



modefair

Home Assessment

Heterogeneous Fleet
Vehicle Routing Problem
(HFVRP)

Name: Daryl Gan En-Wei

Duration: 30th May 2024, 9am ~ 4th June 2024, 9am
(5 days)

Table of Contents

Introduction	1
Problem Description	1
Methodology	2
Algorithm Selection and Justification	2
Implementation	3
Programming Language and Libraries	3
Code Structure & Snippets	3
Results and Analysis	16
Results Comparison	16
Test Case	17
Challenges, Limitation and Future Work	18
Challenges	18
Limitation	18
Future Work	18
Conclusion	19
References	20

Introduction

Problem Description

The Vehicle Routing Problem (VRP) poses a significant challenge for logistics firms. It involves efficiently distributing goods from a central depot to various customers. With the current heightened demand and challenges in logistics, advanced systems are increasingly essential for effective delivery routing and execution. According to Konstantakopoulos et al. (2020), the Heterogeneous Fleet Vehicle Routing Problem (HFVRP) extends the traditional VRP by including a fleet of vehicles with varying types, each with different capacities and varying costs.

Objectives:

- To optimise the routing of a fleet of vehicles for the efficient delivery of goods to multiple customer locations.
- To minimise delivery costs while ensuring that all delivery points are covered and all customer demands are fulfilled.

Hard Constraints:

- Each delivery location must be visited exactly once.
- The total demand of each vehicle route must not exceed its maximum capacity.

Soft Constraint:

- Minimise cost required to meet all demands.

Assumptions:

- The vehicles start and end their routes at the same depot location.
- Each vehicle only travels one round trip. (depart from depot and back to the depot)
- There is no limit on the number of vehicles.
- Travel times between any two locations are the same in both directions.
- Deliveries can be made at any time, there are no time windows for deliveries.
- Vehicle travel distance is calculated using Euclidean distance formula:
$$\text{Distance in km} = 100 * \sqrt{(\text{Longitude2} - \text{Longitude1})^2 + (\text{Latitude2} - \text{Latitude1})^2}$$

Methodology

Algorithm Selection and Justification

Vehicle routing problem and related algorithms for logistics...

Table 4 Correlation between VRP variants and algorithms

VRP variants	Exact	Heuristics			Local search metaheuristics								Population search metaheuristics								Hybrid
		Constrn.	Two-Phase	Local imp.	SA	TS	VNS	ILS	(A)LS	GLS	GRASP	LSA	GA	MA	EA	ACO	PSO	SIA	SS		
CVRP	26	60	3	42	16	20	31	15	23	1	3	8	27	5	17	18	13	9	2	44	
VRPTW	15	29	1	15	9	16	13	3	14		1	2	22	5	14	12	7	6	1	32	
VRPPD	7	16	2	11	6	7	11	6	3		1	2	5		4	4	4			10	
HFVRP	6	10	1	6	4	5	9	5	5				6	2	1	3	2	2		9	
MDVRP	3	12		5	2		4	7	1		1		9			2	2	1		11	
GVRP	5	6	1	3	2	4	5	2	9				2			1		1		5	
DVRP	1	8		3		1	1	1	2		1		3	1	2	4	3	1	1	1	
2D-VRP		4		5	3	1	6	2	2						1			1		2	
OVDP	2	2		2	1	1	2	4		1		1			1		1	1		2	
SDVRP	3	5		3			1	1	1												
TDVRP		4		1			2		1				2			2	1	1		1	
MEVRP	4	3		1			1		2			1	1	2				1		2	
MTVRP	2	3		2			2	1	2				1	1						2	
PVRP	2				1	1										2					
ConVRP		2		2		2	2		1												
TTRP	1	1	1	1	1		1													1	


 Springer

Figure 1: Count of approaches taken by various studies for different variants of VRP

Figure 1 shows that the most commonly used methods for solving HFVRP are Variable Neighborhood Search (VNS) for local search metaheuristics and Genetic Algorithm (GA) for population search metaheuristics. Therefore, I have employed these two algorithms.

Local search metaheuristics iteratively explore the neighbourhood of a given solution to improve its quality, starting with an initial solution and moving to neighbouring solutions that offer improvements in terms of an objective function. Population search metaheuristics maintain a population of candidate solutions and explore the solution space by exchanging information among multiple solutions simultaneously.

While local search algorithms focus on intensifying the search around promising solutions, population search algorithms aim to diversify the search across the entire solution space, leveraging the collective knowledge of multiple solutions to guide the process.

