Introduction To AI

Bashar Beshoti (207370248)

April 5, 2024

Course Information

- **Course Title:** Introduction To AI
- **Course Code:** 324.3610
- **Assignment :** Vehicle Routing Problem
- **Due Date :** 05.04.2024

Vehicle Routing Problem

Introduction

The Vehicle Routing Problem (VRP) is a combinatorial optimization problem that involves the efficient distribution of goods or services from a central depot to a set of geographically dispersed locations using a fleet of vehicles. The goal is to minimize the overall transportation cost, which can include factors such as distance traveled, time, etc. The problem is defined as follows, Given:

1. Depot - A central location where the fleet of vehicles starts and returns after completing their routes.
2. Locations - Locations that require goods or services to be delivered.
3. Transportation cost - expenses incurred in moving from one location to another, encompassing factors like fuel, vehicle maintenance, labor, and associated logistical expenses.
4. Constraints - Various constraints need to be satisfied, such as vehicle capacity constraints (a vehicle cannot exceed its maximum capacity) or window time that the location must be visited.
5. Objective: Minimize the total cost of the delivery routes

In this assignment we'll focus on VRP with 2 factors of transformation costs - distance and time traveled and no constraints.

Class VRP

We'll start with defining class VRP - an instance that parses the data file and holds all neccesary data of the VRP.

1. fill in the function *compute_distance_matrix*

```
# ~~~~TODO 1: fill in this method to initialize the distance matrix
  ~~~~
  def compute_distance_matrix(self):
        dis_mtx = np.zeros((self.dim, self.dim))
        for i in range(self.dim):
            for j in range(self.dim):
                dis_mtx[i, j] = np.sqrt((self.locations[i][0] - self.
                   locations[j][0]) ** 2 +
                                        (self.locations[i][1] - self.
                                           locations[j][1]) ** 2)
        return dis_mtx
```

2. fill in the function *plot locations*

```
#~~~~ TODO 2: plot locations
   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  def plot_locations(self):
    plt.figure(figsize=(8, 6))
    plt.scatter(self.locations[:, 0], self.locations[:, 1], color='
       blue', label='Locations')
    plt.xlabel('X Coordinate')
    plt.ylabel('Y Coordinate')
    plt.title('Locations')
    plt.legend()
    plt.grid(True)
    plt.show()
    pass
```

3. fill in the function *plot_routes*

```
# ~~TODO 3: Plot routes, each route hold ordered locations to visit
   ~~~~
#   Note: Make sure to add depot to the start and end of each route
def plot_routes(self, routes):
    plt.figure(figsize=(8, 6))
    plt.scatter(self.locations[:, 0], self.locations[:, 1], color='
       blue', label='Locations')

    for route in routes:
        # Add depot to the start and end of each route
        route_with_depot = [0] + route + [0]
        route_coords = [self.locations[i] for i in route_with_depot]
        route_coords = np.array(route_coords)
        plt.plot(route_coords[:, 0], route_coords[:, 1], marker='o')

    plt.xlabel('X Coordinate')
    plt.ylabel('Y Coordinate')
    plt.title('Routes')
    plt.legend()
    plt.grid(True)
    plt.show()
  pass
```

4. fill in the function *compute_route_distance*

```
#~~~ TODO 4: given a route calculate the distance it took to travel
   ~~~~
  def compute_route_distance(self, route):
    distance = 0
    for i in range(len(route) - 1):
        distance += self.dis_mtx[route[i], route[i+1]]
```

```
    # Add distance from the last location back to the depot
    distance += self.dis_mtx[route[-1], 0]
    return distance
```

5. fill in the function *compute_route_time*

```
  # TODO 5: given a route calculate the distance it took to travel
     ~~~~~~
  def compute_route_time(self, route):
    time = 0
    for i in range(len(route) - 1):
        time += self.time_mtx[route[i], route[i+1]]
    # Add time from the last location back to the depot
    time += self.time_mtx[route[-1], 0]
    return time
```

Particle swarm optimization

Particle swarm optimization (PSO) is one of the bio-inspired algorithms and it is a simple one to search for an optimal solution in the solution space. Each particle adjusts its position and velocity based on its own experience and shared knowledge within the swarm. The algorithm iteratively refines solutions, striking a balance between global exploration and local exploitation.

pseudo-code for Particle Swarm Optimization (PSO):

