IE 440: Nonlinear Programming

Prof. İ. Kuban Altınel

Homework 2

Group 9

Mete Kalkan - 2022402030

Taha Furkan Torun - 2022402090

Yiğit Bostancı – 2022402096

Table of Contents

Single Facility Problem with Squared Euclidian Distance

Parameters Used:

We have used 4 parameters for the "squared_euclidian_distance_single_facility" function.

The first parameter is *cost_list*, which is a 50-element list where each element holds a float number that represents the information of the selected facility's cost of serving each customer. The data for this parameter is retrieved from one of the rows of the costs_9.csv file. In our case, we used the first row of the file, which represents Facility 0.

The second parameter is *demand_list*, which is a 100-element list of float numbers that hold the demand values of each customer. The data for this parameter is retrieved from the demands_9.csv file.

The third and fourth parameters are *coordinates_x1* and *coordinates_x2*, which are 100-element lists that hold float numbers representing the coordinate information of each customer. The data for this parameter is retrieved from the coordinates_9.csv file.

For this part, we solved the single facility location problem for Facility 0. Customer demands (h_j), coordinates (a_j), and transportation costs (c_0_j) were loaded from the provided data files. The cost per unit squared distance was calculated as C_0_j = h_j * c_0_j.

Summary of Requirements and Solution Plan:

Short Summary: The task is to find the optimal location for a single facility. We have selected Facility 0 for this part. The objective is to find the coordinates that minimize the objective function value. The distance is specifically defined as the squared euclidean distance.

Solution Plan: The general objective function is: $z = \sum_i \sum_j y_{ij} \cdot C_{ij} \cdot d(x_i, a_j)$

For our specific problem in, this is simplified significantly since we are only locating Facility 0 using squared Euclidean distance.

Therefore, our specific objective is to find the location that minimizes:

$$Z(x_0) = \sum_j C_{0j} \cdot \left|\left|x_0 - a_j\right|\right|_2^2$$

This form of the problem has a well-known, direct analytical solution (it does not require an iterative algorithm). The optimal location is simply the "center of gravity," or the weighted average of all customer coordinates, where the weight for each customer is.

Source Code:

```
def squared_euclidian_distance_single_facility(cost_list, demand_list, coordinates_x1, coordinates_x2):
    numerator = 0
    denominator = 0
    n = len(demand_list) # Use the actual length of the data

    for j in range(n): # Use loop variable j and iterate up to n
        numerator += demand_list[j] * cost_list[j] * coordinates_x1[j]
        denominator += demand_list[j] * cost_list[j]

    x1 = numerator / denominator
    numerator = 0 # Reset numerator for y-coordinate calculation
    # The denominator will have the same value for the x2 coordinate, no need to reset or calculate again

    for j in range(n):
        numerator += demand_list[j] * cost_list[j] * coordinates_x2[j]

    x2 = numerator / denominator
    x_star = [x1, x2] # Final coordinate
    f_star = 0

    for j in range(n):
        f_star += demand_list[j] * cost_list[j] * ((x_star[0] - coordinates_x1[j])**2 + (x_star[1] - coordinates_x2[j])**2)
        # Calculating the final cost value

    return x_star, f_star

selected_cost = costs[0] # selecting facility 1 for the single facility minimization

loc1, val1 = squared_euclidian_distance_single_facility(selected_cost, demands, x1_coordinates, x2_coordinates)
```

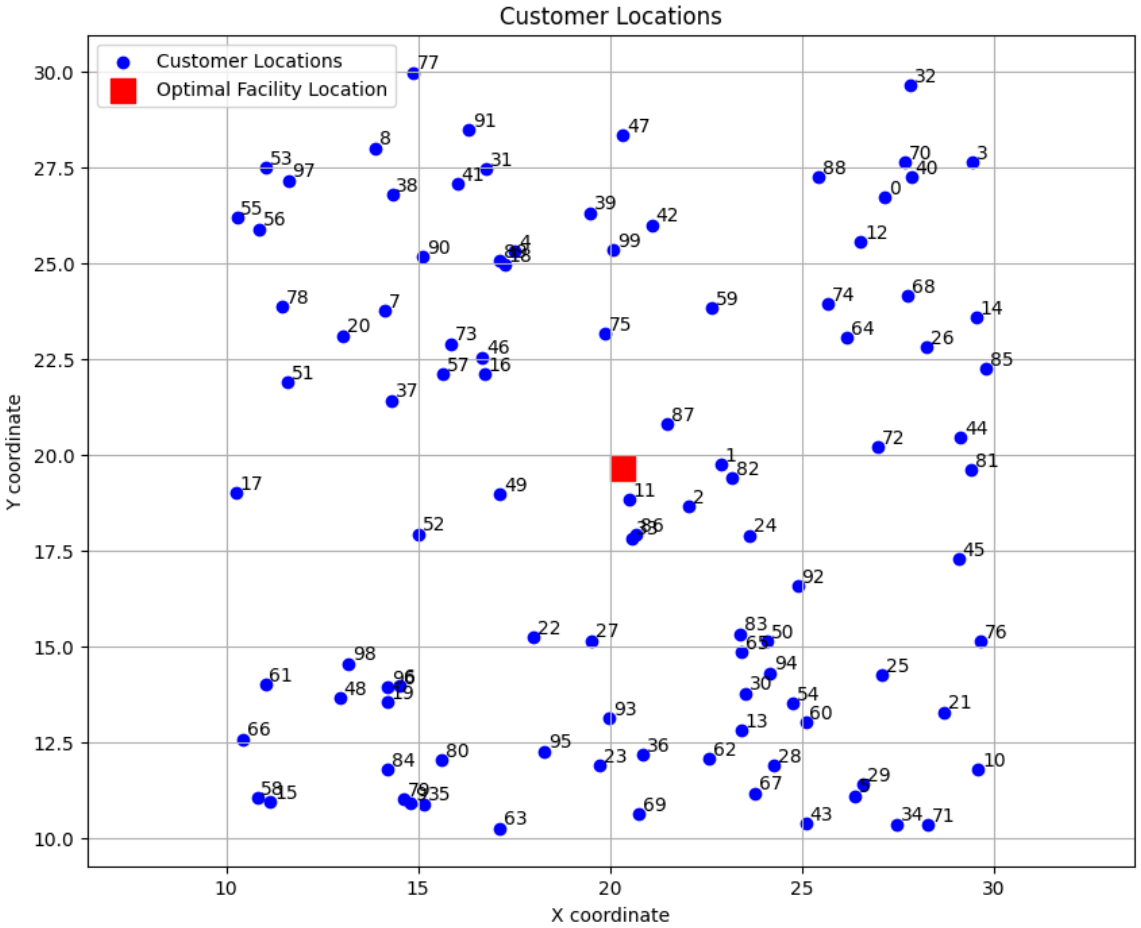
This function implements the direct analytical solution derived from the objective function.

