Lab 3

META HEURISTICS SWARM INTELLIGENCE - CVRP

Kareen Ghattas – 207478728
Hakeem Abu shqara - 207691312

Reading Inputs:

We read the problem details from an input file using read_input(filename). This function reads the dimension, capacity, node coordinates, and demands from the file, and then returns these details for use in the solution.

```python
def read_input(filename):

    with open(filename) as file:
        lines = file.readlines()

    dimension = None
    capacity = None
    node_coord_section = []
    demand_section = []

    for line in lines:
        if line.startswith("DIMENSION"):
            dimension = int(line.split()[-1])
        elif line.startswith("CAPACITY"):
            capacity = int(line.split()[-1])
        elif line.startswith("NODE_COORD_SECTION"):
            node_coord_section = [list(map(int, line.split()[1:])) for line in lines[lines.index(line)+1:lines.index("DEMAND_SECTION\n")]]
        elif line.startswith("DEMAND_SECTION"):
            demand_section = [int(line.split()[-1]) for line in lines[lines.index(line)+1:lines.index("DEPOT_SECTION\n")]]

    print("Dimension:", dimension)
    print("Capacity:", capacity)
    print("Node coordinates:", node_coord_section)
    print("Demands:", demand_section)

    return dimension, capacity, node_coord_section, demand_section
```

Initializing the CVRP Problem

CVRP class stores the problem details and contains methods for calculating the distance matrix, evaluating route costs, displaying and plotting results, generating random solutions, and ensuring capacity constraints are met.

```
class CVRP:
    def __init__(self, dimension, capacity, node_coords, demands):
        self.location = dimension
        self.capacity = capacity
        self.node_coords = node_coords
        self.demands = demands
        self.distance_matrix = self.calculate_distance_matrix()

    def calculate_distance_matrix(self):...

    def euclidean_distance(self, point1, point2):...

    def calculate_total_cost(self, routes):...

    def calculate_route_cost(self, route):...

    def calc_path_cost(self, path):...

    def calc_path_cost_sa(self, path):...

    def display_results(self, routes, start_time, problem_name):...

    def plot_routes(self, routes, problem_name):...

    def random_solution(self):...

    def calculate_route_demand(self, route):...

    def plot_hist(self, algo, problem_name):...
```

In our main program, we prompt the user to select an input file, read the problem details with **read_input**, and then initialize a **CVRP** instance with these details. This sets up the framework for the following optimization stages.

```
def main():
    start_time = time.time()
    random.seed(time.time())

    choose_input = input("Choose input file (1 - 7) \n")
    choose_input = 'input' + choose_input
    dimension, capacity, locations, demands = read_input(choose_input)

    cvrp = CVRP(dimension, capacity, locations, demands)
```

Clustered TSP Algorithm - A Multi-Stage Heuristic:

The Clustered TSP algorithm combines KMeans clustering and the 2-opt algorithm to solve the CVRP problem.

1. **KMeans Clustering**: This step divides cities into a pre-set number of clusters, converting a large problem into smaller, manageable subproblems.
2. **2-opt TSP**: For each cluster, TSP is solved using the 2-opt algorithm. It iteratively optimizes each route, improving the solution quality.

