1}
Q = 5

# Matrice des distances (symétrique)
d = {
    (0,1):4,(1,0):4,
    (0,2):6,(2,0):6,
    (0,3):5,(3,0):5,
    (0,4):7,(4,0):7,
    (0,5):3,(5,0):3,
    (1,2):3,(2,1):3,
    (1,3):4,(3,1):4,
    (1,4):6,(4,1):6,
    (1,5):2,(5,1):2,
    (2,3):2,(3,2):2,
    (2,4):4,(4,2):4,
    (2,5):5,(5,2):5,
    (3,4):3,(4,3):3,
    (3,5):4,(5,3):4,
    (4,5):6,(5,4):6
}
```

```
            for part in partitions(lst[1:]):
                yield [[first]] + [p[:] for p in part]
                for i in range(len(part)):
                    new_part = [p[:] for p in part]
                    new_part[i] = new_part[i] + [first]
                    yield new_part

    # Eviter les doublons
    seen = set()
    best_solution = None
    best_total_cost = float('inf')

    for p in partitions(clients):
        norm = tuple(sorted(tuple(sorted(sub)) for sub in p))
        if norm in seen: continue
        seen.add(norm)
        feasible = True
        total_cost = 0
        routes_info = []
        for subset in norm:
            demand = sum(demands[c] for c in subset)
            if demand > Q:
                feasible = False
                break
            cost, order = best_tsp(subset)
            total_cost += cost
            routes_info.append((subset, order, cost))
        if feasible and total_cost < best_total_cost:
            best_total_cost = total_cost
            best_solution = routes_info

    return best_solution, best_total_cost

# --- Méthode Clarke & Wright ---
def clarke_wright_vrp(clients, demands, Q, d):
    # Initialiser une tournée par client
    routes = {c:[c] for c in clients}

    # Calcul des savings
    savings = []
    for i in clients:
        for j in clients:
            if i<j:
                s = d[(0,i)] + d[(0,j)] - d[(i,j)]
                savings.append((s,i,j))
    savings.sort(reverse=True)  # tri décroissant
```

```python
    for s,i,j in savings:
        # Trouver les routes contenant i et j
        route_i = next((r for r in routes.values() if i in r), None)
        route_j = next((r for r in routes.values() if j in r), None)
        if route_i == route_j: continue  # même route
        # Vérifier capacité
        demand_i = sum(demands[c] for c in route_i)
        demand_j = sum(demands[c] for c in route_j)
        if demand_i + demand_j <= Q:
            # Fusionner : i en fin de route_i, j en début de route_j (ou inverse)
            new_route = route_i + route_j
            # Supprimer anciennes routes
            for key in list(routes.keys()):
                if key in route_i or key in route_j:
                    del routes[key]
            # Ajouter nouvelle route
            routes[new_route[0]] = new_route

    # Calcul du coût total
    total_cost = 0
    final_routes = []
    for r in routes.values():
        c = route_cost(r)
        final_routes.append((r, c))
        total_cost += c
```

```python
            total_cost += c

    return final_routes, total_cost

# --- Exécution ---
comb_solution, comb_cost = combinatorial_vrp(clients, demands, Q)
cw_solution, cw_cost = clarke_wright_vrp(clients, demands, Q, d)

# --- Affichage ---
print("=== Solution combinatoire optimale ===")
for r in comb_solution:
    print(f"Tournée {r[0]} -> ordre optimal {r[1]} -> coût {r[2]}")
print(f"Coût total = {comb_cost}\n")

print("=== Solution Clarke & Wright ===")
for r in cw_solution:
    print(f"Tournée {r[0]} -> coût {r[1]}")
print(f"Coût total = {cw_cost}")
```

Résultats obtenus :

```
=== Solution combinatoire optimale ===
Tournée (1, 2) -> ordre optimal (1, 2) -> coût 13
Tournée (3, 4) -> ordre optimal (3, 4) -> coût 15
Tournée (5,) -> ordre optimal (5,) -> coût 6
Coût total = 34