Calculating X1-coordinate: The first for loop iterates through all customers. In each iteration j, it calculates the total weight $C_{0j} = h_j \cdot c_{0j}$ It computes the numerator $\sum_j C_{0j} \cdot a_{1j}$ and the denominator $\sum_j C_{0j}$ and gives optimal x1 coordinate.

Calculating X2-coordinate: After resetting the temporary variables the second for loop repeats the exact same logic and gives optimal x2 coordinate.

Output: The function returns the final optimal facility location [x1, x2] as [20.341599636850294, 19.65302817517095]

In final for loop, it calculates the final objective function value based on the optimal_location found earlier. It is given as: 373834.525183791



Graph of the final facility location and its customers

This is the graph which explicitly shows optimal facility location [20.341599636850294, 19.65302817517095] and all assigned customers.

Single Facility Problem with Euclidian Distance (Weiszfeld's Algorithm)

Parameters Used:

We have used 4 parameters for the "squared_euclidian_distance_single_facility" function.

The first parameter is *cost_list*, which is a 50-element list where each element holds a float number that represents the information of the selected facility's cost of serving each customer. The data for this parameter is retrieved from one of the rows of the costs_9.csv file. In our case, we used the first row of the file, which represents Facility 0.

The second parameter is *demand_list*, which is a 100-element list of float numbers that hold the demand values of each customer. The data for this parameter is retrieved from the demands_9.csv file.

The third and fourth parameters are *coordinates_x1* and *coordinates_x2*, which are 100-element lists that hold float numbers representing the coordinate information of each customer. The data for this parameter is retrieved from the coordinates_9.csv file.

A different parameter for epsilon is not defined since we decided to define epsilon within the function for uniform accuracy among solutions.

Summary of Requirements and Solution Plan:

Short Summary: The task for Part 2 is to solve the single facility problem for our chosen Facility 0. The key difference from Part 1 is that we must use the true Euclidean Distance, not the squared distance.

Selected Facility: We used the same facility as in Part 1: Facility 0

Distance Metric: Euclidean Distance $d(x_i, a_j) = \sqrt{\left((x_{(1i)} - a_{(1j)})^2 + (x_{(2i)} - a_{(2j)})^2\right)}$

Initialization: The initial facility location was set as [20.341599636850294,19.65302817517095] , which is the analytical solution from Part 1.

Stability Parameter (epsilon): To prevent "division by 0", a small value was added to the denominator of the iterative formula.

Convergence Criterion: The algorithm was set to stop when the change in facility location between two iterations was less than a pre-defined tolerance.

Solution Plan: Our objective function, based on the general formula, simplifies to:

$$Z(x_0) = \sum_j C_{0j} \cdot d(x_0, a_j) = \sum_j C_{0j} \cdot \sqrt{(x_{1,0} - a_{1j})^2 + (x_{2,0} - a_{2j})^2}$$

Because of the square root, this function cannot be solved with a simple, direct formula like in Part 1. Therefore, as required, we must use an iterative method: Weiszfeld's Algorithm.

Our plan is to:

1. Initialize Location : We start the algorithm at the specific location x_k . This location is exactly the optimal solution we found in Part 1.
2. Iterate: We will repeatedly apply the Weiszfeld update formula to find the next location $x_0^{(k+1)}$ from the current location $x_0^{(k)}$. This formula is a new weighted average, where the weights are re-calculated at each step:

$$x_0^{(k+1)} = \frac{\sum_{j=1}^{n} \frac{C_{0j} \cdot a_j}{\left\| x_0^{(k)} - a_j \right\|_2 + \epsilon}}{\sum_{j=1}^{n} \frac{C_{0j}}{\left\| x_0^{(k)} - a_j \right\|_2 + \epsilon}}$$

3. **Terminate:** We will continue this iteration until the location stops changing significantly (i.e., it converges based on our tolerance parameter). The final location will be our result.

