**Capacitated Vehicle Routing Problem (CVRP) Solution Documentation**

1. **Background:**

You are a logistics manager for a delivery company tasked with optimizing the routing of a fleet of vehicles to efficiently deliver goods to various customer locations. Your goal is to optimize the delivery cost while ensuring that all delivery locations are visited and all demands are met.

1. **Task:**

Your task is to develop an algorithm or program that attempt to find the best route for a fleet of vehicles of different types so that the deliveries are completed at the lowest cost while satisfying all hard constraints. Ensure that your solution is scalable to support a larger number of customers beyond the provided test data.

1. **Requirements:**

Hard Constraint:

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

Soft Constraint:

* Minimize 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:
* Distance in km = 100 \* √((Longitude2-Longitude1)^2 + (Latitude2-Latitude1)^2)

Test Data: One Depot, 10 Customer, 2 types of vehicles.

Depot: (Latitude = 4.4184, Longitude = 114.0932)

|  |  |  |  |
| --- | --- | --- | --- |
| Customer | Latitude | Longitude | Demand |
| 1 | 4.3555 | 113.9777 | 5 |
| 2 | 4.3976 | 114.0049 | 8 |
| 3 | 4.3163 | 114.0764 | 3 |
| 4 | 4.3184 | 113.9932 | 6 |
| 5 | 4.4024 | 113.9896 | 5 |
| 6 | 4.4142 | 114.0127 | 8 |
| 7 | 4.4804 | 114.0734 | 3 |
| 8 | 4.3818 | 114.2034 | 6 |
| 9 | 4.4935 | 114.1828 | 5 |
| 10 | 4.4932 | 114.1322 | 8 |

|  |  |  |
| --- | --- | --- |
| Vehicle | Capacity | Cost |
| Type A | 25 | Rm1.2 per km |
| Type B | 30 | Rm1.5 per km |

1. **Solution Approach**

OR-Tools is an open-source software suite for optimization, tuned for tackling the world's toughest problems in vehicle routing, flows, integer and linear programming, and constraint programming.

Key features of OR-Tools include:

* Sophisticated Algorithms: OR-Tools uses the most advanced algorithms like constraint programming, linear programming, and graph algorithms to deal with different optimization issues.
* Efficiency: OR-Tools provides powerful optimization engines capable of addressing multiple constraints and utilizing advanced algorithms and data structures to minimize computation time and resource usage.
* Versatility: OR-Tools has a number of solvers that can be made to fit a specific problem type, thus giving the ability to be used in different optimization tasks.
* Scalability: OR-Tools is an ideal tool for large-scale optimization problems, it can scale well for complex routing scenarios and huge datasets.
* Open Source: OR-Tools is an open source, mature and well-tested framework created by Google. It provides comprehensive documentation, support, and community resources that are easy to implement and debug the complex routing models.

1. **Implementation Details**

OR-Tools Implementation

* + - Data Model Creation:

The create\_data\_model() function initializes the problem data, including:

1. the number and types of vehicles

# Define the number of vehicles of each type

    num\_type\_a\_vehicles = 2

    num\_type\_b\_vehicles = 1

1. vehicle capacities and costs

# Create vehicles based on the specified numbers

    vehicles = []

    for \_ in range(num\_type\_a\_vehicles):

        vehicles.append(Vehicle(type="Type A", capacity=25, cost\_per\_km=1.2))

    for \_ in range(num\_type\_b\_vehicles):

        vehicles.append(Vehicle(type="Type B", capacity=30, cost\_per\_km=1.5))

1. depot location, customer locations and demands

data["locations"] = [

        (4.4184, 114.0932), # Depot

        (4.3555, 113.9777), # Customers

        (4.3976, 114.0049),

        (4.3163, 114.0764),

        (4.3184, 113.9932),

        (4.4024, 113.9896),

        (4.4142, 114.0127),

        (4.4804, 114.0734),

        (4.3818, 114.2034),

        (4.4935, 114.1828),

        (4.4932, 114.1322),

    ]

    data["demands"] = [0, 5, 8, 3, 6, 5, 8, 3, 6, 5, 8] \* len(vehicles)

This function creates the problem instance according to the given parameters and returns a dictionary which has all the required information.