```
1. Initialization:
    1. Initialize a swarm of particles with random positions and velocities.
    2. Initialize personal best positions for each particle as their current positions.
    3. Initialize the global best position as the best position among all particles.
2. Main Loop: Repeat until convergence or a maximum number of iterations:
    1. For each particle in the swarm:
        (a) Evaluate the fitness of the current position.
        (b) If the fitness is better than the personal best -> Update the personal best position.
        (c) If the fitness is better than the global best ->Update the global best position
    2. Update the velocity and position using the formula:
        $v_{t+1} = w_t v_t + c_1 r_1 (p_{best,i} x_i) + c_1 r_1 (g_{best} x_i)$
    3. update positions : $x_{i+1} = x_i + v_{i+1}$
3. Return global bet position
```

```
def pso(s, d, lb, ub, c1, c2, maxiter ,obj_func, wupdate_func):
    #:param s: number of particles
    #:param d: dimension of a particle
    #:param lb: lower bound in the search space
    #:param ub: upper bound in the search space
    #:param c1: constant for velocity update
    #:param c2: constant for velocity update
    #:param maxiter: maximum number of iteration defined to run
    #:param obj_func: function to evaluate particle position
    #:param wupdate_func: function to update the velocity weight

    # initialize swarm
    p = np.random.rand(s, d)    # particle positions
    v = np.zeros_like(p)        # particle velocities
    bp = p                      # best particle positions
    f_p = np.zeros(s)           # current particle function values
    f_bp = np.ones(s) * np.inf # best particle function values
    gp = []                     # best swarm position
    f_gp = np.inf               # best swarm position starting value

    # Initialize the particle's position
    p = lb + p * (ub - lb)

    # Initialize the multiprocessing module if necessary
    processes = 5
    mp_pool = multiprocessing.Pool(processes)

    # Calculate objective function
    f_p = np.array(mp_pool.map(obj_func, p))
    f_bp = f_p.copy()

    # Update swarm's best position
    i_min = np.argmin(f_p)
    if f_p[i_min] < f_gp:
        f_g = f_p[i_min]
        gp = p[i_min, :].copy()

    # Initialize the particle's velocity
    v = -1 + np.random.rand(s, d) * 2

    # Iterate until termination criterion met
    it = 1
    print("Running...")
    while it <= maxiter and np.std(f_p)>1:
      r1 = np.random.uniform(size=(s, d))
      r2 = np.random.uniform(size=(s, d))
```

```
        # Update the particles velocities
        w = wupdate_func(it)
        v = w * v + c1 * r1 * (bp - p) + c2 * r2 * (gp - p)

        # Update the particles' positions
        p = p + v

        # Correct for bound violations
        maskl = p < lb
        masku = p > ub
        p = p * (~np.logical_or(maskl, masku)) + lb * maskl + ub *
            masku

        # Update objectives
        f_p = np.array(mp_pool.map(obj_func, p))

        # Store particle's best position
        i_update = (f_p < f_bp)
        bp[i_update, :] = p[i_update, :].copy()
        f_bp[i_update] = f_p[i_update]

        # Compare swarm's best position with global best position
        i_min = np.argmin(f_bp)
        if f_bp[i_min] < f_gp:
            gp = bp[i_min, :].copy()
            f_gp = f_p[i_min]

        it += 1
    print("Finshed run")
    return gp, f_gp
```

To run the pso provided we need to define two functions:

1. wupdate_func - updating w for velocity computation
2. obj_func - How we evaluate a particle

Define how we update w in each iteration?

In Particle Swarm Optimization (PSO), w represents the inertia weight. The inertia weight is a parameter used to control the trade-off between exploration and exploitation during the optimization process.

The inertia weight influences the particle's velocity update formula in PSO. The velocity update equation for a particle i in a given iteration is typically defined as follows:

$$v_{t+1} = w_t v_t + c_1 r_1 (p_{best,i} - x_i) + c_1 r_1 (g_{best} - x_i)$$

1. Fill in the function w_update to compute w_i