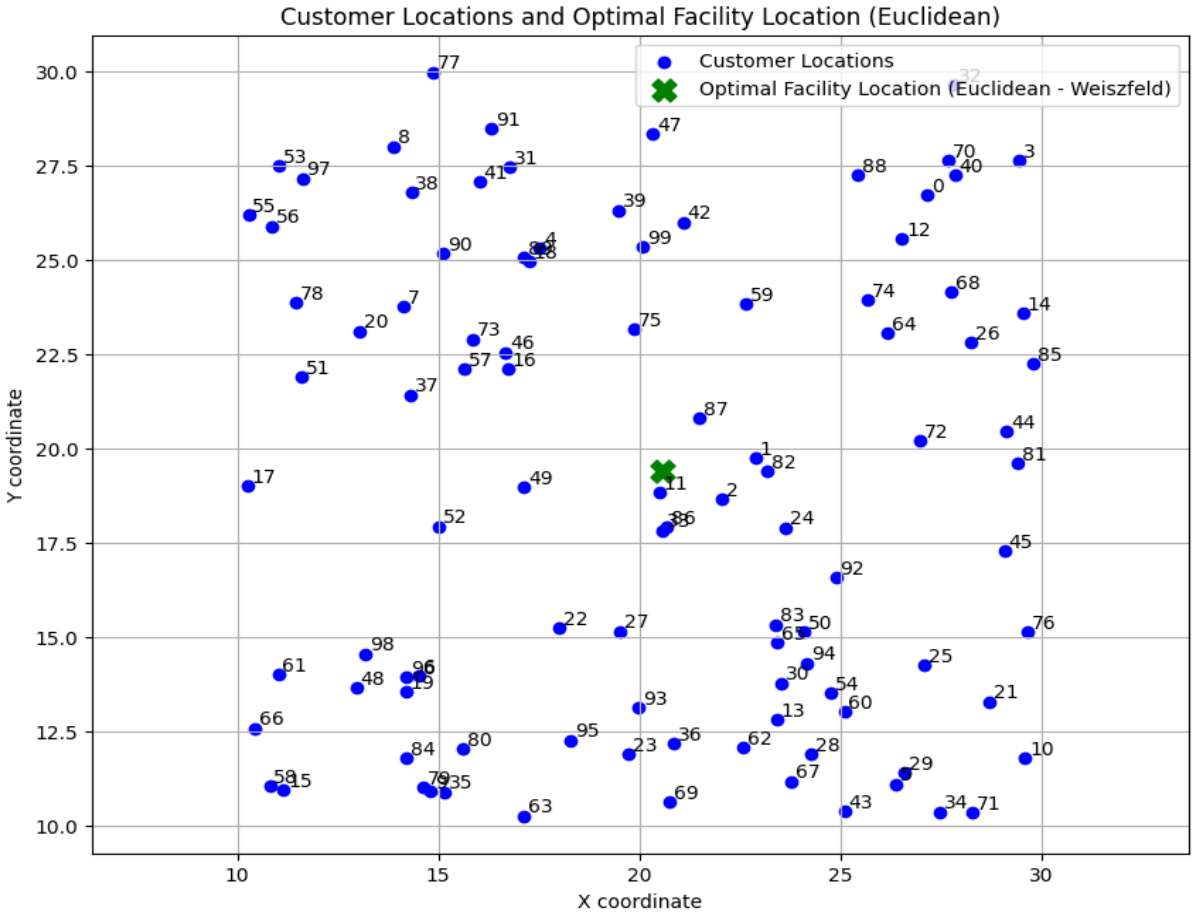
Source Code:

```
def weiszfelds_algorithm_euclidean_distance_single_facility(cost_list, demand_list, coordinates_x1, coordinates_x2):
    epsilon = 0.000001
    x_k = squared_euclidian_distance_single_facility(cost_list, demand_list, coordinates_x1, coordinates_x2)[0]
    x_k_plus_1 = []
    n = len(demand_list)

    while True:
        numerator_x1 = 0
        numerator_x2 = 0
        denominator = 0

        for i in range(n):
            distance = ((x_k[0] - coordinates_x1[i])**2 + (x_k[1] - coordinates_x2[i])**2)**0.5
            if distance < epsilon: # handle  the case where distance is very close to zero to avoid division by zero
                distance = epsilon
            weight = demand_list[i] * cost_list[i] / distance
            numerator_x1 += weight * coordinates_x1[i]
            numerator_x2 += weight * coordinates_x2[i]
            denominator += weight

        x_k_plus_1 = [numerator_x1 / denominator, numerator_x2 / denominator]

        if ((x_k_plus_1[0] - x_k[0])**2 + (x_k_plus_1[1] - x_k[1])**2)**0.5 < epsilon: # stopping condition
            break

        x_k = x_k_plus_1[:]

    x_star = x_k_plus_1
    f_star = 0

    for j in range(n):
        distance = ((x_star[0] - coordinates_x1[j])**2 + (x_star[1] - coordinates_x2[j])**2)**0.5
        f_star += demand_list[j] * cost_list[j] * distance

    return x_star, f_star

loc2, val2 = weiszfelds_algorithm_euclidean_distance_single_facility(selected_cost, demands, x1_coordinates, x2_coordinates)
```

The algorithm is initialized as required using the *x_k* coordinates from our solution in the first part. This starting location is passed to the main function. This function runs a while loop that continues as long as the distance between the previous location (*x_k*) and the new location (*x_k_plus_1*) is greater than our epsilon tolerance. Inside the loop, it iterates through all customers to calculate the new location (*x_k_plus_1*). For each customer, it computes the Euclidean distance and the corresponding customer weight $\frac{c_{0i}}{d_i}$. To prevent division by zero as per the instructions, the code checks if distance < epsilon and adds epsilon to the denominator if true. The new location *x_k_plus_1* is then calculated from the sum of weighted coordinates, and the process repeats until it converges. Final output from this function which is the optimal location is [20.589422000365275, 19.39355251932109].

Calculating the Objective Value

In final for loop, it calculates the final objective function value based on the optimal_location found earlier. It is given as: 40614.802320533585



This is the graph which explicitly shows optimal facility location [20.589422000365275, 19.39355251932109] and all assigned customers.

Multi-Facility Problem with Squared Euclidian Distance

Parameters Used:

We have used 4 parameters for the "squared_euclidian_distance_multi_facility" function.

The first parameter is *cost_list*, which is a 50-element list where each element holds a 100-element list. This list of lists is the cost matrix that holds the information for each facility's cost of serving each customer. The data for this parameter is retrieved from the costs_9.csv file.

The second parameter is *demand_list*, which is a 100-element list of float numbers that hold the demand values of each customer. The data for this parameter is retrieved from the demands_9.csv file.

The third and fourth parameters are *coordinates_x1* and *coordinates_x2*, which are 100-element lists that hold float numbers representing the coordinate information of each customer. The data for this parameter is retrieved from the coordinates_9.csv file.

Different parameters for the number of customers and facilities is not required, since we can already retrieve that information from the lengths of the *cost_list* and *demand_list* parameters.

1) Randomly Allocating Customers to Facilities Initially:

Short Summary: We have to assign 100 customers to 50 facilities randomly without leaving any facilities empty; since leaving a facility empty at initialization causes that facility to never be used again as it will never have an initial location. To solve this, we can assign the first 50 customers to the 50 facilities by their indices, then randomly assign the remaining 50 customers to random facilities, and finally shuffling the group of customers that are assigned to each facility so that we can preserve randomness.

```
def squared_euclidian_distance_multi_facility(cost_list, demand_list, coordinates_x1, coordinates_x2):
    # Define the number of facilities
    num_facilities = len(cost_list)

    # Define the number of customers
    num_customers = len(demand_list)

    # Create a list to store the facility assignment for each customer
    # Ensure each facility has at least one customer initially, then randomly assign the rest
    customer_facility_assignment = list(range(num_facilities)) # Assign the first num_facilities customers uniquely
    remaining_assignments = [random.randint(0, num_facilities - 1) for _ in range(num_customers - num_facilities)]
    customer_facility_assignment.extend(remaining_assignments)
    random.shuffle(customer_facility_assignment) # Shuffle to maintain randomness
```

The code section for (1) in the pseudo-code (explained in the next part)

Here, we do exactly what we planned to do, and achieve a randomized customer-facility assignment. Although this process is not fully random (e.g. customers 1 and 2 can never be on the same facility with this logic), we satisfy the requirement that every facility has a customer and achieve a randomness that is decent enough for initialization.