* + - Distance Matrix Computation:

The compute\_euclidean\_distance\_matrix() function calculates the Euclidean distance between all pairs of locations and constructs a distance matrix. This matrix represents the travel costs between each pair of nodes, which is essential for routing optimization.

def compute\_euclidean\_distance\_matrix(locations):

    """Creates callback to return distance between points."""

    distance\_matrix = [[0 for \_ in locations] for \_ in locations]

    for from\_counter, from\_node in enumerate(locations):

        for to\_counter, to\_node in enumerate(locations):

            if from\_counter == to\_counter:

                distance\_matrix[from\_counter][to\_counter] = 0

            else:

                # Calculate Euclidean distance and scale it up

                distance = math.hypot(

                    (from\_node[0] - to\_node[0]),

                    (from\_node[1] - to\_node[1])

                )

                scaled\_distance = distance \* 100000

                distance\_matrix[from\_counter][to\_counter] = int(scaled\_distance)

    return distance\_matrix

OR tools cannot deal with float numbers if the distance is directly counted by the Euclidean distance formula. To deal with this issue, the Euclidean distance is scaled up to change the distance from a floating-point number to an integer value that is easy to manage in the routing optimization problem.

* + - Solution Printing:

The print\_solution() function prints the optimized solution on the console.

It calculates and displays:

1. the total distance and total cost

# Calculate the total distance and total cost

    for vehicle\_id in range(data["num\_vehicles"]):

        vehicle\_type = data["vehicle\_types"][vehicle\_id]

        cost\_per\_km = data["vehicle\_costs"][vehicle\_id]

        index = routing.Start(vehicle\_id)

        route\_distance = 0

        route\_demand = 0

        route = []

        while not routing.IsEnd(index):

            node\_index = manager.IndexToNode(index)

            route\_demand += data["demands"][node\_index]

            previous\_index = index

            index = solution.Value(routing.NextVar(index))

            route\_distance += routing.GetArcCostForVehicle(previous\_index, index, vehicle\_id)

            route.append(node\_index)

        route\_cost = (route\_distance / 1000) \* cost\_per\_km

        total\_distance += route\_distance

        total\_cost += route\_cost

        vehicle\_routes[vehicle\_id] = {"type": vehicle\_type, "distance": route\_distance, "cost": route\_cost, "demand": route\_demand - data["demands"][data["depot"]], "route": route}

1. detailed route information for each vehicle, including the sequence of customer visits, round trip distance, cost, and demand satisfied.

# Print the details for each vehicle

    for vehicle\_id, route\_info in vehicle\_routes.items():

        print(f"Vehicle {vehicle\_id + 1} ({route\_info['type']}):")

        print(f"Round Trip Distance: {route\_info['distance']/1000:.3f} km, Cost: RM {route\_info['cost']:.2f}, Demand: {route\_info['demand']}")

        print("Depot -> ", end="")

        for node\_index in route\_info["route"]:

            if node\_index != 0:

                distance = data["distance\_matrix"][route\_info["route"][route\_info["route"].index(node\_index) - 1]][node\_index] / 1000

                print(f"C{node\_index} ({distance:.3f} km) -> ", end="")

        distance\_back\_to\_depot = data["distance\_matrix"][route\_info["route"][-1]][0] / 1000

        print(f"Depot ({distance\_back\_to\_depot:.3f} km)")

Note\*\*distance or route /1000 is to convert back the scaling up distance to km.

* + - Route Plotting:

The plot\_routes() function visualizes the optimized routes on a graph using Matplotlib.

def plot\_routes(data, manager, routing, solution):

    """Plots the routes on a graph."""

    color\_cycle = cycle(plt.rcParams['axes.prop\_cycle'].by\_key()['color'])

    depot\_location = data["locations"][data["depot"]]

    plt.scatter(depot\_location[1], depot\_location[0], c='black', marker='s', label='Depot')

    for vehicle\_id in range(data["num\_vehicles"]):

        index = routing.Start(vehicle\_id)

        route\_x = []

        route\_y = []

        while not routing.IsEnd(index):

            node\_index = manager.IndexToNode(index)

            location = data["locations"][node\_index]

            route\_x.append(location[1])

            route\_y.append(location[0])

            plt.text(location[1], location[0], f'C{node\_index}')

            index = solution.Value(routing.NextVar(index))

        route\_x.append(depot\_location[1])

        route\_y.append(depot\_location[0])

        vehicle\_color = next(color\_cycle)

        plt.plot(route\_x, route\_y, '-o', color=vehicle\_color, label=f'Vehicle {data["vehicle\_types"][vehicle\_id]}')

    plt.xlabel('Longitude')

    plt.ylabel('Latitude')

    plt.title('Vehicle Routing Problem')

    plt.legend()

    plt.show()

* + - Routing Model Setup:

The main() function sets up the routing model using the OR-Tools framework. It creates:

1. Routing index manager

* It takes the length of the distance matrix, the number of vehicles, and the index of the depot as input arguments.
* This manager provides methods to convert between routing variable indices and node indices.