```
def w_update(w_min ,w_max, max_iter, i):
  return w_max - ((w_max - w_min) / max_iter) * i
```

Parameters:

1. This function takes four arguments:
2. w_min: Lower bound for the inertia weight.
3. w_max: Upper bound for the inertia weight.
4. max_iter: Maximum number of iterations planned for the PSO run.
5. i: Current iteration number.

Explanation : Computes the inertia weight w at the current iteration. The function implements a linear decrease of w from w_{max} to w_{min} as the number of iterations increases. This approach balances exploration and exploitation during the PSO search.

2. How does the inertia weight impact the trade-off between exploration and exploitation?

Impact of Inertia Weight on Exploration-Exploitation:

– **Exploration**: A higher inertia weight allows particles to maintain a larger velocity, encouraging them to explore different regions of the search space and potentially discover new promising areas.

– **Exploitation**: A lower inertia weight reduces the influence of the previous velocity, causing particles to converge towards the best positions found so far. This helps refine solutions and exploit promising areas more thoroughly.

3. Suggest a second version to update the inertia weight, how is it different from the first one?

```
def wupdate_clerc_kennedy(iteration, w_max, w_min):
# Faster exponential decay factor
  decay_factor = 2  # Higher value leads to faster decrease

  # Generate random number between 0 and 2
  a = np.random.rand()

  # Base inertia weight update
  w = w_max - a * abs(w_max - w_min)

  # Apply faster exponential decay
  w = w * np.exp(-decay_factor)

  w = np.clip(w,w_min,w_max)

  return w
```

Differences:

- The linear approach offers a more intuitive understanding of the weight change based on the iteration number.
- The Clerc and Kennedy (2002) approach might introduce a slightly faster decrease (Exponential decrease) in inertia weight in the initial iterations.

Defining an objective function

In this section you'll need to implement an objective function to evaluate the particle position with regard to the vrp instance. Each particle is an array with d elements:

- d - number of locations (depot not included).
- Each element holds a value in range [lb, ub] where lower bound will be set to 0 and upper bound will be set to max number of routes allowed.

1. Discretization

Particle Swarm Optimization (PSO) is generally well-suited for continuous optimization problems due to its mathematical formulation involving continuous variables and velocity updates. However, it can also be adapted for discrete optimization tasks through specific modifications and strategies.

Discretization techniques can be applied to continuous PSO solutions to convert them into valid discrete solutions. This involves rounding or mapping continuous values to the nearest valid discrete values in the solution space.

Fill in method $disctetization(x, lb, ub)$ to return a discrete x Note: Use make sure that values are in range $[lb, ub]$.

```
def discretization(x, lb, ub):
  return np.clip(np.round(x), lb, ub)
```

2. Split into assigned route

After processing into discrete domain, we have a representation of a solution for the VRP, Each value in the array corresponds to a route number for that location. For example, given an array [0,1,2,1,2] corresponds to a solution where we assign locations to routes in the following way:

location 1 -> route 0, location 2 -> route 1, location 3 -> route 2, location 4 -> route 1.

Result: route 0 will visit location 1, route 1 will visit locations 2,4 and route 2 will visit location 3.

Fill in the function *split_into_assigned_route* that given a an array splits the array into route-assignments as described above and return the route-assignments.

```
def split_into_assigned_routes(a, ub):
    assigned_routes = {}

    for i, route in enumerate(a):
        # route_index = np.round(route)
        route_index = int(route)
        if route_index not in assigned_routes:
            assigned_routes[route_index] = [i]
        else:
            assigned_routes[route_index].append(i)

    for i in range(ub + 1):
        assigned_routes.setdefault(i, [])

    return assigned_routes
```

3. Best route for assignment

After splitting the array into route-assignments,for each assignment we want to find a good order to visit each of the assigned locations starting and ending in the depot- in other words, TSP. A good order will have to factor both distance and time. Choosing the next location based only on distance may lead to longer time (perhaps there's a traffic jam).

On the other hand - choosing only based on time may lead to longer distance (perhaps another road will take a little longer but the distance is significantly smaller).

Using methods you've learned in lecture, implement a function to find this route for an assignment.