GA mimics natural selection and genetics to explore the solution space and find near-optimal solutions. GA begins by initialising a population of potential routes for the fleet of vehicles and iteratively improves them through selection, crossover, mutation, and replacement operations. GA excels at balancing exploration and exploitation, enabling effective navigation of large solution spaces to find high-quality solutions even in complex problems.

VNS starts with an initial solution and iteratively applies local search within different neighbourhood structures to explore and improve solution quality. Upon reaching a local optimum or exhausting a neighbourhood, VNS perturbs the solution to explore new regions of the solution space. Through these methods, VNS effectively escapes local optima to find high-quality solutions.

Implementation

Programming Language and Libraries

Programming Language: Python

Libraries:

```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
from scipy.spatial import distance_matrix
import random
```

Code Structure & Snippets

Data Code Segment:

- Read customer and vehicle data from csv files
- Declare depot location
- Plot VRP map

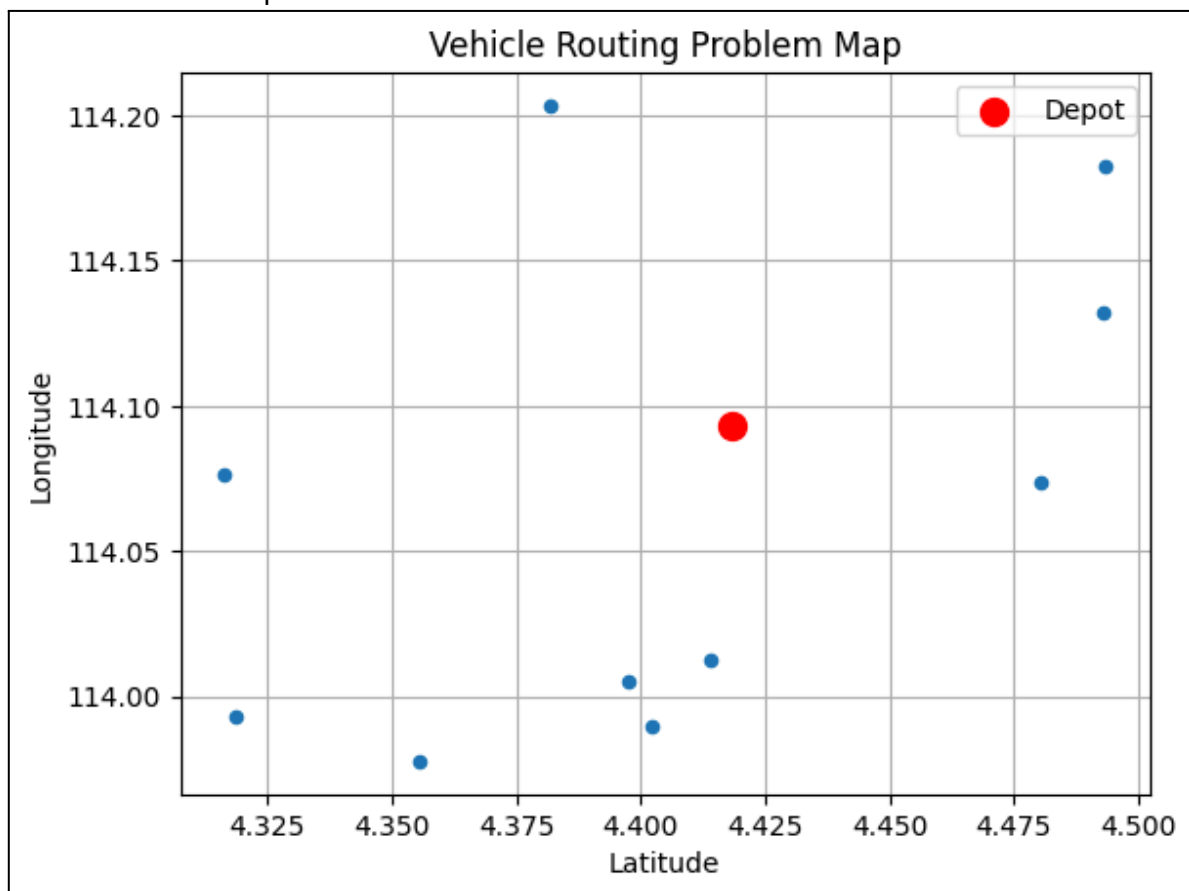


Figure 2: Vehicle Routing Problem Map

- Calculate distance matrix

Functions used by both Algorithms Code Segment:

- `create_initial_solution()`

Description: Generate a random initial solution

Parameters: None

Returns: Random permutation of customers

Pseudocode:

RETURN random permutation of customers

- `evaluate(solution)`

Description: Calculate total distance and cost of given solution

Parameters: solution

Returns:

total_distance: Total distance travelled of given solution

total_cost: Total cost of given solution

Pseudocode:

INITIALIZE total_distance and total_cost to 0

INITIALIZE route, routes, route_demand, and vehicle_index

FOR each customer_index in solution:

SET demand to the customer's demand

IF route_demand + demand <= current vehicle capacity:

ADD customer_index to route and update route_demand

ELSE:

SAVE route and reset for next vehicle

UPDATE vehicle_index

IF there are remaining customers in route:

SAVE route

FOR each saved route:

CALCULATE route_distance

UPDATE total_distance and total_cost

RETURN (total_distance, total_cost)

- `print_solution(best_solution, customers, vehicles, dist_matrix, depot)`

Description: Display details of given solution

Parameters:

best_solution: List of customer indices
customers: DataFrame containing customer information
vehicles: DataFrame containing vehicle information
dist_matrix: Matrix of distances between each pair of locations
depot: Location of depot

Returns: None

Pseudocode:

```
INITIALIZE total_distance and total_cost to 0
INITIALIZE route, routes, route_demand, and vehicle_index

FOR each customer_index in solution:
    SET demand to the customer's demand
    IF route_demand + demand <= current vehicle capacity:
        ADD customer_index to route and update route_demand
    ELSE:
        SAVE route and reset for next vehicle
        UPDATE vehicle_index

IF there are remaining customers in route:
    SAVE route

FOR each saved route:
    CALCULATE route_distance
    FORMAT and print route details
    UPDATE total_distance and total_cost

PRINT total_distance and total_cost
CALL plot_routes(routes, customers, depot)
```

- `plot_routes(routes, customers, depot)`

Description: Visualise routes of vehicles

Parameters:

routes: List of customer indices, vehicle, demand

customers: DataFrame containing customer information

depot: Location of depot

Returns: None

Pseudocode:

PLOT depot location

FOR each customer location in customers:

PLOT customer location

DEFINE colours for routes

FOR each route, vehicle_index, route_demand in routes:

PLOT route with colour

FOR each customer index in route:

ANNOTATE customer location with its index

SET title, labels, legend, and grid properties for the plot

DISPLAY the plot

Genetic Algorithm Code Segment:

- `select(population, fitnesses, k=3)`

Description: Tournament selection to choose k individuals from a population based on their fitness values and selects the individual with the lowest fitness

Parameters:

population: List of solutions

fitnesses: List of fitness values corresponding to individual

k: Tournament size

Returns: List of selected individuals

Pseudocode:

INITIALIZE selected as an empty list

FOR each individual in population:

 tournament = randomly select k individuals from population with replacement

 SELECT individual with lowest fitness from the tournament

 ADD selected individual to selected list

RETURN selected

- `crossover(parent1, parent2)`

Description: Randomly select two crossover points, copies gene segment from one parent to one child, and fills remaining genes from the other parent

Parameters:

parent1: First parent individual

parent2: Second parent individual

Returns: Two child individuals

Pseudocode:

size = length of parent1

cxpoint1, cxpoint2 = sorted(randomly select two crossover points from range(0, size))

INITIALIZE children

child1[cxpoint1:cxpoint2] = genes from parent1[cxpoint1:cxpoint2]

child2[cxpoint1:cxpoint2] = genes from parent2[cxpoint1:cxpoint2]

fill_child(child1, parent2, cxpoint2, size)

fill_child(child2, parent1, cxpoint2, size)

RETURN child1, child2

- `fill_child(child, parent, start, size)`

Description: Complete child individual by copying genes from parent individual, ensuring that each gene is copied only once

Parameters:

child: Child individual

parent: Parent individual

start: Position in child individual where copying begins

size: Size of individual

Returns: None

Pseudocode:

SET pos to start

FOR each index i in range(size):

IF gene at index i of parent is not already in child:

WHILE gene at position pos in child is not None:

INCREMENT pos by 1, wrapping around if necessary

COPY gene at index i of parent to position pos in child

- `mutate(solution, mutpb)`

Description: Randomly swap elements within the solution based on mutation probability

Parameters:

solution: Individual

mutpb: Mutation probability

Returns: None

Pseudocode:

FOR each index i in range(length of solution):