=== Solution Clarke & Wright ===
Tournée [5] -> coût 6
Tournée [3, 4] -> coût 15
Tournée [1, 2] -> coût 13
Coût total = 34
```

F. Lien avec le cas réel :

Le cas réel de routage des camions d'eau correspond à une modélisation du Vehicle Routing Problem (VRP), dans laquelle les concepts théoriques sont appliqués à une situation concrète afin d'optimiser les tournées des véhicules.

4. Cas réel : routage des camions d'eau

A. Problème initial :

Dans les quartiers périphériques, l'approvisionnement en eau est assuré par des camions-citernes partant d'un dépôt central. L'objectif est de planifier les tournées de distribution vers plusieurs points de demande afin de minimiser la distance totale parcourue (donc le coût) tout en respectant la capacité journalière des véhicules.
Le problème consiste donc à déterminer les routes à suivre et les quantités livrées à chaque point, en garantissant que la somme des volumes transportés sur une tournée ne dépasse pas la capacité du camion.

B. Données de départ :

- Dépôt 0 → (50,50)
- Capacité camion = 1000 L

client	id	coordonnees	demande
1	1	(52,54)	400
2	2	(48,57)	250
3	3	(55,47)	300
4	4	(45,49)	150
5	5	(535,60)	600
6	6	(46,45)	200

C. Méthode de calcul :

La résolution du cas réel repose sur une méthode heuristique, choisie pour sa capacité à fournir des solutions efficaces et adaptables aux contraintes pratiques du problème de routage des camions d'eau.

Etape 1 : distance euclidienne

$$d(A,B) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Dépôts → clients :

0-i	resultat
0-1	4,472
0-2	7,280
0-3	5,831
0-4	5,099
0-5	10,601
0-6	6,403

Clients → Clients :

i-j	distance
1-2	5
1-3	7,615
1-4	8,602
1-5	6,185
1-6	10,817
2-3	12,207
2-4	8,544
2-5	6,265
2-6	12,166
3-4	10,198
3-5	13,085
3-6	9,220
4-5	13,896
4-6	4,123
5-6	16,77

Etape 2 : Calcul des savings :

$$S(i,j) = d(0,i) + d(0,j) - d(i,j)$$

i-j	savings
1-2	6,752
1-3	2,688
1-4	0,969
1-5	8,888
1-6	-0,054
2-3	0,904
2-4	3,835
2-5	11,616
2-6	1,517
3-4	0,732
3-5	3,347
3-6	3,014
4-5	1,804
4-6	7,379
5-6	0,234

Etape 3 : Construction des tournées

La règle de fusion consiste à regrouper plusieurs demandes dans un même trajet lorsque la somme des quantités demandées ne dépasse pas la capacité maximale du véhicule Q.
Si la somme des demandes est inférieure ou égale à Q , alors les demandes peuvent être fusionnées et traitées ensemble. Dans le cas contraire, elles doivent être traitées séparément afin de respecter la contrainte de capacité.

Route	Clients	Charge
R1	1	400
R2	2	250
R3	3	300
R4	4	150
R5	5	600
R6	6	200

- Paire (1,5)

  Saving = 8,888

  5 ∈ R(2,5)

  Charge totale > 1000

  ⇒ Fusion impossible

- Paire (4,6)

  Saving = 7,849

  4 ∈ R(4), 6 ∈ R(6)

  Charge fusionnée ≤ 1000

  Clients aux extrémités : oui

  ⇒ Fusion acceptée

  Nouvelle route : R(4,6)

  Routes actuelles :

  R1 = (1)

  R2 = (2,5)

  R3 = (3)

  R4 = (4,6)

- Paire (1,2)

  Saving = 6,752

  2 ∈ R(2,5)

  Charge totale > 1000

  ⇒ Fusion impossible

- Paire (2,4)

  Saving = 3,835

  2 ∈ R(2,5), 4 ∈ R(4,6)

  Charge totale > 1000

  ⇒ Fusion impossible

- Paire (3,5)

  Saving = 3,347

  $5 \in R(2,5)$

  Charge totale > 1000

  $\Rightarrow$ Fusion impossible

- Paire (3,6)

  Saving = 3,014

  $6 \in R(4,6)$

  Charge totale $\leq$ 1000

  $\Rightarrow$ Fusion acceptée

  Routes finales :

  R1 = (1)

  R2 = (2,5)

  R3 = (3,4,6)

Vérification des charges finales :

R1	Client 1	charge=400
R2	Client 2 ->5	charge=850
R3	Client 3->4->6	charge=650

⇨ Toutes respectent la capacité camion .

Etape 4 : Ajout dépôt aux extrémités et calcul des distances approximatives

Route	Séquence complete	Change	Distance approximative
R1	0-->1-->0	400	8,944
R2	0-->2-->5-->0	850	24,146
R3	0-->3-->4-->6-->0	650	26,555

- Distance totale = 57,645
- Nombres de camions utilisés = 3

D. Résolution à l'aide d'un outil informatique :

Le problème de routage capacitaire (CVRP) a été résolu à l'aide de l'outil informatique OR-Tools.

Cet outil permet de générer automatiquement des tournées optimales tout en respectant la contrainte de capacité des véhicules et en minimisant la distance totale parcourue.

Voici le code