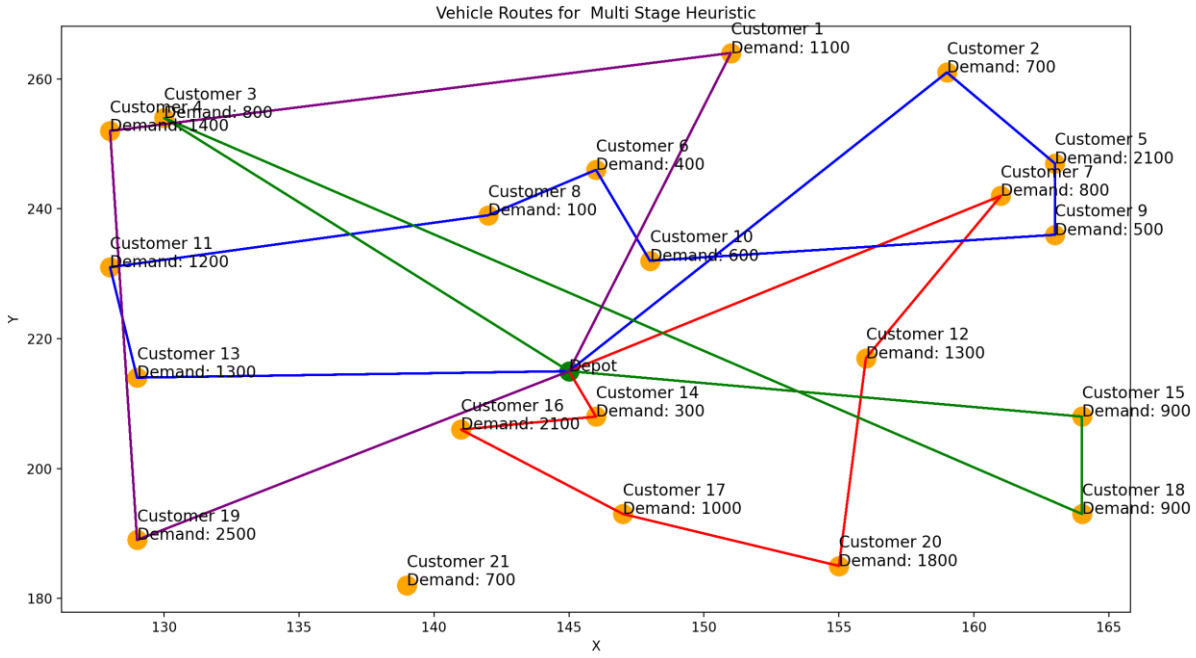
This multi-stage heuristic is part of the **ClusteredTSPSolution** class, which contains methods for clustering (**solve()**), distance calculation (**distance()**), and route optimization (**two_opt()**).

In our experience, this approach consistently provides high-quality solutions. It is computationally efficient and well-suited to problems of larger size, where a traditional, single-stage approach may struggle. However, the choice of the number of clusters must be carefully managed based on the number of vehicles and their capacities. (So, in our code we set it to 4 but you can change it in the main function based on the chosen input.)

```python
class ClusteredTSPSolution:
    def __init__(self, cvrp_problem, n_clusters):
        self.cvrp = cvrp_problem
        self.n_clusters = n_clusters
        self.best_costs = []

    def solve(self):
        # Perform k-means clustering
        kmeans = KMeans(n_clusters=self.n_clusters, random_state=0).fit(self.cvrp.node_coords[1:])
        labels = kmeans.labels_

        # For each cluster, solve the TSP
        routes = []
        for i in range(self.n_clusters):
            cluster_indices = [index for index, label in enumerate(labels) if label == i]
            cluster_indices = [index + 1 for index in cluster_indices]
            cluster_indices = [0] + cluster_indices + [0]
            cluster_points = [self.cvrp.node_coords[index] for index in cluster_indices]
            dist_matrix = distance_matrix(cluster_points, cluster_points)
            cluster_indices_route = list(range(len(cluster_points)))
            cluster_route = self.two_opt(cluster_indices_route, dist_matrix)
            original_indices_route = [cluster_indices[index] for index in cluster_route]
            routes.append(original_indices_route)
        return routes

    def distance(self, route, dist_matrix):
        return sum(dist_matrix[route[i - 1]][route[i]] for i in range(1, len(route)))

    def two_opt(self, route, dist_matrix):
        best = route
        improved = True
        while improved:
```

For input 1:

```
-----------------------------------------------------
Solving the problem using Clustered TSP:
Total cost: 398.63143128809025
Vehicle 1: [0, 12, 15, 14, 16, 13, 0]
Vehicle 2: [0, 10, 8, 6, 3, 4, 11, 0]
Vehicle 3: [0, 17, 19, 21, 20, 18, 0]
Vehicle 4: [0, 1, 2, 5, 7, 9, 0]
Clock Ticks: 0.546875s
Time Elapsed: 2.33398175239563s
-----------------------------------------------------
```