IF random number between 0 and 1 < mutpb:

randomly select position j within the solution

SWAP elements at positions i and j in the solution

- `genetic_algorithm(customers, vehicles, population_size, generations, cxpb, mutpb)`

Description: Iteratively evolve a population of solutions over a number of generations using selection, crossover, and mutation operations

Parameters:

customers: DataFrame containing customer information
vehicles: DataFrame containing vehicle information
population_size: Size of population
generations: Number of generations
cxpb: Crossover probability
mutpb: Mutation probability

Returns:

best_solution: List of customer indices
best_fitness: Fitness value of best solution

Pseudocode:

INITIALIZE population as a list of initial solutions

INITIALIZE fitnesses as a list of fitness values corresponding to each solution

FOR each generation in range(generations):

 SELECT individuals from the population based on their fitness

 GENERATE the next population by applying crossover and mutation operations

 EVALUATE the fitness of the next population

IDENTIFY the best solution and its fitness value from the final population

RETURN the best solution and its fitness value

- `hyperparameter_tuning_ga(customers, vehicles, population_size_list, generations_list, cxpb_list, mutpb_list)`

Description: Find the best combination of hyperparameters

Parameters:

customers: DataFrame containing customer information

vehicles: DataFrame containing vehicle information

population_size_list: List of population sizes

generations_list: List of number of generations

cxpb_list: List of crossover probability

mutpb_list: List of mutation probability

Returns:

best_hyperparams: Best combination of hyperparameters

best_solution: Best solution

best_distance: Total distance of best solution

best_cost: Total cost of best solution

Pseudocode:

INITIALIZE best_hyperparams, best_solution, best_distance, and best_cost

SET best_distance and best_cost to infinity

FOR each population_size in population_size_list:

 FOR each generations in generations_list:

 FOR each cxpb in cxpb_list:

 FOR each mutpb in mutpb_list:

 APPLY genetic algorithm with the current hyperparameters

 UPDATE best_hyperparams, best_solution, best_distance, and best_cost if a better solution is found

 PRINT hyperparameters and performance of the current iteration

RETURN best_hyperparams, best_solution, best_distance, and best_cost

- Call `hyperparameter_tuning_ga`

```

population_size: 150, generations: 150, cxpb: 0.6, mutpb: 0.1 -> Total Distance: 115.932 km, Total Cost: RM 152.66
population_size: 150, generations: 150, cxpb: 0.6, mutpb: 0.2 -> Total Distance: 118.923 km, Total Cost: RM 153.58
population_size: 150, generations: 150, cxpb: 0.6, mutpb: 0.3 -> Total Distance: 122.152 km, Total Cost: RM 158.90
population_size: 150, generations: 150, cxpb: 0.7, mutpb: 0.1 -> Total Distance: 110.489 km, Total Cost: RM 143.37
population_size: 150, generations: 150, cxpb: 0.7, mutpb: 0.2 -> Total Distance: 122.344 km, Total Cost: RM 157.14
population_size: 150, generations: 150, cxpb: 0.7, mutpb: 0.3 -> Total Distance: 121.856 km, Total Cost: RM 160.23
population_size: 150, generations: 150, cxpb: 0.8, mutpb: 0.1 -> Total Distance: 111.557 km, Total Cost: RM 148.43
population_size: 150, generations: 150, cxpb: 0.8, mutpb: 0.2 -> Total Distance: 111.186 km, Total Cost: RM 142.30
population_size: 150, generations: 150, cxpb: 0.8, mutpb: 0.3 -> Total Distance: 128.112 km, Total Cost: RM 162.80

Best Hyperparameters: population_size = 150, generations = 100, cxpb = 0.6, mutpb = 0.1

```

Figure 3: Iterations to find the best combination of hyperparameters for GA

- Output final solution

```

Vehicle 1 (Type A):
Round Trip Distance: 40.557 km, Cost: RM 48.67, Demand: 22
Route: Depot -> C7 (6.508 km) -> C10 (6.018 km) -> C9 (5.060 km) -> C8 (11.358 km) -> Depot (11.612 km)

Vehicle 2 (Type B):
Round Trip Distance: 30.229 km, Cost: RM 45.34, Demand: 26
Route: Depot -> C5 (10.483 km) -> C1 (4.839 km) -> C2 (5.012 km) -> C6 (1.834 km) -> Depot (8.061 km)

Vehicle 3 (Type A):
Round Trip Distance: 32.812 km, Cost: RM 39.37, Demand: 9
Route: Depot -> C4 (14.142 km) -> C3 (8.323 km) -> Depot (10.347 km)

Total Distance = 103.597 km
Total Cost = RM 133.39

