Introduction

The report discusses the implementation of Ant Colony Optimization (ACO) algorithm to solve a Vehicle routing Problem(VRP). The problem involves optimizing the routing of a fleet of vehicles to efficiently deliver goods to various customer locations while minimizing the total cost.

Problem Definition

The problem consists of a single depot, 10 customers with specific demands, and two types of vehicles with different capacities (and costs per kilometer. The objective is to find the most cost-effective routes that satisfy all customer demands while respecting vehicle capacity constraints.

Implementation

The decision to use ACO and its specific implementation was inspired by this research paper demonstrating ACO's effectiveness in solving vehicle routing problems with particular constraints, such as customers with varied demands and vehicles with limited capacities.

Algorithm Overview

In ACO for the VRP, artificial ants construct solutions by iteratively moving from one customer to another, depositing pheromones along their paths. The concentration of pheromone on an edge represents the desirability of that edge. Ants probabilistically choose the next customer to visit based on the pheromone levels and heuristic information (e.g., cost to travel) of the neighboring customers.

The mathematics formula of **edge selection** is represented by:

$$p_{xy}^{k} = \frac{(\tau_{xy}^{\alpha})(\eta_{xy}^{\beta})}{\sum_{z \in \text{allowed}_y} (\tau_{xz}^{\alpha})(\eta_{xz}^{\beta})}$$

- τ_{xy} is the amount of pheromone deposited for transition from state x to y.
- η_{xy} is the desirability of state transition from x to y, with the formula of $\frac{1}{\text{distance (x to y)}}$
- $\propto$ is a parameter to control the influence of τ_{xy}.
- β is a parameter to control the influence of η_{xy}.
- k is a constant which represents the number of artificial ants.
- The denominator is the sum of the product of pheromone and heuristics value for all possible path from node x.

After computing the probabilities, the ant uses a random selection mechanism to probabilistically choose the next node. A random number is generated between 0 and 1. The next node is selected based on where this random number falls within the cumulative probability intervals.

The **Pheromone Update** is represented by:

$$\tau_{xy} \leftarrow (1-\rho)\tau_{xy} + \sum_{k}^{m} \Delta\tau_{xy}^{k}$$

$$\Delta\tau_{xy}^{k} = \begin{cases} Q/L_k & \text{if ant } k \text{ uses curve } xy \text{ in its tour} \\ 0 & \text{otherwise} \end{cases}$$

- **ρ** is the evaporation rate that controls the pheromone levels to prevent premature convergence.
- **Q** is a constant that determines the amount of pheromone deposited.
- L_k is the length of the path constructed by ant k.

Specific Implementations

The parameters were set as follows:

```
# ACO parameters
NUM_ANTS = 50

MAX_ITERATIONS = 1000
ALPHA = 1  # Pheromone importance
BETA = 1   # Heuristic information importance
RHO = 0.2  # Evaporation rate
Q = 5    # Pheromone deposit factor
INITIAL_PHEROMONE = 1.0
```

The attributes defined in the **constructor**:

```
class ACO:

    def __init__(self):
        self.num_nodes = len(CUSTOMERS) + 1

        self.distances = calculate_distances() # Matrix representation of the distance

        self.pheromones = np.full((self.num_nodes, self.num_nodes), INITIAL_PHEROMONE) # Matrix representation of pheromone

        self.best_solution = None
        self.best_cost = float('inf')
        self.total_demand = sum(customer[2] for customer in CUSTOMERS)
```

Most of the attributes are self-explanatory:

- **'self.distances'** stores the distances between all pairs of customers and the depot in a matrix form with dimensions 11 x 11.

- **'self.pheromones'** keeps track of the pheromone levels on every possible path and was initialized to 1.0 in the constructor.

- **'self.total_demand'** tracks the remaining customer demand to complete the task. This attribute is used for vehicle selection for each new route.

Distance and Pheromones Matrices

The distance and pheromone matrices are structured as symmetric N x N matrix, where N is the number of customers plus the depot. The structure is as follows:

	Customer 1	Customer 2	Customer 3	Customer 4	...	Depot
Customer 1						
Customer 2						
Customer 3						
Customer 4						
...						
Depot						

- The last row and column of each matrix correspond to the depot.
- As ants traverse the paths between locations, both the distance calculations and pheromone levels are updated in their respective matrices.

Vehicle Selection

```python
VEHICLES = [
    {"type": "A", "capacity": 25, "cost_per_km": 1.2},
    {"type": "B", "capacity": 30, "cost_per_km": 1.5}
]


def pick_vehicle(remaining_demand, vehicles):

    max_capacity_vehicle = max(vehicles, key=lambda x: x["capacity"])

    # If the remaining demand is more than the vehicle with the largest capacity, deploy the vehicle with largest capacity for the next route
    if remaining_demand > max_capacity_vehicle["capacity"]:
        return max_capacity_vehicle

    else:
        suitable_vehicles = [vehicle for vehicle in vehicles if vehicle["capacity"] >= remaining_demand]

        if suitable_vehicles:

    #If the remaining demand is less than the capacity of the largest vehicle,
    #deploy the vehicle with the smallest capacity that can meet the remaining demand to avoid underutilization.

            return min(suitable_vehicles, key=lambda x: x["capacity"])
        else:
            return None
```

The algorithm uses a simple vehicle selection method that chooses the smallest vehicle capable of handling the remaining demand.

Framework

The following pseudocode summarized the structure of the algorithm:

```
procedure ACO do:
    for _ in range(MAX_ITERATIONS = 1000):
        compute_solutions_by_every_ant()
        update_pheromone_on_each_edge()
        curr_best_cost, curr_best_path= optimal(compute_solutions_for_every_ant())

            if current_best_cost < self.best_cost:
                self.best_cost = curr_best_cost
                self.best_solution = curr_best_path

            if ( best cost remains unchanged for the last 50 iterations):
                Execute early stopping and break the for loop
            else:
                Continue

    return self.best_cost, self_best_solution
```

- The 'compute_solutions_by_every_ant()' function manages the process of edge selection for each ant.
- A key feature of this function is its consideration of the current vehicle's remaining capacity when calculating the probability of selecting each potential path.
- This ensures that the algorithm complies with vehicle capacity constraints while constructing routes.

Output

```
Iteration 21: Best cost = 136.32
Iteration 22: Best cost = 136.32
Iteration 23: Best cost = 136.32
Iteration 24: Best cost = 136.32
Iteration 25: Best cost = 136.32
...
Iteration 75: Best cost = 133.94
Iteration 76: Best cost = 133.94
Iteration 77: Best cost = 133.94
Early stopping at iteration 77: No improvement in the last 50 iterations.
```

```
Vehicle 1 (Type B):
Round Trip Distance: 51.09 km, Cost: RM 76.64, Demand: 30
Depot -> 5 -> 6 -> 9 -> 8 -> 7 -> Depot

Vehicle 2 (Type B):
Round Trip Distance: 38.20 km, Cost: RM 57.31, Demand: 27
Depot -> 2 -> 3 -> 0 -> 4 -> 1 -> Depot

Total Distance = 89.30 km
Total Cost = RM 133.94
```

- The algorithm successfully satisfies all hard constraints specified in the problem.
- Regarding the soft constraint of cost minimization, extensive parameter tuning was performed.
- The best solution converged to a total cost of RM 133.94.
- Notably, the early stopping mechanism typically activates within the first 100 iterations, indicating rapid convergence to a near-optimal solution.