```
def find_order_for_assignment(locations, vrp):
    if not locations:
        return [0]

    def distance(location1, location2):
        return vrp.dis_mtx[location1][location2]

    current_route = locations[:]
    n = len(current_route)

    def two_opt(route):
        best_distance = calculate_route_distance(route)
        improved = True
        while improved:
            improved = False
            for i in range(1, n - 2):
                for j in range(i + 1, n):
                    if j - i == 1:
                        continue
                    new_route = route[:]
                    new_route[i:j + 1] = reversed(new_route[i:j + 1])
                         # Apply 2-opt exchange
                    new_distance = calculate_route_distance(new_route
                        )
                    if new_distance < best_distance:
                        route[:] = new_route[:]
                        best_distance = new_distance
                        improved = True
                        break
                if improved:
                    break
        return route

    def calculate_route_distance(route):
        total_distance = 0
        for i in range(n - 1):
            total_distance += distance(route[i], route[i + 1])
        return total_distance

    current_route = two_opt(current_route)

    current_route.insert(0, 0)
    current_route.append(0)

    return current_route
```

4. Integrate all the methods above into an objective function

Using all the functions you've implemented, implement an objective function The function need to return a value indicating how good/bad the particle is.

```python
def objective_function(vrp, lb, ub, particle):
    discrete_solution = discretization(particle, lb, ub)

    route_assignments = split_into_assigned_routes(discrete_solution,
        ub)

    total_cost = 0
    weight_distance = 0.5
    weight_time = 1 - weight_distance

    for route, locations in route_assignments.items():
        ordered_route = find_order_for_assignment(locations, vrp)

        route_distance = vrp.compute_route_distance(ordered_route)

        route_time = vrp.compute_route_time(ordered_route)

        route_cost = weight_distance * route_distance + weight_time *
            route_time

        total_cost += route_cost

    return total_cost
```

5. Answer Questions

1.A What is the role of particles, positions, velocities, and fitness evaluation in PSO when solving the VRP.

1. Particles: Each particle in a PSO swarm represents a potential solution to the VRP problem. It typically encodes the route assignments for all locations.

2. Positions: The position of a particle defines the specific route assignments it currently suggests. This position could be represented in various ways, such as a list of route indices for each location or a permutation of all locations.

3. Velocities: The velocity of a particle determines the direction and magnitude of change in its position during the optimization process. It influences how a particle explores the search space and potentially moves towards better solutions.

4. Fitness Evaluation: Each particle's fitness is evaluated using an objective function. This function measures the quality of the solution represented by the particle's position. In the context of VRP, the objective function might consider factors like total distance traveled, travel time, or a combination of both.

1.B. How are these elements utilized in finding the optimal or near-optimal solutions?

1. Initialization: A swarm of particles is created, with each particle having a random position and velocity.
2. Fitness Evaluation: The objective function is used to calculate the fitness score for each particle in the swarm.
3. Updating Velocity: The velocity of each particle is updated based on its current velocity, the fitness of its own best position (**pbest**), and the fitness of the global best position (**gbest**) found so far in the swarm. This update incorporates both exploration and exploitation.
4. Updating Position: Using the updated velocity, the position of each particle is moved in the search space. This can involve modifying the route assignments based on the velocity values.
5. Iteration: Steps 2 to 4 are repeated for a predefined number of iterations, allowing particles to explore and potentially converge towards better solutions.

The combination of these elements gives a deep look of the search process. Particles explore the search space due to their velocities, and the fitness evaluation lead them towards areas with better solutions as defined by the objective function. The influence of **pbest** and **gbest** encourages both individual exploration and convergence towards promising regions discovered by the swarm.

2. Suggest your own way to represent a particle for the VRP problem.
Describe the objective function for the new representation.

Instead of iterating through route indices, it would iterate through the permutation. It would calculate the cost of the entire route based on the order of locations in the permutation, considering factors like distance and travel time between consecutive locations.

For example:Let the permutation be [2, 1, 4, 3, 0]. Therefore :

– Location 2 is assigned to the first route.

– Location 1 is assigned to the second route (as it appears next in the permutation).

The objective function would calculate the total cost of the route by summing the distances/travel times between locations in the following order:

Depot -> Location 2 , Location 2 -> Location 1 ,Depot -> Location 4 , Location 4 -> Location 3 , Location 3 -> Depot.

3. Explain the concept of local and global best solutions in PSO. How are these solutions utilized during the optimization process of the VRP?

A. Local and Global Best Solutions in PSO:

– Local Best (pbest): The best position a specific particle has encountered in its own search. It represents the particle's personal best solution found so far.

– Global Best (gbest): This represents the best position discovered by the entire swarm during the optimization process. It's essentially the best solution found among all particles.

B. Utilizing Local and Global Best:

– During velocity update, a particle considers both its own pbest and the global gbest. This allows particles to: A. Explore based on their own experience (pbest).
B. Be influenced by promising solutions discovered by the swarm (gbest), potentially leading them towards better areas of the search space.

– This balance between exploration and exploitation helps PSO avoid getting stuck in local optima and potentially converge towards more optimal solutions.

MAIN

Now that we have all our ducks in order, Let's run the PSO algorithm on VRP instance.