```

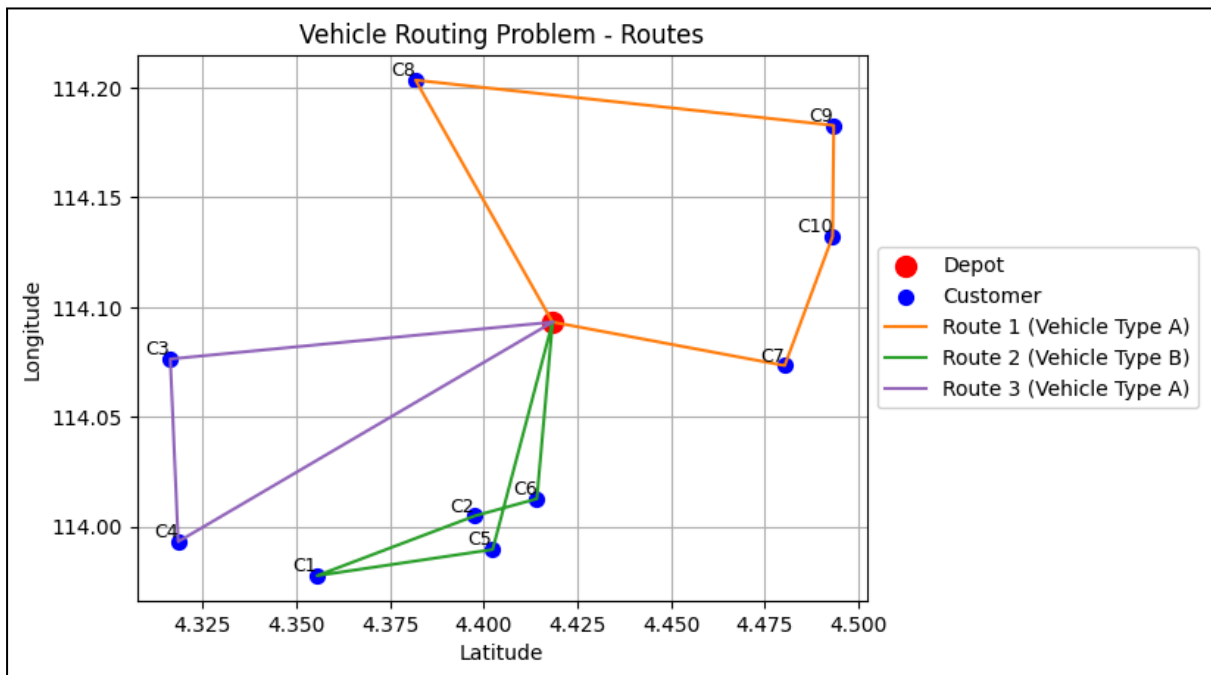


Figure 4: Final output of solution and plot of routes for GA

Variable Neighbourhood Search Code Segment:

- `local_search(solution)`

Description: Iteratively swaps consecutive customers in solution to find better solution

Parameters:

solution: Initial solution

Returns:

best_solution: Improved solution

best_distance: Total distance of improved solution

best_cost: Total cost of improved solution

Pseudocode:

SET best_solution to a copy of the initial solution

SET best_distance and best_cost to the evaluation of the initial solution

FOR each index i in range(length of solution):

 FOR each index j in range(i + 1, length of solution):

 CREATE a new solution by swapping two consecutive customers

 EVALUATE the distance and cost of the new solution

 IF the cost of the new solution is lower than the best cost:

 UPDATE best_solution, best_distance, and best_cost with the new solution

RETURN best_solution, best_distance, and best_cost

- `shake(solution, k)`

Description: Randomly swaps customers in solution to find better solution

Parameters:

solution: Initial solution

k: Number of random swaps

Returns:

new_solution: Solution after k random swaps

Pseudocode:

SET new_solution as a copy of the initial solution

FOR each shake in range(k):

 SELECT two random indices i and j from the range of solution length

 SWAP the elements at indices i and j in the new solution

RETURN new_solution

- `variable_neighbourhood_search(customers, vehicles, max_iterations, k_max)`

Description: Explore different neighbourhoods of solutions by iteratively applying local search and shakes

Parameters:

customers: DataFrame containing customer information

vehicles: DataFrame containing vehicle information

max_iterations: Maximum number of iterations

k_max: Maximum level of neighbourhood

Returns:

best_solution: Best solution

best_distance: Total distance of best solution

best_cost: Total cost of best solution

Pseudocode:

SET best_solution to an initial solution created by create_initial_solution()

SET best_distance and best_cost to the evaluation of the initial solution

SET iteration to 0

WHILE iteration < max_iterations:

 SET k to 1

 WHILE k <= k_max:

 CREATE a new solution by shake function with k

 IMPROVE the new solution with local search

 IF the cost of the new solution is lower than the best cost:

 UPDATE best_solution, best_distance, and best_cost with the new solution

 SET k to 1 to restart search beginning with new best solution

 ELSE:

 INCREMENT k by 1

 INCREMENT iteration by 1

RETURN best_solution, best_distance, and best_cost

- `hyperparameter_tuning_vns(customers, vehicles, max_iterations_list, k_max_list)`

Description: Find the best combination of hyperparameters

Parameters:

customers: DataFrame containing customer information
vehicles: DataFrame containing vehicle information
max_iterations_list: List of maximum number of iterations
k_max_list: List of maximum level of neighbourhood

Returns:

best_hyperparams: Best combination of hyperparameters
best_solution: Best solution
best_distance: Total distance of best solution
best_cost: Total cost of best solution

Pseudocode:

```
INITIALIZE best_hyperparams, best_solution, best_distance, and best_cost
SET best_distance and best_cost to infinity

FOR each max_iterations in max_iterations_list:
    FOR each k_max in k_max_list:
        APPLY variable neighbourhood search with the current hyperparameters
        UPDATE best_hyperparams, best_solution, best_distance, and best_cost if a better
solution is found
        PRINT hyperparameters and performance of the current iteration

RETURN best_hyperparams, best_solution, best_distance, and best_cost
```


- Call `hyperparameter_tuning_vns`

```
max_iterations: 50, k_max: 3 -> Total Distance: 110.343 km, Total Cost: RM 141.28
max_iterations: 50, k_max: 5 -> Total Distance: 101.240 km, Total Cost: RM 132.95
max_iterations: 50, k_max: 7 -> Total Distance: 101.240 km, Total Cost: RM 132.95
max_iterations: 100, k_max: 3 -> Total Distance: 94.883 km, Total Cost: RM 125.32
max_iterations: 100, k_max: 5 -> Total Distance: 103.380 km, Total Cost: RM 140.13
max_iterations: 100, k_max: 7 -> Total Distance: 107.014 km, Total Cost: RM 138.72
max_iterations: 150, k_max: 3 -> Total Distance: 103.380 km, Total Cost: RM 140.13
max_iterations: 150, k_max: 5 -> Total Distance: 107.014 km, Total Cost: RM 138.72
max_iterations: 150, k_max: 7 -> Total Distance: 105.343 km, Total Cost: RM 135.04

Best Hyperparameters: max_iterations = 100, k_max = 3
```

Figure 5: Iterations to find the best combination of hyperparameters for VNS

- Output final solution

Vehicle 1 (Type A):
Round Trip Distance: 40.557 km, Cost: RM 48.67, Demand: 22
Route: Depot -> C8 (11.612 km) -> C9 (11.358 km) -> C10 (5.060 km) -> C7 (6.018 km) -> Depot (6.508 km)

Vehicle 2 (Type B):
Round Trip Distance: 38.205 km, Cost: RM 57.31, Demand: 27
Route: Depot -> C2 (9.072 km) -> C5 (1.604 km) -> C1 (4.839 km) -> C4 (4.021 km) -> C3 (8.323 km) -> Depot (10.347 km)

Vehicle 3 (Type A):
Round Trip Distance: 16.122 km, Cost: RM 19.35, Demand: 8
Route: Depot -> C6 (8.061 km) -> Depot (8.061 km)