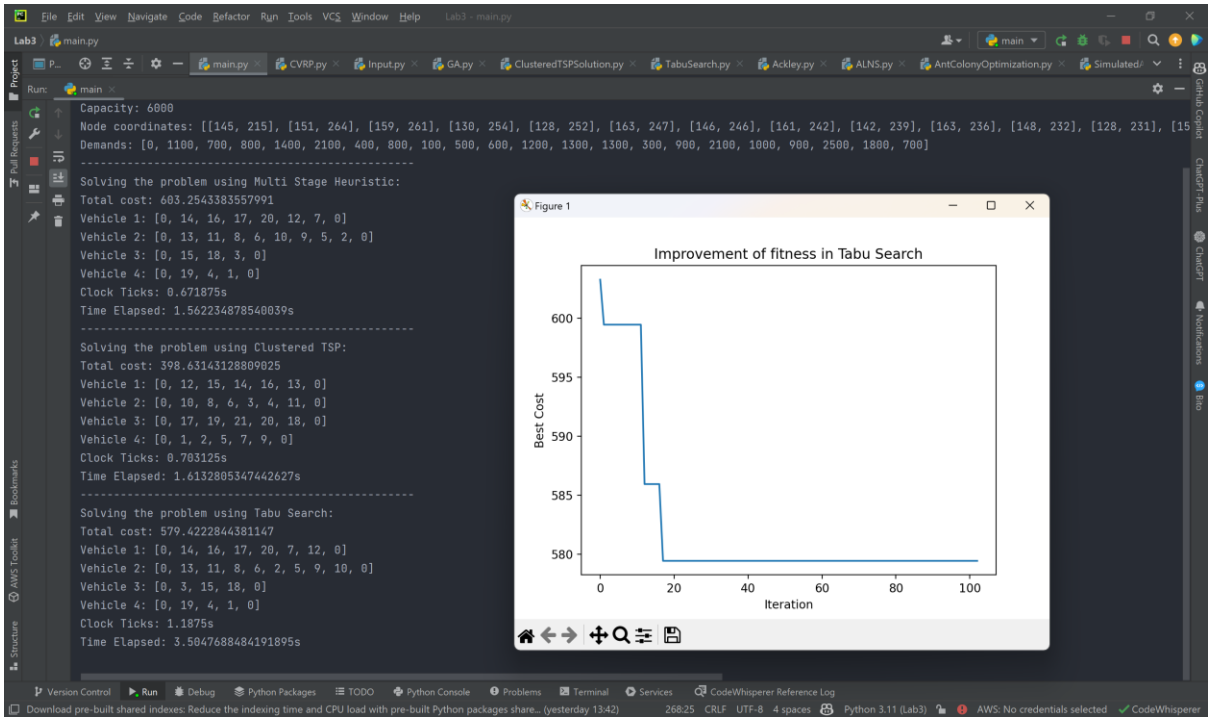
Tabu Search Algorithm:

In our implementation of TS for CVRP includes: The **get_neighborhood()** function implements the 2-opt swap, and the **tabu_search()** function applies the main tabu search mechanism. The **adaptive_tabu_tenure()** function determines the tabu tenure based on the current solution cost, which is an approach to dynamically controlling the balance between exploration and exploitation.

The time complexity of the TS algorithm is O(n^2) for each iteration due to the calculation of all 2-opt swaps. This makes it a practical choice for problems of moderate size.

For the initial solution of the Tabu Search, we tried some initial solutions, and it works best with a heuristic based on a Multi-Stage approach. The Multi-Stage heuristic provides a good starting point by forming routes that satisfy vehicle capacity and select the closest city for servicing.

```
class TabuSearch:
    def __init__(self, TS_problem):
        self.cvrp = TS_problem
        self.frequency_matrix = np.zeros((self.cvrp.location, self.cvrp.location))
        self.best_costs = []

    def get_neighborhood(self, solution):
        neighborhood = []
        for i in range(1, len(solution) - 1):
            for j in range(i + 1, len(solution)):
                new_solution = solution.copy()
                new_solution[i:j] = reversed(solution[i:j])  # this is a 2-opt Swap
                neighborhood.append(new_solution)
        return neighborhood

    def calculate_cost_with_penalty(self, route):
        cost = self.cvrp.calculate_route_cost(route)
        penalty = sum(self.frequency_matrix[route[i]][route[i + 1]] for i in range(len(route) - 1))
        return cost + 0.1 * penalty

    def tabu_search(self, initial_solution, min_tabu_tenure, max_tabu_tenure, max_iterations):
        best_solution = initial_solution
        best_cost = self.cvrp.calculate_total_cost(initial_solution)

        current_solution = initial_solution
        tabu_list = {}

        for iteration in range(max_iterations):
            best_neighbor = None
            best_neighbor_cost = float('inf')
            best_neighbor_route_index = None
            best_neighbor_move = None
```


