Evolutionary Algorithms

Muhammad Meesum Ali Qazalbash - mq06861

Syed Ibrahim Ali Haider - sh06565

March 19, 2023

CS451 - Computational Intelligence

Assignment 2

Contents

1 Abstract **3**

2 Problems **4**

2.1 Capacitated Vehicle Routing using ACO 4

2.1.1 External Modules 4

2.1.2 Chromosome 5

2.1.3 Initialization 6

2.1.4 Reading Input 7

2.1.5 Mathematics 8

2.1.6 Simulation 10

2.1.7 Plotting 15

2.1.8 Running 16

3 Analysis **18**

3.1 Capacitated Vehicle Routing using ACO 18

Chapter 1

Abstract

The assignment focuses on swarm intelligence and provides students hands-on with Ant Colony Optimization (ACO) and Particle Swarm Optimization (PSO) techniques to solve complex optimization problems.

Chapter 2

Problems

2.1 Capacitated Vehicle Routing using ACO

2.1.1 External Modules

There are three external modules used. Run the following command to download them.

```
pip install numpy vrplib matplotlib
```

If this does not work then use,

```
pip3 install numpy vrplib matplotlib
```

```python
import numpy as np
import vrplib
from matplotlib import pyplot as plt
```

Listing 2.1: External modules for ACO

2.1.2 Chromosome

The class `Ant_t` is used to represent the chromosome. It contains routes and distance.

```python
class Ant_t:
    """Ant Class for the Ant Colony Optimization Algorithm"""

    def __init__(self, routes: list, distance: int | float, **kwargs) -> None:
        """Constructor for the Ant Class

        Args:
            routes (list): Routes taken by the Ant
            distance (int | float): Distance travelled by the Ant
        """
        self.routes = routes
        self.distance = distance
```

Listing 2.2: Chromosome for ACO

2.1.3 Initialization

```python
class Ant_Colony_Optimization:
    """Ant Colony Optimization Algorithm Class"""

    def __init__(self, alpha: int | float, beta: int | float, iteration: int,
                 num_ants: int, rho: int | float, path: str, *args,
                 **kwargs) -> None:
        """Constructor for the Ant Colony Optimization Algorithm Class

        Args:
            alpha (int | float): Used to control the pheromone influence
            beta (int | float): Used to control the heuristic influence
            iteration (int): Number of iterations
            num_ants (int): Number of Ants
            rho (int | float): Used to control the evaporation rate
            path (str): Path to the Distance Matrix file
        """
        self.alpha: int | float = alpha  # pheromone influence
        self.beta: int | float = beta  # heuristic influence
        self.ITERATION: int = iteration  # number of iterations
        self.NUM_ANTS: int = num_ants  # number of ants
        self.evap_rate: float = 1.0 - rho  # evaporation rate
        self.path = path  # path to the instance file
```

Listing 2.3: Initilization of ACO

2.1.4 Reading Input

Note that the `read_file` does not depends on the object that's why it is a static method. This function would work when internet is avaible. If you get an error about the http request then make sure that you are connected to the internet.

```python
    @staticmethod
    def read_file(path: str, *args, **kwargs) -> dict:
        """Reads a VRP instance file.

        Args:
            path (str): Path to the instance file.

        Returns:
            dict: A dictionary containing the instance data.
        """
        vrplib.download_instance(path, f"instances_CVRP/{path}.vrp")
        return vrplib.read_instance(f"instances_CVRP/{path}.vrp")

    def extract(self, *args, **kwargs) -> None:
        """Extracts the data from the Distance Matrix file"""
        file = self.read_file(self.path)
        self.capacity = file["capacity"]
        self.depot = file["depot"][0]
        self.n = file["dimension"]
        self.demand = file["demand"]
        self.distances = file["edge_weight"]
        self.min_distance = float("inf")
        self.eta = np.reciprocal(self.distances,
                                 out=np.zeros_like(self.distances),
```

```
                                 where=self.distances != 0)
        self.min_route = None
        self.mean_distances = []
        self.tau = np.zeros((self.n, self.n))
```