2) Applying the ALA Heuristic:

Short Summary: We have to apply the pseudo code we have seen in the lectures to apply the algorithm. The pseudo-code for the ALA heuristic is for the squared weighted distance problem is as below:

Comments on the heuristic are given in the source code explanations below.

begin

initialization: Assign customers to facilities randomly (1)

repeat

determine the customer subsets S_i (i = 1,…, m) for each facility (2)

solve m single facility location problems, one for each S_i, using *squared_euclidian_distance_single_facility* to locate facilities (3)

calculate new objective value (4)

assign customers to their nearest facilities in terms of weighted distance (5)

until no improvement

output the last objective value, facility locations, and the customers they serve (6)

end

Now, we can examine the source code for the execution of the algorithm.

```
facility_customer_assignment = {i: [] for i in range(num_facilities)}
for i in range(num_customers):
    facility = customer_facility_assignment[i]
    facility_customer_assignment[facility].append(i)
```

The code section for (2) in the pseudo-code

facility_customer_assignment is initialized as a variable that holds a dictionary which holds facility indices as keys, and assigns each facility to their customers that were assigned in (1).

```
objective_function_value_k = float('inf') # Initialize with a infinitely large value
objective_function_value_k_plus_1 = 1000000000000 # Initialize also with a large value, but less than infinite so that the while loop can begin
facility_locations = {}

# Iterative process
while objective_function_value_k - objective_function_value_k_plus_1 > 0: # Convergence criterion
    objective_function_value_k = objective_function_value_k_plus_1 # Update current objective value
    objective_function_value_k_plus_1 = 0 # Reset next objective value

    # Initialize dictionaries to store grouped data
    grouped_demand = {i: [] for i in range(num_facilities)}
    grouped_cost = {i: [] for i in range(num_facilities)}
    grouped_coordinates_x1 = {i: [] for i in range(num_facilities)}
    grouped_coordinates_x2 = {i: [] for i in range(num_facilities)}

    # Group the data based on customer_facility_assignment
    for i in range(num_customers):
        facility_index = customer_facility_assignment[i]
        grouped_demand[facility_index].append(demand_list[i])
        grouped_cost[facility_index].append(cost_list[facility_index][i])
        grouped_coordinates_x1[facility_index].append(coordinates_x1[i])
        grouped_coordinates_x2[facility_index].append(coordinates_x2[i])
```

Part (2) in the pseudo-code continued

To begin the loop, we initialize *objective_function_value_k* and *objective_function_value_k_plus_1* as large values so that the loop can begin. Also, we create an empty dictionary *facility_locations* outside of the loop.

Inside the loop, we first update the current objective value and reset the next objective value. Then, we create all required customer subsets as dictionaries and assign all customers' properties to their corresponding facilities.

```
# Calculate facility locations using squared_euclidian_distance_single_facility for each group
for facility_index in range(num_facilities):
    if grouped_demand[facility_index]:
        facility_location = squared_euclidian_distance_single_facility(
        grouped_cost[facility_index],
        grouped_demand[facility_index],
        grouped_coordinates_x1[facility_index],
        grouped_coordinates_x2[facility_index]
    )[0]
        facility_locations[facility_index] = facility_location
```

The code section for (3) in the pseudo-code

We execute a for loop that calculates the locations of each facility with the *squared_euclidian_distance_single_facility* function we had defined previously. However, if a facility has no customers assigned to it in this iteration, this calculation is not made for that facility, and the facility stays in the exact same coordinates as in the previous iteration (this is ensured by the if statement).

We keep the facility locations same for facilities that have no customers, because in the next iterations, some customers may be closer to these facilties in terms of weighted distance. If we were to instead set their locations as None, this would result in these facilities disappearing in further iterations, which is not sensible for this optimization problem, as there is no cost to open a facility.

```
# Calculate new objective value
for facility in facility_customer_assignment:
    customers = facility_customer_assignment[facility]
    if customers:
        for customer in customers:
            customer_location = [coordinates_x1[customer], coordinates_x2[customer]]
            customer_demand = demand_list[customer]
            customer_cost = cost_list[facility][customer]
            if facility_locations[facility] is not None:
                distance_squared = (facility_locations[facility][0] - customer_location[0])**2 + (facility_locations[facility][1] - customer_location[1])**2
                objective_function_value_k_plus_1 += customer_demand * customer_cost * distance_squared
```

The code section for (4) in the pseudo-code

Here, we calculate the objective function value for the current iteration in nested for loops. The first loop iterates over all facilities and identifies the customers of each facility. The second loop iterates over all the customers within each facility to calculate the weighted distance and multiplies it with customer demand and adds it to the objective function value iteratively.

```
    # Reassign customers to nearest facilities
    new_customer_facility_assignment = []
    new_facility_customer_assignment = {i: [] for i in range(num_facilities)}

    for i in range(num_customers):
        customer_location = [coordinates_x1[i], coordinates_x2[i]]
        min_weighted_distance_squared = float('inf')
        nearest_facility_index = -1

        for facility_index, facility_location in facility_locations.items():
            if facility_location is not None: # Only consider facilities with a calculated location
                # Calculate squared Euclidean distance
                distance_squared = (customer_location[0] - facility_location[0])**2 + (customer_location[1] - facility_location[1])**2

                # Calculate weighted distance squared
                weighted_distance_squared = demand_list[i] * cost_list[facility_index][i] * distance_squared

                if weighted_distance_squared < min_weighted_distance_squared:
                    min_weighted_distance_squared = weighted_distance_squared
                    nearest_facility_index = facility_index

        new_customer_facility_assignment.append(nearest_facility_index)

    for i in range(num_customers):
        facility = new_customer_facility_assignment[i]
        new_facility_customer_assignment[facility].append(i)

    customer_facility_assignment = new_customer_facility_assignment
    facility_customer_assignment = new_facility_customer_assignment # Update assignments for the next iteration

return facility_locations, facility_customer_assignment, objective_function_value_k_plus_1, num_facilities
```