Capacity: 6000
Node coordinates: [[145, 215], [151, 264], [159, 261], [130, 254], [128, 252], [163, 247], [146, 246], [161, 242], [142, 239], [163, 236], [148, 232], [128, 231], [15
Demands: [0, 1100, 700, 800, 1400, 2100, 400, 800, 100, 500, 600, 1200, 1300, 1300, 300, 900, 2100, 1000, 900, 2500, 1800, 700]
Solving the problem using Multi Stage Heuristic:
Total cost: 603.2543383557991
Vehicle 1: [0, 14, 16, 17, 20, 12, 7, 0]
Vehicle 2: [0, 13, 11, 8, 6, 10, 9, 5, 2, 0]
Vehicle 3: [0, 15, 18, 3, 0]
Vehicle 4: [0, 19, 4, 1, 0]
Clock Ticks: 0.671875s
Time Elapsed: 1.562234878540039s
Solving the problem using Clustered TSP:
Total cost: 398.63143128809025
Vehicle 1: [0, 12, 15, 14, 16, 13, 0]
Vehicle 2: [0, 10, 8, 6, 3, 4, 11, 0]
Vehicle 3: [0, 17, 19, 21, 20, 18, 0]
Vehicle 4: [0, 1, 2, 5, 7, 9, 0]
Clock Ticks: 0.703125s
Time Elapsed: 1.6132805347442627s
Solving the problem using Tabu Search:
Total cost: 579.4222844381147
Vehicle 1: [0, 14, 16, 17, 20, 7, 12, 0]
Vehicle 2: [0, 13, 11, 8, 6, 2, 5, 9, 10, 0]
Vehicle 3: [0, 3, 15, 18, 0]
Vehicle 4: [0, 19, 4, 1, 0]
Clock Ticks: 1.1875s
Time Elapsed: 3.5047688484191895s
Figure 1
Improvement of fitness in Tabu Search
Best Cost
Iteration
600
595
590
585
580
0
20
40
60
80
100