Listing 2.4: Reading input for ACO

2.1.5 Mathematics

The mathematics for Ant Colony Optimization is implemented in the following functions.

```
    def evaluate_tau(self, *args, **kwargs) -> list[list]:
        """Evaluates the pheromone matrix based on the Ants' routes

        Returns:
            list[list]: Pheromone matrix
        """
        delta_tau = np.zeros((self.n, self.n))
        for ant in self.ants:
            for route in ant.routes:
                for path in range(len(route) - 1):
                    u, v = route[path], route[path + 1]
                    inverse_distance = 1 / ant.distance
                    delta_tau[u][v] += inverse_distance
                    delta_tau[v][u] += inverse_distance
        return delta_tau

    def evaluate_prob(self, current_city: int, potential_cities: list
, *args,
                      **kwargs) -> list:
```

```
    """Evaluates the probabilities of the Ants' next move based on the
    pheromone, heuristic matrices, Ant's current city and potential cities

    Args:
        current_city (int): Current city of the ant
        potential_cities (list): Potential cities the ant can move to

    Returns:
        list: Probabilities of the Ants' next move
    """
    P = list(
        map(
            lambda i: (self.tau[current_city][i]**self.alpha) +
            (self.eta[current_city][i]**self.beta), potential_cities))

    normalization_factor = 1 / sum(P)

    P = list(map(lambda x: x * normalization_factor, P))

    pp = []
    start = 0

    for i in range(len(P)):
        pp.append((start, start + P[i]))
        start += P[i]

```

```python
        return pp

    def revise_tau(self, *args, **kwargs):
        """Updates the pheromone values"""
        delta_tau = self.evaluate_tau()
        self.tau = self.tau * self.evap_rate + delta_tau
```

Listing 2.5: Mathematics for ACO

2.1.6 Simulation

All the functions related to simulation are implemented in the following functions.

```python
    def get_next_city(self, current_city: int, unvisited: list,
                      truck_capacity: int, *args, **kwargs) -> int:
        """Gets the next city the Ant will move to

        Args:
            current_city (int): Current city of the ant
            unvisited (list): List of unvisited cities
            truck_capacity (int): Truck's capacity

        Returns:
            int: Next city the Ant will move to
        """
        potential_cities = [
            city for city in unvisited
            if self.demand[city] <= truck_capacity and city != current_city
        ]

```

```
        proportional_probabilities = self.evaluate_prob(current_city, potential_cities)

        p = np.random.uniform(0, 1)

        for city in range(len(proportional_probabilities)):
            if (proportional_probabilities[city][0] <= p <
                    proportional_probabilities[city][1]):
                next_city = city
                break

        return potential_cities[next_city]

    def _simulate_Ants(self,
                       initialize: bool = False,
                       *args,
                       **kwargs) -> Ant_t:
        """Simulates the Ants

        Args:
            initialize (bool, optional): If True, the Ants will be initialized
            at the Depot and will not return to it. Defaults to False.

        Returns:
            Ant_t: _description_
        """
        total_distance = 0
```

```python
        current_city = self.depot
        truck_capacity = self.capacity
        route = []
        path = [current_city]
        unvisited = list(range(self.n))
        bound = not initialize

        if initialize:
            unvisited.pop(0)

        while bound < len(unvisited):
            if initialize:
                i = np.random.randint(len(unvisited))
                next_city = unvisited[i]

                if truck_capacity < self.demand[next_city]:
                    total_distance += self.distances[current_city][self.depot]

                    route.append(path)

                    current_city = self.depot
                    path = [self.depot]

                    truck_capacity = self.capacity
            else:
                next_city = self.get_next_city(current_city, unvisited,
                                                          truck_capacity)
            truck_capacity -= self.demand[next_city]
```

```
                total_distance += self.distances[current_city][next_city]

                current_city = next_city
                path.append(current_city)

                if initialize:
                    unvisited.pop(i)
                elif current_city == self.depot:
                    truck_capacity = self.capacity
                    route.append(path)
                    path = [self.depot]
                else:
                    unvisited.remove(current_city)

            path.append(self.depot)
            total_distance += self.distances[current_city][self.depot]
            route.append(path)

            if total_distance < self.min_distance:
                self.min_distance = total_distance
                self.min_route = route

            self.mean_distances.append(total_distance)
            return Ant_t(route, total_distance)

        def _Ant_Colony_Simulation(self,
                                   initialize: bool = False,
                                   *args,
                                   **kwargs) -> None:
            """Simulates the Ant Colony Optimization Algorithm
```

```

        Args:
            initialize (bool, optional): If True, the Ants will be initialized
            at the Depot and will not return to it. Defaults to False.
        """
        self.ants = list(
            map(lambda x: self._simulate_Ants(initialize),
                range(self.NUM_ANTS)))

    def run_simulation(self, initialize: bool = True, *args, **kwargs) -> None:
        """Runs the Ant Colony Optimization Algorithm"""

        self.extract()

        min_list = []
        mean_list = []

        self._Ant_Colony_Simulation(initialize)

        self.tau = self.evaluate_tau()

        for _ in range(self.ITERATION):

            self._Ant_Colony_Simulation(False)
            self.revise_tau()

            min_list.append(self.min_distance)
```

```
            mean_list.append(np.average(self.mean_distances))

        return min_list, mean_list
```