The code section for (5) and (6) in the pseudo-code

Finally, we reassign customers to their nearest facilities in terms of weighted distance. In the nested for loop, for every customer, we find the smallest weighted distanced facilities by calculating the weighted distance for each facilty to the customer, and select the facility that yields the minimum weighted distance value for that customer. Then we add them into the the *new_customer_ facility_assignment* in order. In the last for loop, we assign facilities to the customers for setting up (2) for the next iteration step. The assignment variables are updated and the loop continues.

In the end, the facility locations, their customer assignments, objective function value are returned along with the number of facilities.

3) Results of the Heuristic Algorithm:

Short Summary: We have to execute the heuristic function 1000 times with different seeds. After determining the best result with simple Python operations, we save its seed and execute the function again with that seed. Finally, we print out the values.

From 1000 trials, results of the best outcome and the average of all outcomes of the ALA heuristic with squared euclidian distance are as shown below:

Final Facility Locations:

Facility 0: [18.02, 15.230000000000002]
Facility 1: [10.910905958289153, 13.74102675923016]
Facility 2: [20.58531338474134, 18.239956476131457]
Facility 3: [24.420592331624892, 13.431880889077128]
Facility 4: [23.514310729465375, 15.184729745795703]
Facility 5: [14.134535233082184, 13.88813768844976]
Facility 6: [14.892353804896342, 25.641233545084912]
Facility 7: [16.332896453606942, 22.32512824536204]
Facility 8: [11.057083949942628, 26.79411464612397]
Facility 9: [22.65, 23.83]
Facility 10: [26.98, 20.2]
Facility 11: [28.195849487666344, 13.593211891516448]
Facility 12: [10.86363659605919, 11.025802077513744]
Facility 13: [22.85513369228978, 19.280243755786458]
Facility 14: [13.932251334685818, 17.98526347581509]
Facility 15: [13.71210790457499, 15.710767014654182]
Facility 16: [19.786090324597396, 25.81025548064417]
Facility 17: [10.9564053021018, 20.49591711033322]
Facility 18: [27.79292327505344, 29.16646485714503]
Facility 19: [27.86907214559846, 10.345604855230437]
Facility 20: [15.350838439653776, 10.854078559597761]
Facility 21: [21.1, 25.99]
Facility 22: [16.557846204777668, 27.3378955688362]
Facility 23: [13.621574361101981, 23.204066267331747]
Facility 24: [26.420590023006284, 11.356104824950709]
Facility 25: [23.63, 17.89]
Facility 26: [11.46, 23.85]
Facility 27: [29.09, 17.27]
Facility 28: [29.59, 11.79]
Facility 29: [14.560531444570097, 29.351800975054736]
Facility 30: [19.13045911441395, 20.59108052997876]
Facility 31: [19.84113666152502, 12.200127089050335]
Facility 32: [19.52, 15.15]
Facility 33: [20.33, 28.32]
Facility 34: [18.836201870357957, 17.446621570244524]
Facility 35: [21.5119498723422, 20.714094530126975]
Facility 36: [24.30717124663193, 10.885720115352028]
Facility 37: [26.259358974691715, 23.639160308654155]
Facility 38: [20.74, 10.61]
Facility 39: [19.87, 23.16]
Facility 40: [25.44, 27.23]
Facility 41: [25.113095377960683, 19.639604808831297]
Facility 42: [26.827182356360062, 26.093999546584485]
Facility 43: [29.0641503856997, 27.522538140068484]
Facility 44: [24.889999999999997, 16.58]
Facility 45: [17.324579200187298, 25.113094475414684]
Facility 46: [29.662939586917464, 22.5557241460248]
Facility 47: [15.052057592051847, 17.940871988499488]

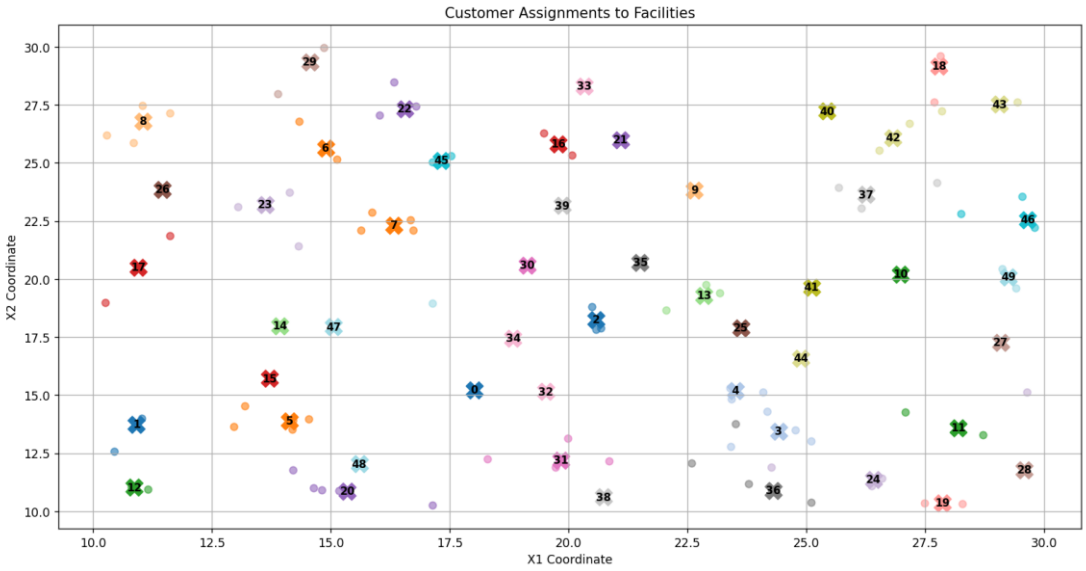
Facility 48: [15.6, 12.030000000000001]
Facility 49: [29.254890439592756, 20.083941814986762]