Total Distance = 94.883 km
Total Cost = RM 125.32

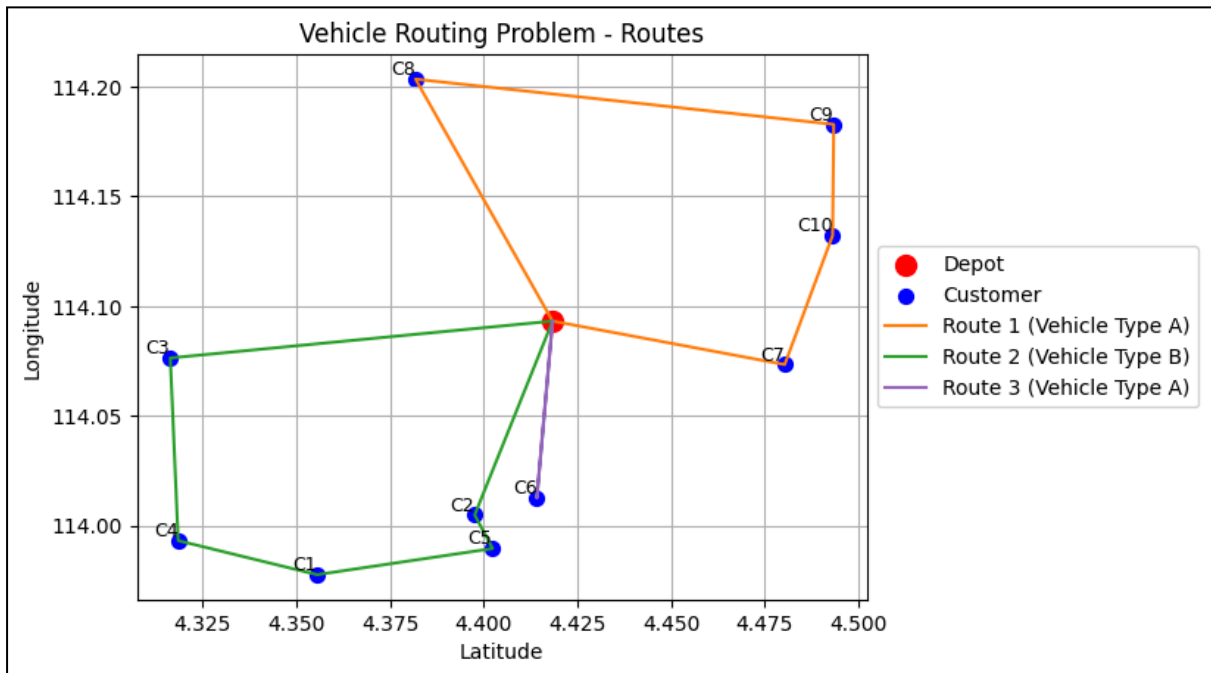


Figure 6: Final output of solution and plot of routes for VNS

Results and Analysis

Results Comparison

Table 1: Results Comparison

Algorithm	Total Distance	Total Cost
Genetic Algorithm	103.597 km	RM 133.39
Variable Neighbourhood Search	94.883 km	RM 125.32

According to the data in Table 1, VNS showed superior performance, demonstrating lower total distance and total cost. Therefore, it has been selected as the optimal algorithm.

Test Case

Added random new customer:

```
new_customer_data = {  
    'Customer': [11],  
    'Latitude': [4.3625],  
    'Longitude': [114.1500],  
    'Demand': [3] }
```

Results:

Vehicle 1 (Type A):
Round Trip Distance: 51.475 km, Cost: RM 61.77, Demand: 23
Route: Depot -> C1 (13.152 km) -> C4 (4.021 km) -> C3 (8.323 km) -> C11 (8.690 km) -> C8 (5.678 km) -> Depot (11.612 km)

Vehicle 2 (Type B):
Round Trip Distance: 28.759 km, Cost: RM 43.14, Demand: 24
Route: Depot -> C7 (6.508 km) -> C6 (8.982 km) -> C5 (2.594 km) -> C2 (1.604 km) -> Depot (9.072 km)

Vehicle 3 (Type A):
Round Trip Distance: 25.187 km, Cost: RM 30.22, Demand: 13
Route: Depot -> C10 (8.436 km) -> C9 (5.060 km) -> Depot (11.691 km)