```
import functools

#Initializing vrp instance
path = '/content/drive/MyDrive/HW1-Datasets/Ex1-d5'
vrp = VRP(path)

# Defining parameter for the run
SWARM_SIZE = ##
MAX_ROUTES = ##
C1 = ##
C2 = ##
W_MIN = ##
W_MAX = ##
MAX_ITER = ##

# Define functions to pass to pso
pso_obj_func = functools.partial(objective_function, vrp, 0,
   MAX_ROUTES-1)
pso_w_update = functools.partial(w_update, W_MIN ,W_MAX, MAX_ITER)


# Run PSO
sol,f_sol = pso(s = SWARM_SIZE,
            d = vrp.dim-1,
            lb = 0,
            ub = MAX_ROUTES - 1,
            c1 = C1,
            c2 = C2,
            maxiter = MAX_ITER,
            obj_func = pso_obj_func,
            wupdate_func = pso_w_update)

# Print and plot solution
d_sol = discretization(sol,0,MAX_ROUTES-1)
route_assignments = split_into_assigned_routes(d_sol,MAX_ROUTES-1)
sol = [find_order_for_assignment(r, vrp) for r in route_assignments]


vrp.plot_routes(sol)
vrp.print_routes(sol)
```

1. Define parameters for the algorithm and run the PSO algorithm on the given data sets.

Explanation:

VRP Instance: The code assumes you have set the `path` variable correctly to point to the data file for the VRP instance you want to solve (e.g., `Ex1-d5`). **Parameters:**

1. `SWARM_SIZE`: Number of particles in the swarm (set to 50 here).
2. `MAX_ROUTES`: Maximum number of allowed routes (set to 5 here).
3. `C1` and `C2`: PSO constants for velocity update.
4. `W_MIN` and `W_MAX`: Minimum and maximum values for the inertia weight.
5. `MAX_ITER`: Maximum number of iterations for the PSO algorithm.

Function Definitions:

– `pso_obj_func`: This is a partial function that binds the `objective_function`.

– `pso_w_update`: This is a partial function that binds the `w_update`.

Running PSO: The `pso` function is called with the defined parameters, objective function, and weight update function. It returns the best solution found (`sol`) and its corresponding objective function value (`f_sol`).

Algorithms and functions:

– `discretization`: Discretizes the continuous solution (`sol`) to valid route assignments.