Final Customer Assignments:

Facility 0: Customers [22]
Facility 1: Customers [61, 66]
Facility 2: Customers [11, 33, 86]
Facility 3: Customers [13, 54, 60, 94]
Facility 4: Customers [50, 65, 83]
Facility 5: Customers [6, 19, 48, 96, 98]
Facility 6: Customers [38, 90]
Facility 7: Customers [16, 46, 57, 73]
Facility 8: Customers [53, 55, 56, 97]
Facility 9: Customers [59]
Facility 10: Customers [72]
Facility 11: Customers [21, 25]
Facility 12: Customers [15, 58]
Facility 13: Customers [1, 2, 82]
Facility 14: Customers []
Facility 15: Customers []
Facility 16: Customers [39, 99]
Facility 17: Customers [17, 51]
Facility 18: Customers [32, 70]
Facility 19: Customers [34, 71]
Facility 20: Customers [9, 35, 63, 79, 84]
Facility 21: Customers [42]
Facility 22: Customers [31, 41, 91]
Facility 23: Customers [7, 20, 37]
Facility 24: Customers [5, 28, 29, 76]
Facility 25: Customers [24]
Facility 26: Customers [78]
Facility 27: Customers [45]
Facility 28: Customers [10]
Facility 29: Customers [8, 77]
Facility 30: Customers []
Facility 31: Customers [23, 36, 93, 95]
Facility 32: Customers [27]
Facility 33: Customers [47]
Facility 34: Customers []
Facility 35: Customers [30, 87]
Facility 36: Customers [43, 62, 67]
Facility 37: Customers [64, 68, 74]
Facility 38: Customers [69]
Facility 39: Customers [75]
Facility 40: Customers [88]
Facility 41: Customers []
Facility 42: Customers [0, 12]
Facility 43: Customers [3, 40]
Facility 44: Customers [92]
Facility 45: Customers [4, 18, 89]
Facility 46: Customers [14, 26, 85]
Facility 47: Customers [49, 52]
Facility 48: Customers [80]
Facility 49: Customers [44, 81]

Final Objective Function Value: 1930.6007305669764

Average of 1000 trials: 3390.4236025861



Graph of the customer assignments to the facilities (Best Solution)

Multi-Facility Problem with Euclidian Distance

Parameters Used:

We have used 4 parameters for the "euclidian_distance_multi_facility" function.

The first parameter is *cost_list*, which is a 50-element list where each element holds a 100-element list. This list of lists is the cost matrix that holds the information for each facility's cost of serving each customer. The data for this parameter is retrieved from the costs_9.csv file.

The second parameter is *demand_list*, which is a 100-element list of float numbers that hold the demand values of each customer. The data for this parameter is retrieved from the demands_9.csv file.

The third and fourth parameters are *coordinates_x1* and *coordinates_x2*, which are 100-element lists that hold float numbers representing the coordinate information of each customer. The data for this parameter is retrieved from the coordinates_9.csv file.

Different parameters for the number of customers and facilities is not required, since we can already retrieve that information from the lengths of the *cost_list* and *demand_list* parameters.

1) Randomly Allocating Customers to Facilities Initially:

Short Summary: We have to assign 100 customers to 50 facilities randomly without leaving any facilities empty; since leaving a facility empty at initialization causes that facility to never be used again as it will never have an initial location. To solve this, we can assign the first 50 customers to the 50 facilities by their indices, then randomly assign the remaining 50 customers to random facilities, and finally shuffling the group of customers that are assigned to each facility so that we can preserve randomness.

```
def euclidian_distance_multi_facility(cost_list, demand_list, coordinates_x1, coordinates_x2):
    # Define the number of facilities
    num_facilities = len(cost_list)

    # Define the number of customers
    num_customers = len(demand_list)

    # Create a list to store the facility assignment for each customer
    # Ensure each facility has at least one customer initially, then randomly assign the rest
    customer_facility_assignment = list(range(num_facilities)) # Assign the first num_facilities customers uniquely
    remaining_assignments = [random.randint(0, num_facilities - 1) for _ in range(num_customers - num_facilities)]
    customer_facility_assignment.extend(remaining_assignments)
    random.shuffle(customer_facility_assignment) # Shuffle to maintain randomness
```

The code section for (1) in the pseudo-code (explained in the next part)

Here, we do exactly what we planned to do, and achieve a randomized customer-facility assignment. Although this process is not fully random (e.g. customers 1 and 2 can never be on the same facility with this logic), we satisfy the requirement that every facility has a customer and achieve a randomness that is decent enough for initialization.

2) Applying the ALA Heuristic:

Short Summary: We have to apply the pseudo code we have seen in the lectures to apply the algorithm. The pseudo-code for the ALA heuristic is for the weighted distance problem is as below:

Comments on the heuristic are given in the source code explanations below.

begin

initialization: Assign customers to facilities randomly (1)

repeat

determine the customer subsets S_i (i = 1,…, m) for each facility (2)

solve m single facility location problems, one for each S_i, using *weiszfelds_algorithm_euclidean_distance_single_facility* to locate facilities (3)

calculate new objective value (4)

assign customers to their nearest facilities in terms of weighted distance (5)

until no improvement

output the last objective value, facility locations, and the customers they serve (6)

end

Now, we can examine the source code for the execution of the algorithm.

```
facility_customer_assignment = {i: [] for i in range(num_facilities)}
for i in range(num_customers):
    facility = customer_facility_assignment[i]
    facility_customer_assignment[facility].append(i)
```