```python
# cvrp_ortools.py
# Résout le CVRP (capacitated VRP) avec OR-Tools pour nos données d'exemple.
# Prérequis : pip install ortools

from ortools.constraint_solver import pywrapcp, routing_enums_pb2
import math

# Données
depot = (50.0, 50.0)
clients = [
    (52.0, 54.0, 400),   # id 1
    (48.0, 57.0, 250),   # id 2
    (55.0, 47.0, 300),   # id 3
    (45.0, 49.0, 150),   # id 4
    (53.5, 60.0, 600),   # id 5
    (46.0, 45.0, 200),   # id 6
]
capacity = 1000
num_vehicles = 5  # on alloue plus de véhicules que nécessaire ; OR-Tools n'utilisera que ceux nécessaires

# Construction des nœuds (0 = dépôt)
nodes = [depot] + [(c[0], c[1]) for c in clients]
demands = [0] + [c[2] for c in clients]

# Matrice des distances (float)
def euclid(a,b):
    return math.hypot(a[0]-b[0], a[1]-b[1])

dist_matrix = [[euclid(nodes[i], nodes[j]) for j in range(len(nodes))] for i in range(len(nodes))]
```

```python
# OR-Tools requires integer costs -> on scale
SCALE = 1000

# Manager & RoutingModel
manager = pywrapcp.RoutingIndexManager(len(dist_matrix), num_vehicles, 0)
routing = pywrapcp.RoutingModel(manager)

# Transit (distance) callback
def distance_callback(from_index, to_index):
    from_node = manager.IndexToNode(from_index)
    to_node = manager.IndexToNode(to_index)
    return int(dist_matrix[from_node][to_node] * SCALE)

trans_callback_idx = routing.RegisterTransitCallback(distance_callback)
routing.SetArcCostEvaluatorOfAllVehicles(trans_callback_idx)

# Demand callback (capacity)
def demand_callback(from_index):
    from_node = manager.IndexToNode(from_index)
    return demands[from_node]

demand_callback_idx = routing.RegisterUnaryTransitCallback(demand_callback)
routing.AddDimensionWithVehicleCapacity(
    demand_callback_idx,
    0,                      # null capacity slack
    [capacity] * num_vehicles,  # vehicle capacities
    True,                   # start cumul to zero
    'Capacity'
)
```

```python
# Search parameters
search_parameters = pywrapcp.DefaultRoutingSearchParameters()
search_parameters.first_solution_strategy = routing_enums_pb2.FirstSolutionStrategy.PATH_CHEAPEST_ARC
search_parameters.local_search_metaheuristic = routing_enums_pb2.LocalSearchMetaheuristic.GUIDED_LOCAL_SEARCH
search_parameters.time_limit.seconds = 10

# Solve
solution = routing.SolveWithParameters(search_parameters)
if not solution:
    print("Aucune solution trouvée (vérifier l'installation OR-Tools / paramètres).")
    exit(1)

# Extract routes
total_distance = 0.0
used_vehicles = 0
for v in range(num_vehicles):
    index = routing.Start(v)
    if routing.IsEnd(solution.Value(routing.NextVar(index))):
        continue
    used_vehicles += 1
    route = []
    load = 0
    route_distance = 0.0
    while not routing.IsEnd(index):
        node = manager.IndexToNode(index)
        route.append(node)
        load += demands[node]
        next_index = solution.Value(routing.NextVar(index))
        if not routing.IsEnd(next_index):
            route_distance += dist_matrix[node][manager.IndexToNode(next_index)]
        else:
            # add last leg back to depot
            route_distance += dist_matrix[node][0]
        index = next_index
    # ensure depot at ends
    if route[0] != 0:
        route = [0] + route
    if route[-1] != 0:
        route = route + [0]
    print(f"Vehicle {v+1}: route = {' -> '.join(map(str, route))}, load = {load} L, distance ≈ {route_distance:.3f} km")
    total_distance += route_distance

print(f"\nTotal distance (all vehicles): {total_distance:.3f} km")
print(f"Vehicles used: {used_vehicles}")
```

Résultats obtenus :

```
Vehicle 1: route = 0 -> 2 -> 4 -> 6 -> 3 -> 0, load
 = 900 L, distance ≈ 34.998 km
Vehicle 2: route = 0 -> 1 -> 5 -> 0, load = 1000 L,
 distance ≈ 21.252 km

Total distance (all vehicles): 56.249 km
Vehicles used: 2
```