Listing 2.6: Simulation for ACO

2.1.7 Plotting

The plotting is done using the `matplotlib` library. The following function is used to plot the results.

```
    def plot_results(self, show: bool = False, save: bool = False) -> None:
        """Plots the results of the simulation

        Args:
            show (bool, optional): If true shows the plot. Defaults to False.
            save (bool, optional): If true saves the plot. Defaults to False.
        """
        min_list, mean_list = self.run_simulation(self.path)

        min_fitness = round(min_list[-1], 2)
        mean_fitness = round(mean_list[-1], 2)

        plt.plot(range(1, ITERATION + 1), min_list, label="minimum")
        plt.plot(range(1, ITERATION + 1), mean_list, label="mean")
        plt.axhline(y=min_fitness,
                    color="r",
                    linestyle="--",
```

```
                    label=f"min fitness: {min_fitness}")
        plt.axhline(y=mean_fitness,
                    color="g",
                    linestyle="--",
                    label=f"mean fitness: {mean_fitness}")
        plt.xlabel("Number of iteration")
        plt.ylabel("Fitness")
        plt.tight_layout()
        plt.legend()
        plt.grid(True)
        if save:
            plt.savefig(
                f"plots/{self.path}-{self.ITERATION}-{self.NUM_ANTS}.
    png")
        if show:
            plt.show()
        plt.close()
```

Listing 2.7: Plotting for ACO

2.1.8 Running

The following piece of code is used to run the program.

```
if __name__ == "__main__":
    ITERATION: int = 30
    NUM_ANTS: int = 30
    ALPHA: float = 4.0
    BETA: float = 4.0
    INITIALIZATION: bool = True
    RHO: float = 0.5

    files = ["A-n32-k5", "A-n44-k6", "A-n60-k9", "A-n80-k10"]

    ACO = Ant_Colony_Optimization(ALPHA, BETA, ITERATION, NUM_ANTS,
    RHO, None)
    for file in files:
        ACO.path = file
        ACO.plot_results(show=False, save=True)
```

Listing 2.8: Running ACO

Chapter 3

Analysis

3.1 Capacitated Vehicle Routing using ACO

The analysis is done by varying the number of ants and the number of iterations. The results are shown in the following figures. In each grid number of ants are changing from 20 to 30 horizontally and number of iterations are changing from 20 to 30 vertically. Other parameters are $\alpha = 4$, $\beta = 4$, and $\rho = 0.5$.