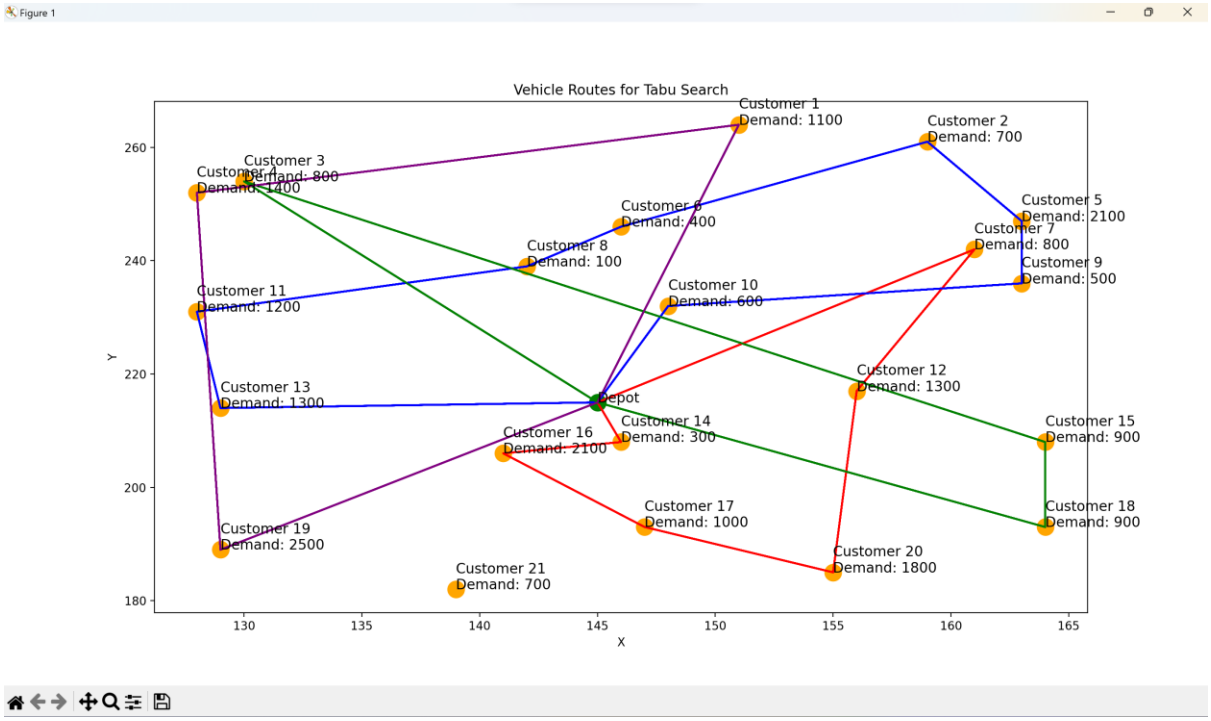


Figure 1
Vehicle Routes for Tabu Search
Customer 1
Demand: 1100
Customer 2
Demand: 700
Customer 3
Demand: 800
Customer 4
Demand: 1400
Customer 5
Demand: 2100
Customer 6
Demand: 400
Customer 7
Demand: 800
Customer 8
Demand: 100
Customer 9
Demand: 500
Customer 10
Demand: 600
Customer 11
Demand: 1200
Customer 12
Demand: 1300
Customer 13
Demand: 1300
Depot
Customer 14
Demand: 300
Customer 15
Demand: 900
Customer 16
Demand: 2100
Customer 17
Demand: 1000
Customer 18
Demand: 900
Customer 19
Demand: 2500
Customer 20
Demand: 1800
Customer 21
Demand: 700
X
Y
260
240
220
200
180
130
135
140
145
150
155
160
165

Ant Colony Optimization Algorithm:

We implemented the ACO for the CVRP follows the basic principles of ACO. The probabilistic decision rule encourages the ants to explore different solutions, while the pheromone update rule encourages them to exploit promising solutions. The evaporation of pheromones prevents the algorithm from prematurely converging to a suboptimal solution.

Heuristics we used:

1. **Pheromone Trails (tau):** These represent the desirability of moving from one city to another. The more pheromones on a path, the more likely it is to be selected.

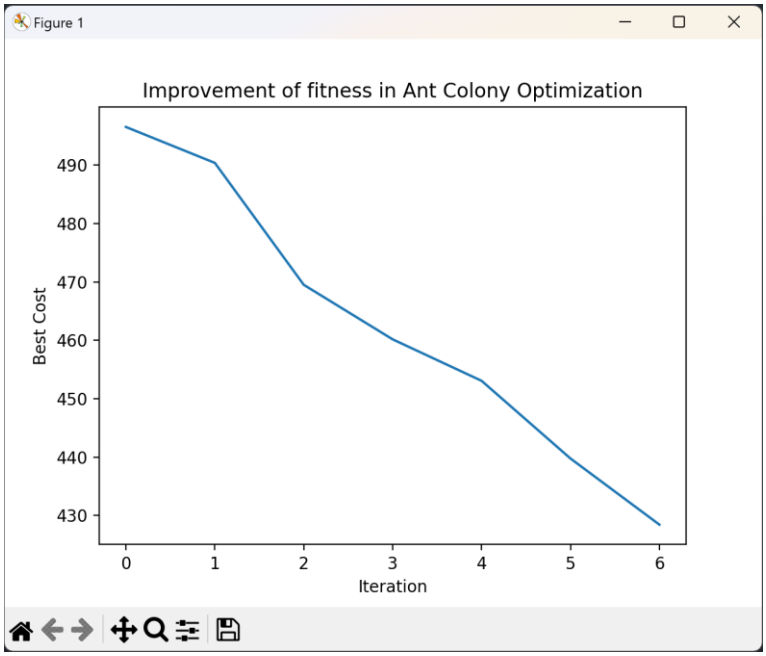
```
def update_pheromone(self, routes, costs):
    self.tau = (1 - self.rho) * self.tau
    for routes_list, cost in zip(routes, costs):
        for route in routes_list:
            for i in range(len(route) - 1):
                self.tau[route[i]][route[i + 1]] += self.q / (cost + 1e-10)
```

2. **Heuristic Information (eta):** provides a measure of the attractiveness of moving from one city to another. In this code, visibility is defined as the reciprocal of the distance between cities. This encourages ants to prefer shorter routes, reducing the overall distance travelled.

```
class AntColonyOptimization:
    def __init__(self, cvrp, n_ants, alpha, beta, rho, q, max_iter):
        self.cvrp = cvrp
        self.n_ants = n_ants
        self.alpha = alpha
        self.beta = beta
        self.rho = rho
        self.q = q
        self.max_iter = max_iter
        self.best_costs = []

        # Initialize pheromone levels
        self.tau = np.ones(self.cvrp.distance_matrix.shape)  # pheromone levels

        # Prevent division by zero
        distance_matrix_with_zeros = np.where(self.cvrp.distance_matrix == 0, 1e-10, self.cvrp.distance_matrix)
        self.eta = 1 / distance_matrix_with_zeros  # heuristic information

    def solve(self):
        best_cost = float('inf')
        best_solution = []

        for i in range(self.max_iter):
            routes, costs = self.construct_solutions()
            if min(costs) < best_cost:
                best_cost = min(costs)
                self.best_costs.append(best_cost)
                best_solution = routes[np.argmin(costs)]
            self.update_pheromone(routes, costs)
            # self.best_costs.append(best_cost)

        return best_solution, best_cost
```