# Create the routing index manager.

    manager = pywrapcp.RoutingIndexManager(

        len(data["distance\_matrix"]), data["num\_vehicles"], data["depot"]

    )

1. Initializes the routing model

* It is initialized with the routing index manager created in the previous step.

# Create Routing Model.

    routing = pywrapcp.RoutingModel(manager)

1. Registers transit and demand callbacks

* Callbacks are functions used to provide dynamic data to the routing model during the optimization process.
* A transit callback is registered to provide the distance between two nodes.

# Create and register a transit callback.

    def distance\_callback(from\_index, to\_index):

        """Returns the distance between the two nodes."""

        # Convert from routing variable Index to distance matrix NodeIndex.

        from\_node = manager.IndexToNode(from\_index)

        to\_node = manager.IndexToNode(to\_index)

        return data["distance\_matrix"][from\_node][to\_node]

    transit\_callback\_index = routing.RegisterTransitCallback(distance\_callback)

1. Adds capacity constraints

* Capacity constraints ensure that the total demand served by each vehicle does not exceed its capacity.
* A demand callback is registered to provide the demand associated with each node.

# Add Capacity constraint.

    def demand\_callback(from\_index):

        """Returns the demand of the node."""

        # Convert from routing variable Index to demands NodeIndex.

        from\_node = manager.IndexToNode(from\_index)

        return data["demands"][from\_node]

demand\_callback\_index = routing.RegisterUnaryTransitCallback(demand\_callback)

* The AddDimensionWithVehicleCapacity method is called to add a dimension representing vehicle capacities to the routing model.
* This dimension enforces the capacity constraint for each vehicle's route.

routing.AddDimensionWithVehicleCapacity(

        demand\_callback\_index,

        0,  # null capacity slack

        data["vehicle\_capacities"],  # vehicle maximum capacities

        True,  # start cumul to zero

        "Capacity",

    )

1. Sets the first solution heuristic for optimization.

* The first solution strategy "PATH\_CHEAPEST\_ARC" strategy is chosen, which starts by selecting the edge (or arc) between two nodes that has the lowest cost. This cost typically represents the distance or travel time between the nodes.
* Additionally, a local search metaheuristic (GUIDED\_LOCAL\_SEARCH) is specified to enhance the search process.
* A time limit is set to control the duration of the optimization process. In this project time set to 1 second.

# Setting first solution heuristic.

    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.FromSeconds(1)

* + - Routing Optimization:

The routing model is solved using the routing.SolveWithParameters() method, which applies the specified search parameters to find an optimized solution to the CVRP.

# Solve the problem.

    solution = routing.SolveWithParameters(search\_parameters)

If a solution is found, the print\_solution() and plot\_routes() functions are called to print the solution details and visualize the optimized routes, respectively.

# Print solution on console.

    if solution:

        print\_solution(data, manager, routing, solution)

        plot\_routes(data, manager, routing, solution)

1. **Output for 2 Type A & 1 Type B vehicles**

Total Distance = 94.875 km

Total Cost = RM 125.31

Vehicle 1 (Type A):

Round Trip Distance: 40.554 km, Cost: RM 48.66, Demand: 22

Depot -> C7 (6.508 km) -> C10 (6.017 km) -> C9 (5.060 km) -> C8 (11.358 km) -> Depot (11.611 km)

Vehicle 2 (Type A):

Round Trip Distance: 16.120 km, Cost: RM 19.34, Demand: 8

Depot -> C6 (8.060 km) -> Depot (8.060 km)

Vehicle 3 (Type B):

Round Trip Distance: 38.201 km, Cost: RM 57.30, Demand: 27

Depot -> C2 (9.071 km) -> C5 (1.603 km) -> C1 (4.838 km) -> C4 (4.020 km) -> C3 (8.322 km) -> Depot (10.347 km)

A screenshot of a graph

Description automatically generated

1. **Future Work and Improvements**
2. User Interface Development: Create a user-friendly interface or visualization tool that can be used by non-programmers to input data, view the solutions, and analyse the routing outcomes, thus making the tool more accessible to the users.
3. Machine Learning Integration: The integration of machine learning techniques to determine customer demand patterns, to optimize routing strategies and to dynamically adjust routing decisions based on historical data and real-time information is the way to go.