- `split_into_assigned_routes`: Splits the discretized solution into individual routes.
- `find_order_for_assignment`: Optimizes the order of locations within each route using a 2-opt heuristic.
- `vrp.plot_routes`: Plots the optimized routes on a map.
- `vrp.print_routes`: Prints details about each route, including distance and travel time.

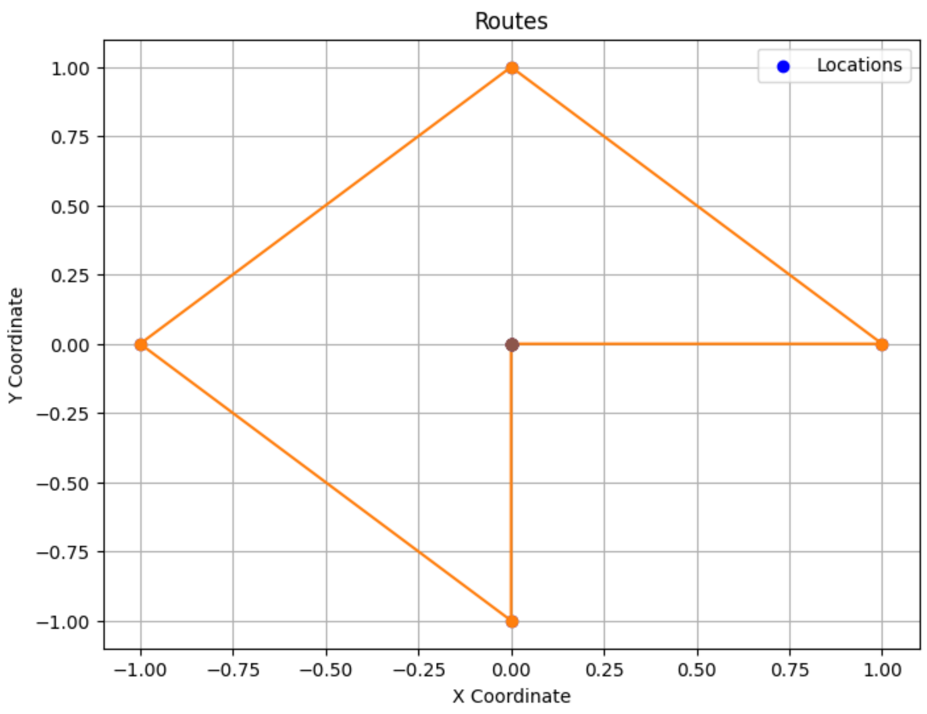
2. Report how different parameters influence the solution quality and the run-time.

Table 1: Parameters versus Solution Quality & Runetime

Parameter	Solution Quality	Runtime
SWARM_SIZE	Increasing the swarm size, the higher probability of finding a good solution by exploring a wider search space.	A larger swarm size requires more computations per iteration, increasing the overall runtime.
MAX_ITER	More iterations allow the PSO to explore a larger portion of the search space and probably find a better solution.	Runtime increases linearly with the number of iterations as more iterations needed to do.
(W_MIN, W_MAX)	A high initial inertia weight (W_MAX) promotes exploration. While A low final inertia weight (W_MIN) encourages exploitation	No effect on runtime.
C1	A higher C1 value emphasizes a particle's own experience	No effect on runtime.
C2	A higher C2 value emphasizes the swarm's knowledge	No effect on runtime.

3. Submit your best results for each of the given data sets.

0.0.1 Ex1-d5



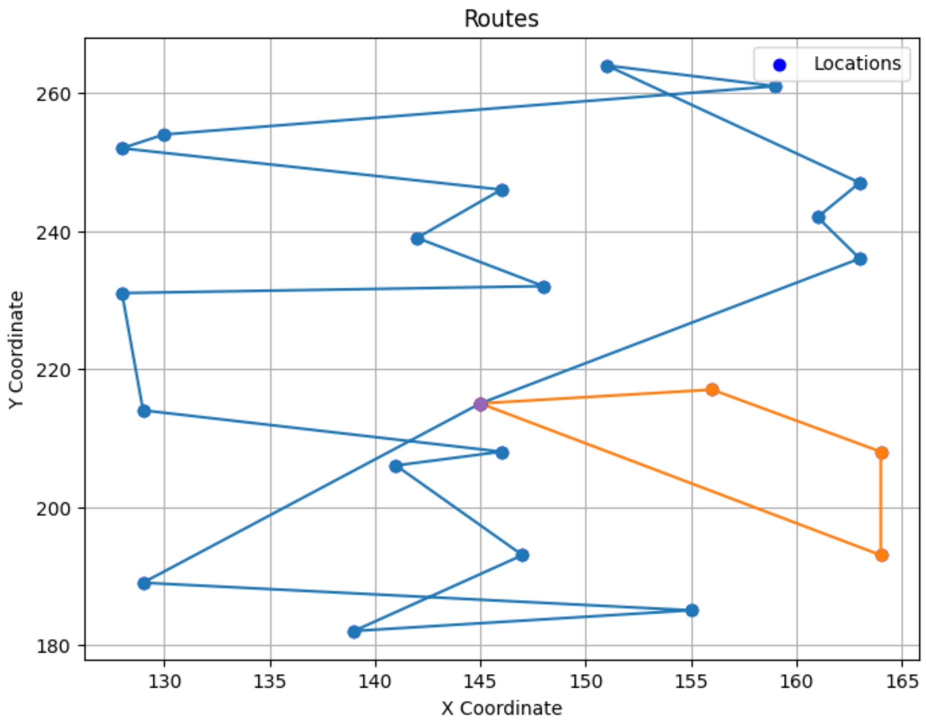
```
vehicle 1 route: depot ->0 -> 0 -> 0 -> depot
Distance for vehicle 1 r_distance=0.0, Time traveled = 0.0
vehicle 2 route: depot ->0 -> 0 -> 1 -> 2 -> 3 -> 4 -> 0 -> depot
Distance for vehicle 2 r_distance=6.242640687119285, Time traveled =
    6.229999999999997
vehicle 3 route: depot ->0 -> depot
Distance for vehicle 3 r_distance=0.0, Time traveled = 0.0
vehicle 4 route: depot ->0 -> depot
Distance for vehicle 4 r_distance=0.0, Time traveled = 0.0
vehicle 5 route: depot ->0 -> depot
Distance for vehicle 5 r_distance=0.0, Time traveled = 0.0
Total Distance is: 6.242640687119285, Total Time = 6.229999999999997
```

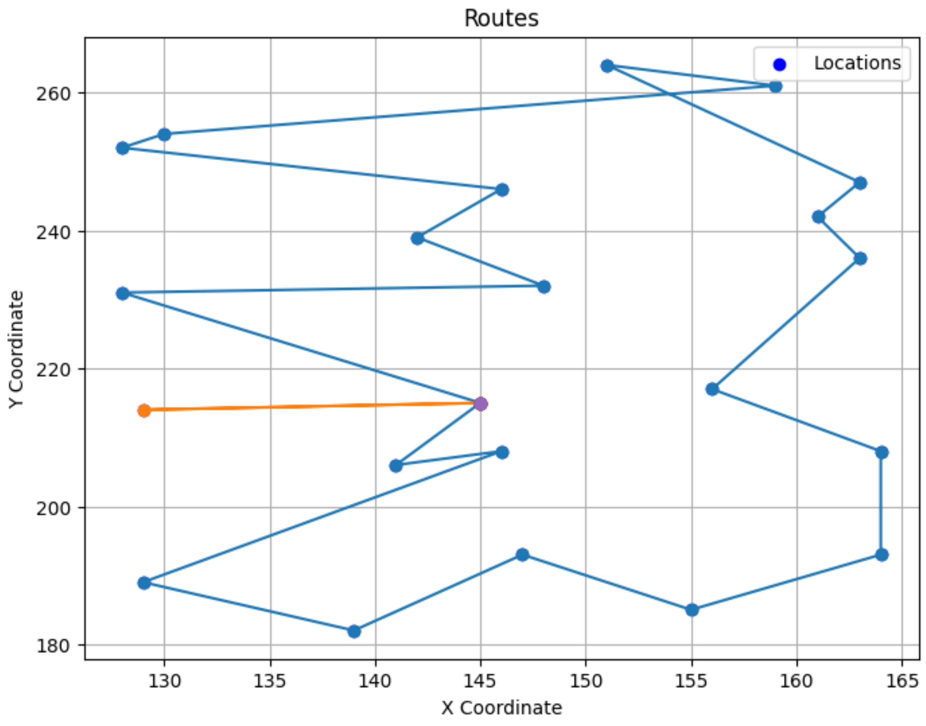
0.0.2 Ex2-d22