For input 1:

```
--------------------------------------------------
Solving the problem using Ant Colony Optimization:
Total cost: 405.0724353984207
Vehicle 1: [0, 16, 14, 20, 18, 15, 0]
Vehicle 2: [0, 12, 10, 8, 6, 5, 7, 9, 0]
Vehicle 3: [0, 17, 21, 19, 13, 0]
Vehicle 4: [0, 11, 4, 3, 1, 2, 0]
Clock Ticks: 4.5625s
Time Elapsed: 184.38089203834534s
--------------------------------------------------
```



Simulated Annealing Algorithm:

We used an initial random solution and iteratively tries to improve it by making changes to it and observing the results. The initial temperature **T** controls the willingness of the algorithm to accept worse solutions in the initial stages. As the iterations increase, the temperature is decreased according to a factor **alpha**, reducing the likelihood of accepting worse solutions and allowing the algorithm to focus on exploiting the promising regions of the solution space.

The heuristics used in this implementation:

1. **Neighbourhood Moves:** The algorithm uses three neighbourhood moves (swap_in_routes, swap_between_routes, reverse_subroute) to generate new solutions.

```
def swap_in_route(self, solution):
    route_index = random.randint(0, len(solution) - 1)
    route = solution[route_index]
    # Ensure there are at least 2 cities in the route to swap
    if len(route) > 3:
        # Randomly choose two cities within the same route and swap them
        i, j = random.sample(range(1, len(route) - 1), 2)
        route[i], route[j] = route[j], route[i]
    return solution

def swap_between_routes(self, solution):
    # Ensure there are at least 2 routes
    if len(solution) > 2:
        # Randomly choose two routes and two cities, then swap the cities
        i, j = random.sample(range(len(solution)), 2)
        route1, route2 = solution[i], solution[j]
        if len(route1) > 2 and len(route2) > 2:  # each route must have at least 1 city
            city1, city2 = random.choice(route1[1:-1]), random.choice(route2[1:-1])
            index1, index2 = route1.index(city1), route2.index(city2)
            route1[index1], route2[index2] = city2, city1
    return solution

def reverse_subroute(self, solution):
    # Pick a random route
    route_index = random.randint(0, len(solution) - 1)
    route = solution[route_index]
    if len(route) > 4:
        # Randomly choose a subroute and reverse it
        i, j = sorted(random.sample(range(1, len(route) - 1), 2))  # exclude depot
        route[i:j] = reversed(route[i:j])
    return solution
```

2. **Multiple Restarts:** The algorithm has a mechanism to restart the search from a new random solution. This can help to ensure a more comprehensive search of the solution space.

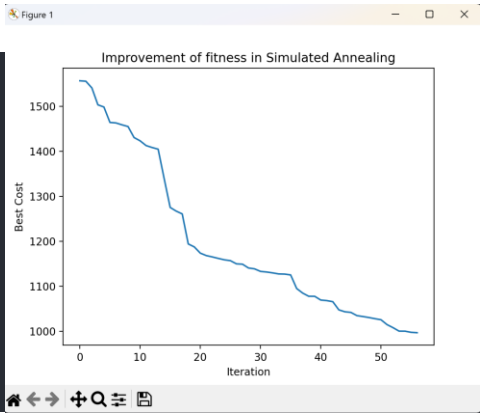
The advantages of this SA:

1. **Effective Exploration and Exploitation:** The use of temperature in SA helps balance the exploration and exploitation during the search. Initially, it allows the algorithm to explore a large solution space (exploration). As the temperature decreases, the algorithm becomes more selective, focusing more on promising areas (exploitation).
2. **Avoidance of Local Optima.**
3. **Flexibility of Moves:** The multiple types of neighbourhoods move allow the algorithm to make diverse changes to the solution, increasing the chances of finding a better solution.

For these advantages we see that the SA gave us a good results but still we have a little bit of randomness so not perfect always.