Total Distance = 105.421 km
Total Cost = RM 135.13

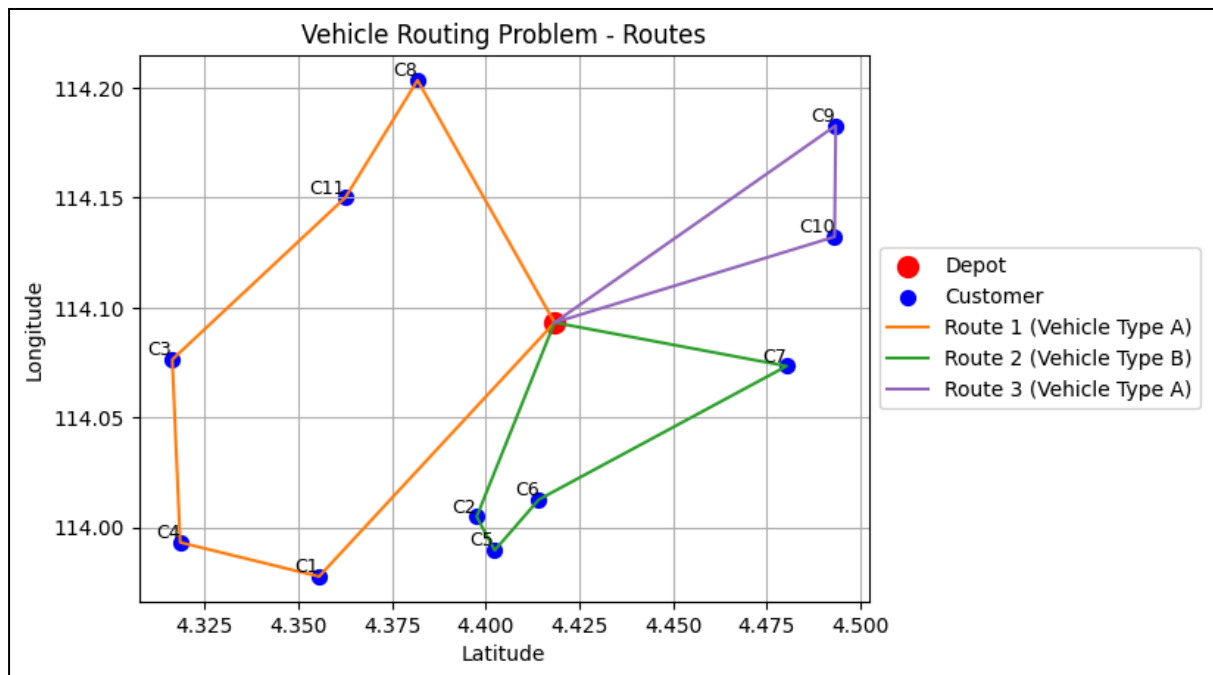


Figure 7: Final output of solution and plot of routes for VNS (Test Case)

Challenges, Limitation and Future Work

Challenges

Variable Neighborhood Search (VNS)

- **Local Optima:** Due to the complex and multi-dimensional solution space created by constraints like different vehicle types, capacities, and customer demands. VNS often get trapped in locally optimal solutions, where small, local changes fail to yield better results, leading to stagnation. As a result, the algorithm repeatedly explores the same neighbourhoods without making progress, reducing overall quality and efficiency.

Genetic Algorithm (GA)

- **Parameter Sensitivity:** Parameters such as generation, population size, crossover rate, and mutation rate heavily influence performance. Poorly chosen parameters can lead to premature convergence, where the algorithm gets stuck in suboptimal solutions, or excessive diversification, where the search becomes inefficient. Tuning these parameters to achieve optimal performance is time-consuming.

Limitation

Computational Intensity: VNS requires extensive computational resources due to its iterative exploration of multiple neighbourhoods, each involving multiple local searches to find improvements. Similarly, GA demands substantial computational power to evaluate, crossover, and mutate numerous candidate solutions across many generations. Both methods can become time-consuming and resource-intensive, particularly for large-scale problems, limiting their practical applicability in real-time scenarios.

Future Work

Hybrid Approach: By combining the strengths of different optimization algorithms, such as integrating the global search capability of GA with the local refinement efficiency of VNS, it is possible to enhance solution quality and robustness. Hybrid methods can balance exploration and exploitation more effectively, escape local optima, and adaptively adjust search strategies based on problem characteristics.

Conclusion

HFVRP is inherently complex due to the diverse characteristics of vehicles and customers. Numerous algorithms have been suggested to address HFVRP. Choosing the most optimal algorithm depends on factors such as problem size, computational resources, and desired solution accuracy. Overall, the problem-solving process for HFVRP highlights the need for continuous refinement and innovation in optimization techniques to meet the evolving demands of complex logistical challenges.

References

Genetic Algorithms. (2024, March 8). GeeksforGeeks. Retrieved May 30, 2024, from

<https://www.geeksforgeeks.org/genetic-algorithms/>

Konstantakopoulos, G. D., Gayialis, S. P., & Kechagias, E. P. (2020). Vehicle routing problem and related algorithms for logistics distribution: a literature review and classification.

Operational Research, 22(3), 2033–2062.

<https://doi.org/10.1007/s12351-020-00600-7>

Mrabet, A. (2023, October 3). Solving Travelling Salesman Problem With Variable

Neighborhood Search. *Medium*. Retrieved May 30, 2024, from

<https://medium.com/@mrabetahmed/solving-travelling-salesman-problem-with-variable-neighborhood-search-ba89e4e7507>