```
i forgot to copy routes print :)
Total Distance is: 366.4285625707137, Total Time = 113.75
```

Another run with routes print :

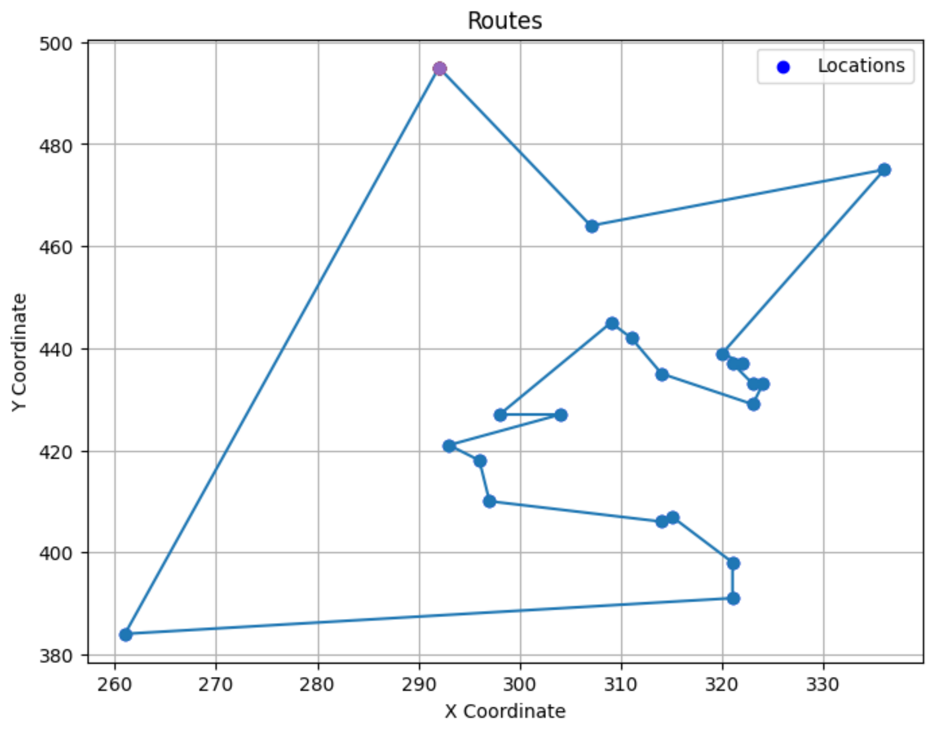
```
vehicle 1 route: depot ->0 -> 0 -> 16 -> 14 -> 19 -> 21 -> 17 -> 20
    -> 18 -> 15 -> 12 -> 9 -> 7 -> 5 -> 1 -> 2 -> 3 -> 4 -> 6 -> 8 ->
    10 -> 11 -> 0 -> depot
Distance for vehicle 1 r_distance=290.531859134782, Time traveled =
```


Routes
Locations
Y Coordinate
X Coordinate
260
240
220
200
180
130
135
140
145
150
155
160
165


Routes
Locations
Y Coordinate
X Coordinate
260
240
220
200
180
130
135
140
145
150
155
160
165