For input 2:

```
--------------------------------------------------
Solving the problem using Simulated Annealing:
Total cost: 996.9609601769835
Vehicle 1: [0, 7, 10, 25, 24, 23, 22, 20, 21, 18, 19, 0]
Vehicle 2: [0, 4, 5, 11, 14, 30, 0]
Vehicle 3: [0, 29, 16, 28, 15, 13, 6, 12, 2, 0]
Vehicle 4: [0, 3, 8, 9, 32, 1, 31, 26, 27, 17, 0]
Clock Ticks: 2.671875s
Time Elapsed: 11.578656673431396s
--------------------------------------------------
```



For input 1:

```
--------------------------------------------------
Solving the problem using Simulated Annealing:
Total cost: 453.9086786972572
Vehicle 1: [0, 6, 8, 13, 16, 0]
Vehicle 2: [0, 7, 5, 2, 10, 0]
Vehicle 3: [0, 12, 9, 1, 3, 4, 11, 0]
Vehicle 4: [0, 17, 19, 21, 20, 18, 15, 0]
Vehicle 5: [0, 14, 0]
Clock Ticks: 5.109375s
Time Elapsed: 186.87732481956482s
--------------------------------------------------
```

GA with the ISLAND MODEL:

In our GA implementation we tried some methods, but we also didn't get the best results.

Heuristic Methods:

1. **Nearest Neighbor Heuristic**: This is used to generate the initial population. Each island's population is created by shuffling unvisited cities and picking the nearest city iteratively. This heuristic has the advantage of being relatively fast and providing a good starting point for the population.

2. **Clarke Wright Savings Heuristic**: This heuristic method generates the initial population based on savings by merging routes whenever possible and valid. This heuristic prioritizes merging routes that result in the most savings in total distance travelled.

Crossover Methods:

1. **Partially Mapped Crossover (PMX)**

2. **Order Crossover (OX)**: In OX, a segment of the first parent (route list) is combined with the remaining elements from the second parent while preserving their order. The benefit of OX is that it maintains a sequence of routes from each parent.

3. **Adaptive Crossover**: This method adjusts the crossover rate over time, gradually decreasing it from a high initial rate to a lower final rate. The aim is to encourage exploration in the early stages of the algorithm and exploitation in the later stages.

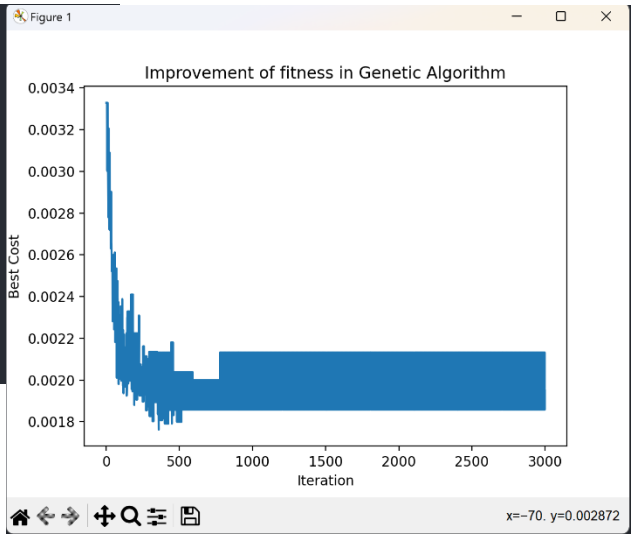
Mutation Method: Inversion Mutation**,** Swap Mutation

Selection Methods: Tournament Selection**,** Roulette Wheel Selection

Migration: The GA uses island model parallelization, with individuals occasionally migrating between islands. Migration promotes diversity in the population and prevents premature convergence.

For input 1:

```
--------------------------------------------------
Solving the problem using Genetic Algorithm:
Total cost: 584.9404599050937
Vehicle 1: [15, 7, 9, 2, 1, 10]
Vehicle 2: [19, 21, 16]
Vehicle 3: [4, 5, 11, 8, 6]
Vehicle 4: [17, 20, 3, 13]
Vehicle 5: [12, 14]
Clock Ticks: 6.203125s
Time Elapsed: 19.8451769351959235
--------------------------------------------------
```



Adaptive Large Neighbourhood Search (ALNS) Algorithm:

Heuristics: Two types of heuristics are utilized in this code: destroy and repair heuristics. Destroy heuristic (**worst_removal**) is a method that destruct part of the current solution, while repair heuristic (**greedy_insertion**) is a method that try to repair a destroyed solution to obtain a new solution.

- **worst_removal** removes a proportion of customers that contribute the most to the total cost of the solution.
- **greedy_insertion** tries to insert removed customers back into the solution in a way that minimizes the increase in total cost. If a customer cannot be inserted into any existing route without exceeding the vehicle's capacity, a new route is created for this customer.