The code section for (2) in the pseudo-code

facility_customer_assignment is initialized as a variable that holds a dictionary which holds facility indices as keys, and assigns each facility to their customers that were assigned in (1).

```
objective_function_value_k = float('inf') # Initialize with a infinitely large value
objective_function_value_k_plus_1 = 1000000000000 # Initialize also with a large value, but less than infinite so that the while loop can begin
facility_locations = {}

# Iterative process
while objective_function_value_k - objective_function_value_k_plus_1 > 0: # Convergence criterion
    objective_function_value_k = objective_function_value_k_plus_1 # Update current objective value
    objective_function_value_k_plus_1 = 0 # Reset next objective value

    # Initialize dictionaries to store grouped data
    grouped_demand = {i: [] for i in range(num_facilities)}
    grouped_cost = {i: [] for i in range(num_facilities)}
    grouped_coordinates_x1 = {i: [] for i in range(num_facilities)}
    grouped_coordinates_x2 = {i: [] for i in range(num_facilities)}

    # Group the data based on customer_facility_assignment
    for i in range(num_customers):
        facility_index = customer_facility_assignment[i]
        grouped_demand[facility_index].append(demand_list[i])
        grouped_cost[facility_index].append(cost_list[facility_index][i])
        grouped_coordinates_x1[facility_index].append(coordinates_x1[i])
        grouped_coordinates_x2[facility_index].append(coordinates_x2[i])
```

Part (2) in the pseudo-code continued

To begin the loop, we initialize *objective_function_value_k* and *objective_function_value_k_plus_1* as large values so that the loop can begin. Also, we create an empty dictionary *facility_locations* outside of the loop.

Inside the loop, we first update the current objective value and reset the next objective value. Then, we create all required customer subsets as dictionaries and assign all customers' properties to their corresponding facilities.

```
# Calculate facility locations using squared_euclidian_distance_single_facility for each group
for facility_index in range(num_facilities):
    if grouped_demand[facility_index]:
        facility_location = weiszfelds_algorithm_euclidean_distance_single_facility(grouped_cost[facility_index],
        grouped_demand[facility_index],
        grouped_coordinates_x1[facility_index],
        grouped_coordinates_x2[facility_index]
        )[0]

        facility_locations[facility_index] = facility_location
```

The code section for (3) in the pseudo-code

We execute a for loop that calculates the locations of each facility with the *weiszfelds_algorithm_euclidean_distance_single_facility* function we had defined previously. However, if a facility has no customers assigned to it in this iteration, this calculation is not made for that facility, and the facility stays in the exact same coordinates as in the previous iteration (this is ensured by the if statement).

We keep the facility locations same for facilities that have no customers, because in the next iterations, some customers may be closer to these facilties in terms of weighted distance. If we were to instead set their locations as None, this would result in these facilities disappearing in further iterations, which is not sensible for this optimization problem, as there is no cost to open a facility.

```
# Calculate new objective value
for facility in facility_customer_assignment:
    customers = facility_customer_assignment[facility]
    if customers:
        for customer in customers:
            customer_location = [coordinates_x1[customer], coordinates_x2[customer]]
            customer_demand = demand_list[customer]
            customer_cost = cost_list[facility][customer]
            if facility_locations[facility] is not None:
                distance = ((facility_locations[facility][0] - customer_location[0])**2 + (facility_locations[facility][1] - customer_location[1])**2)**0.5
                objective_function_value_k_plus_1 += customer_demand * customer_cost * distance
```

The code section for (4) in the pseudo-code

Here, we calculate the objective function value for the current iteration in nested for loops. The first loop iterates over all facilities and identifies the customers of each facility. The second loop iterates over all the customers within each facility to calculate the weighted distance and multiplies it with customer demand and adds it to the objective function value iteratively.

```
    # Reassign customers to nearest facilities
    new_customer_facility_assignment = []
    new_facility_customer_assignment = {i: [] for i in range(num_facilities)}

    for i in range(num_customers):
        customer_location = [coordinates_x1[i], coordinates_x2[i]]
        min_weighted_distance = float('inf')
        nearest_facility_index = -1

        for facility_index, facility_location in facility_locations.items():
            if facility_location is not None: # Only consider facilities with a calculated location
                # Calculate Euclidean distance
                distance = ((customer_location[0] - facility_location[0])**2 + (customer_location[1] - facility_location[1])**2)**0.5

                # Calculate weighted distance
                weighted_distance = demand_list[i] * cost_list[facility_index][i] * distance

                if weighted_distance < min_weighted_distance:
                    min_weighted_distance = weighted_distance
                    nearest_facility_index = facility_index

        new_customer_facility_assignment.append(nearest_facility_index)

    for i in range(num_customers):
        facility = new_customer_facility_assignment[i]
        new_facility_customer_assignment[facility].append(i)

    customer_facility_assignment = new_customer_facility_assignment
    facility_customer_assignment = new_facility_customer_assignment # Update assignments for the next iteration

return facility_locations, facility_customer_assignment, objective_function_value_k_plus_1, num_facilities
```

The code section for (5) and (6) in the pseudo-code

Finally, we reassign customers to their nearest facilities in terms of weighted distance. In the nested for loop, for every customer, we find the smallest weighted distanced facilities by calculating the weighted distance for each facilty to the customer, and select the facility that yields the minimum weighted distance value for that customer. Then we add them into the the *new_customer_ facility_assignment* in order. In the last for loop, we assign facilities to the customers for setting up (2) for the next iteration step. The assignment variables are updated and the loop continues.

In the end, the facility locations, their customer assignments, objective function value are returned along with the number of facilities.

3) Results of the Heuristic Algorithm:

Short Summary: We have to execute the heuristic function 1000 times with different seeds. After determining the best result with simple Python operations, we save its seed and execute the function again with that seed. Finally, we print out the values.

From 1000 trials, results of the best outcome and the average of all outcomes of the ALA heuristic with euclidian distance are as shown below:

Final Facility Locations:

Facility 0: [23.419999641011334, 12.800001190355976]
Facility 1: [20.489999977183313, 18.82000005328004]
Facility 2: [14.2, 13.919999952349658]
Facility 3: [11.610000000000001, 21.88]
Facility 4: [15.02999999924123, 17.930000053694194]
Facility 5: [14.529999724898305, 13.97999957071599]
Facility 6: [10.250000022960597, 18.99999998278561]
Facility 7: [29.649999999999995, 15.13]
Facility 8: [10.810000226013965, 11.039999940172773]
Facility 9: [29.589999852369445, 11.79000025283853]
Facility 10: [22.649999867632822, 23.82999995905744]
Facility 11: [17.2599988050213, 24.960006865088236]
Facility 12: [14.330000000000002, 26.78]
Facility 13: [13.040000172475791, 23.100000101270194]
Facility 14: [29.549999997958885, 23.57000007496468]
Facility 15: [21.49000015769673, 20.789999983052628]
Facility 16: [24.89, 16.58]
Facility 17: [26.380000370127597, 11.100000521543434]
Facility 18: [29.089999999999996, 17.27]
Facility 19: [27.82, 29.62]
Facility 20: [23.789999984929324, 11.17000014528807]
Facility 21: [19.52, 15.15]
Facility 22: [25.439999999436296, 27.230000000077347]
Facility 23: [26.53, 25.560000000000002]
Facility 24: [15.130000050723302, 25.160000103818394]
Facility 25: [27.789472219484136, 27.271130712984014]
Facility 26: [23.430000502220754, 14.840000237771703]
Facility 27: [20.21696039679449, 17.53707710462311]
Facility 28: [20.329999813737192, 28.31999955735192]
Facility 29: [11.020000050678101, 14.009999990857045]
Facility 30: [28.27999974515031, 10.330000009677835]
Facility 31: [13.890000001810261, 27.989999993907187]
Facility 32: [25.68000203841942, 23.939998289071802]
Facility 33: [27.08, 14.26]
Facility 34: [25.1, 10.39]
Facility 35: [20.080000692210838, 25.34000043180443]
Facility 36: [20.580000207449984, 17.820000111833323]
Facility 37: [22.0600002438299, 18.650000161101897]
Facility 38: [23.63, 17.89]
Facility 39: [29.809999584814804, 22.230000103142913]
Facility 40: [29.130000081010362, 20.449999762555823]

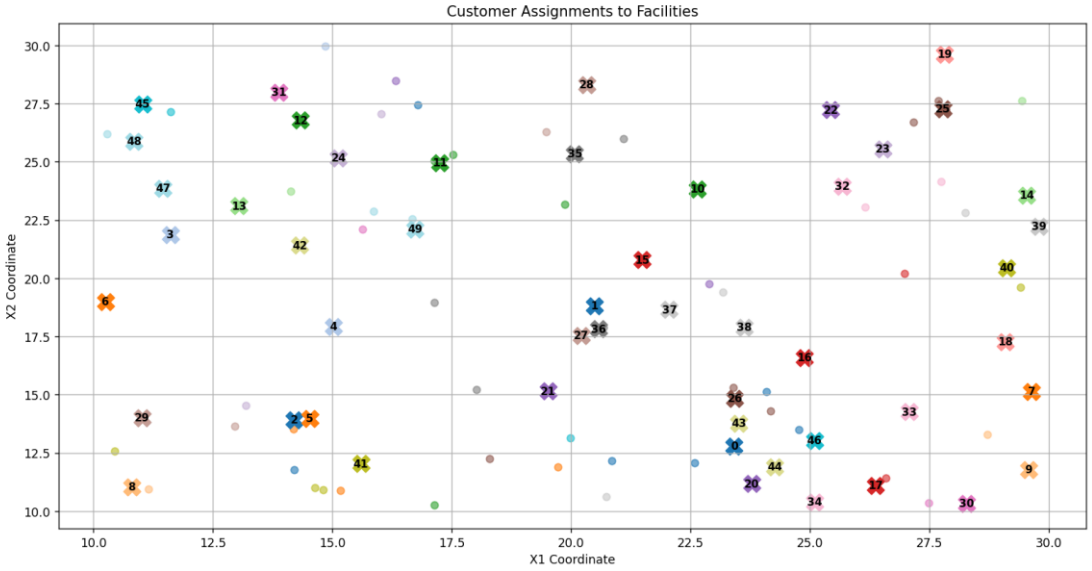
Facility 41: [15.599974196370214, 12.029968319826015]
Facility 42: [14.32, 21.41]
Facility 43: [23.52, 13.77]
Facility 44: [24.269999999999996, 11.890000000000002]
Facility 45: [11.03000280234544, 27.469998403283952]
Facility 46: [25.1, 13.03]
Facility 47: [11.46, 23.85]
Facility 48: [10.849998935032177, 25.880000579193023]
Facility 49: [16.729979268438164, 22.110044666201095]

Final Customer Assignments:

Facility 0: Customers [13, 36, 50, 54, 62]
Facility 1: Customers [11, 31]
Facility 2: Customers [84, 96]
Facility 3: Customers [51]
Facility 4: Customers [52, 77]
Facility 5: Customers [6, 19, 35]
Facility 6: Customers [17, 23]
Facility 7: Customers [76]
Facility 8: Customers [15, 58]
Facility 9: Customers [10, 21]
Facility 10: Customers [59, 63, 75]
Facility 11: Customers [4, 18, 89]
Facility 12: Customers [38]
Facility 13: Customers [7, 20]
Facility 14: Customers [3, 14]
Facility 15: Customers [72, 87]
Facility 16: Customers [92]
Facility 17: Customers [5, 29]
Facility 18: Customers [45]
Facility 19: Customers [32]
Facility 20: Customers [1, 67]
Facility 21: Customers [27]
Facility 22: Customers [88, 91]
Facility 23: Customers [12]
Facility 24: Customers [41, 90, 98]
Facility 25: Customers [0, 40, 70, 95]
Facility 26: Customers [65, 83, 94]
Facility 27: Customers []
Facility 28: Customers [39, 47]
Facility 29: Customers [48, 61]
Facility 30: Customers [34, 71]
Facility 31: Customers [8, 57]
Facility 32: Customers [64, 68, 74]
Facility 33: Customers [25]
Facility 34: Customers [43]
Facility 35: Customers [42, 49, 99]
Facility 36: Customers [22, 33, 86]
Facility 37: Customers [2, 82]
Facility 38: Customers [24]

Facility 39: Customers [26, 69, 85]
Facility 40: Customers [44, 81]
Facility 41: Customers [9, 66, 79, 80]
Facility 42: Customers [37]
Facility 43: Customers [30]
Facility 44: Customers [28]
Facility 45: Customers [53, 93, 97]
Facility 46: Customers [60]
Facility 47: Customers [78]
Facility 48: Customers [55, 56]
Facility 49: Customers [16, 46, 73]

Final Objective Function Value: 1310.9886714492902
Average of 1000 trials: 1811.1987235695171



Graph of the customer assignments to the facilities (Best Solution)