```
   111.22999999999999
vehicle 2 route: depot ->0 -> 13 -> 0 -> depot
Distance for vehicle 2 r_distance=32.0624390837628, Time traveled =
   6.8100000000000005
vehicle 3 route: depot ->0 -> depot
Distance for vehicle 3 r_distance=0.0, Time traveled = 0.0
vehicle 4 route: depot ->0 -> depot
Distance for vehicle 4 r_distance=0.0, Time traveled = 0.0
vehicle 5 route: depot ->0 -> depot
Distance for vehicle 5 r_distance=0.0, Time traveled = 0.0
Total Distance is: 322.5942982185448, Total Time = 118.03999999999999
```

0.0.3 Ex3-d33



```
vehicle 1 route: depot ->0 -> 0 -> 3 -> 4 -> 5 -> 7 -> 6 -> 8 -> 9 ->
    10 -> 11 -> 12 -> 2 -> 1 -> 13 -> 14 -> 15 -> 17 -> 19 -> 18 ->
   21 -> 20 -> 16 -> 0 -> depot
Distance for vehicle 1 r_distance=404.5910687601283, Time traveled =
   128.27
vehicle 2 route: depot ->0 -> depot
Distance for vehicle 2 r_distance=0.0, Time traveled = 0.0
vehicle 3 route: depot ->0 -> depot
Distance for vehicle 3 r_distance=0.0, Time traveled = 0.0
vehicle 4 route: depot ->0 -> depot
```

```
Distance for vehicle 4 r_distance=0.0, Time traveled = 0.0
vehicle 5 route: depot ->0 -> depot
Distance for vehicle 5 r_distance=0.0, Time traveled = 0.0
Total Distance is: 404.5910687601283, Total Time = 128.27
```