Solving the ALNS algorithm: It starts by generating an initial solution and computing its cost. At each iteration, it randomly selects a destroy method and a repair method based on their weights, applies them to get a new solution, and computes the cost of this new solution.

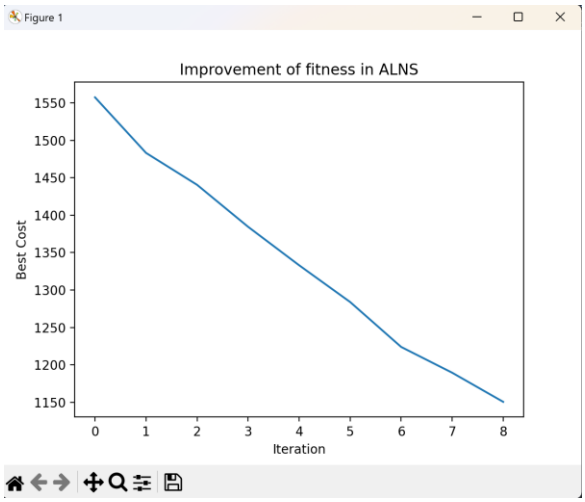
The acceptance of the new solution depends on the **acceptance_criteria** method, If the new solution is accepted, the weights of the chosen destroy and repair methods are updated accordingly. If the new solution improves the best found so far, it replaces the best solution and resets a counter of iterations without improvement (**no_improve**); otherwise, **no_improve** is incremented. The temperature for the acceptance criterion is decreased at each iteration.

Advantages of this ALNS approach: its ability to escape from local optima (thanks to the destroy and repair heuristics) and its adaptiveness to the problem's characteristics (due to the update of the weights of the heuristics based on their success).

```python
class ALNS:
    def __init__(self, cvrp, destroy_methods, repair_methods, weights, iterations, acceptance_criteria, initial_temperature, cooling_rate):
        self.cvrp = cvrp
        self.destroy_methods = destroy_methods
        self.repair_methods = repair_methods
        self.destroy_weights = dict(zip(destroy_methods, weights))
        self.repair_weights = dict(zip(repair_methods, weights))
        self.iterations = iterations
        self.acceptance_criteria = acceptance_criteria
        self.initial_temperature = initial_temperature
        self.cooling_rate = cooling_rate
        self.no_improve = 0
        self.best_costs = []

    def select_method(self, methods, weights):
        return np.random.choice(methods, p=weights / np.sum(weights))

    def update_weights(self, method, success, weights, rate=0.2):
        weights[method] += rate if success else -rate

    def solve(self):
        # Generate an initial solution
        solution = helpers.initial_solution(self.cvrp)
        best_solution = solution
        best_cost = self.cvrp.calculate_total_cost(solution)
        temperature = self.initial_temperature

        for iteration in range(self.iterations):
            # Decrease the temperature
            if self.no_improve > 10:
                temperature *= 2 * self.cooling_rate
            else:
                temperature *= self.cooling_rate
```

For input 2:

```
--------------------------------------------------
Solving the problem using Adaptive Large Neighbourhood Search:
Total cost: 1150.596999297265
Vehicle 1: [0, 4, 8, 5, 6, 9, 7, 2, 0]
Vehicle 2: [0, 1, 17, 25, 22, 10, 32, 13, 11, 0]
Vehicle 3: [0, 14, 15, 21, 20, 23, 24, 28, 30, 0]
Vehicle 4: [0, 31, 26, 29, 16, 27, 0]
Vehicle 5: [0, 19, 18, 12, 3, 0]
Clock Ticks: 0.875s
Time Elapsed: 2.1430163383483887s
```



Conclusions:

Based on most of the results we got we can say that:

1. **Efficiency:** In terms of computational time, the Clustered TSP and Tabu Search algorithms are the most efficient. The Genetic Algorithm is the least efficient in terms of time.
2. **Optimality**: The Clustered TSP algorithm provided the closest result to the optimal solution. The solution from the Adaptive Large Neighbourhood Search (ALNS) algorithm had the highest cost, making it the least optimal among the compared methods.

   We can also see that the Ant Colony Optimization algorithm performed reasonably well, considering both computation time and solution quality, finding a near-optimal solution in a moderate amount of time.

   Tabu search an SA also gave some good results, but they needed a good initial solution.

In conclusion, while different algorithms have varying efficiencies and optimality, the Clustered TSP provided the most efficient and optimal solution.