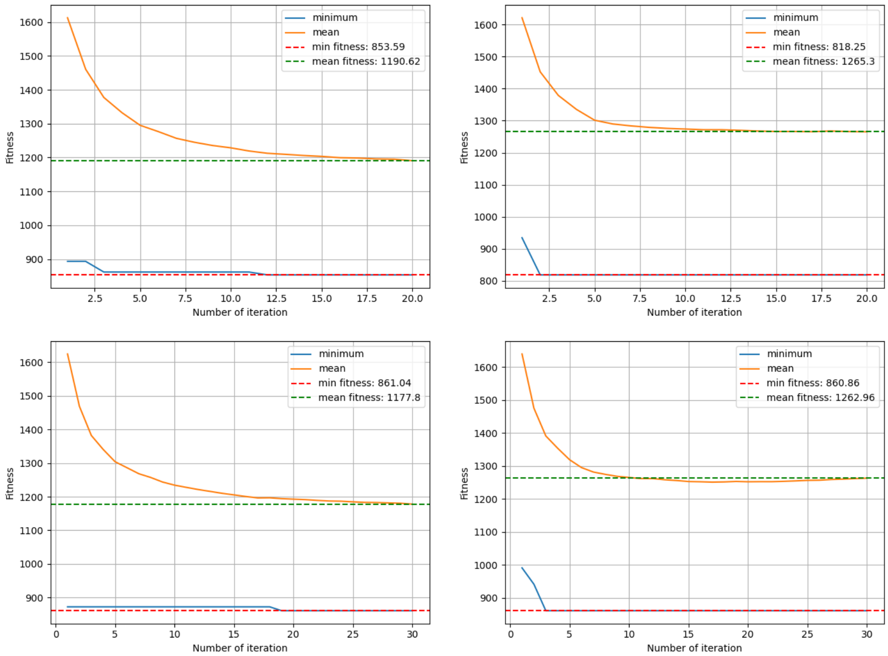

Figure 3.1: Dataset: A-n32-k5

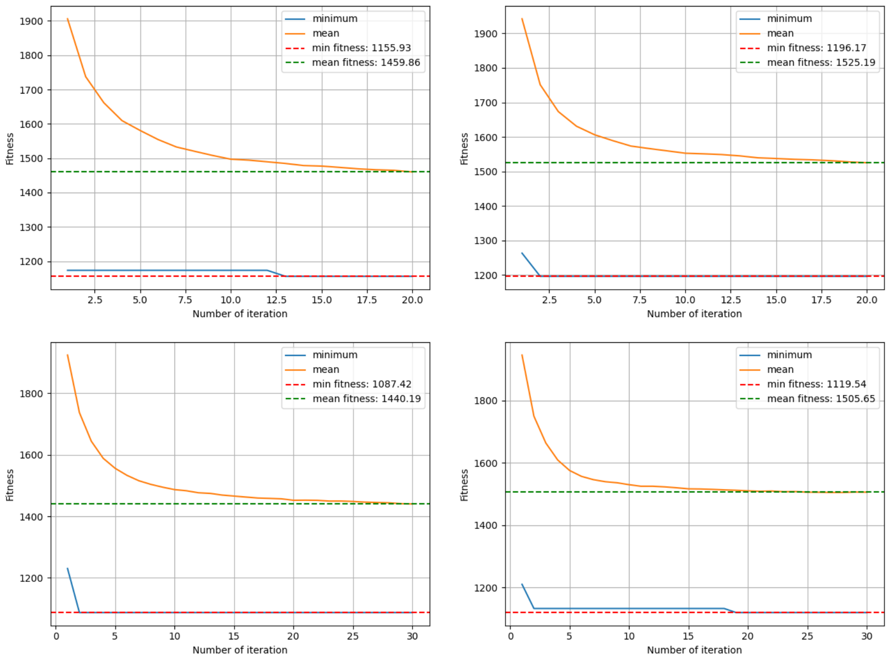

Figure 3.2: Dataset: A-n44-k6

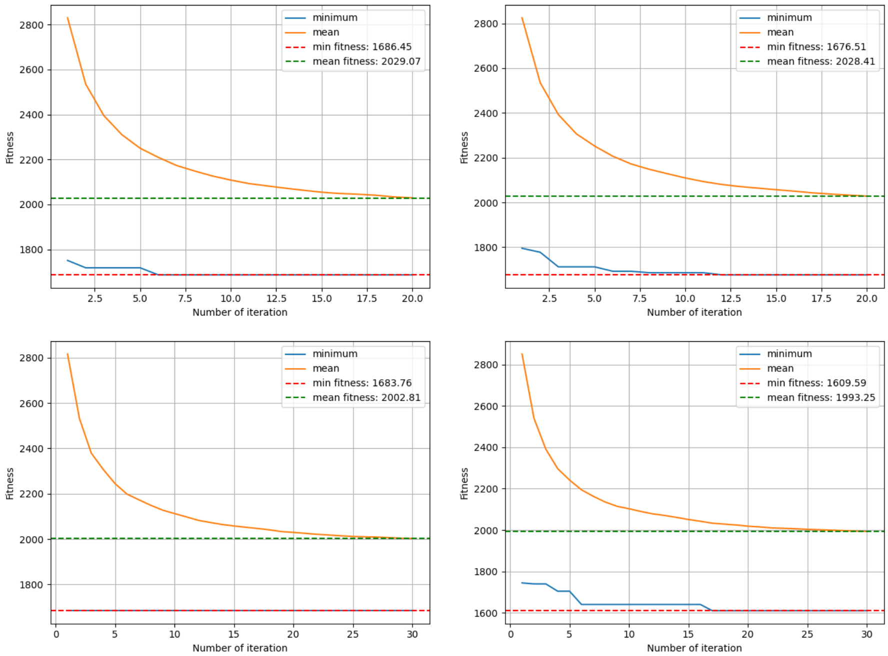

Figure 3.3: Dataset: A-n60-k9

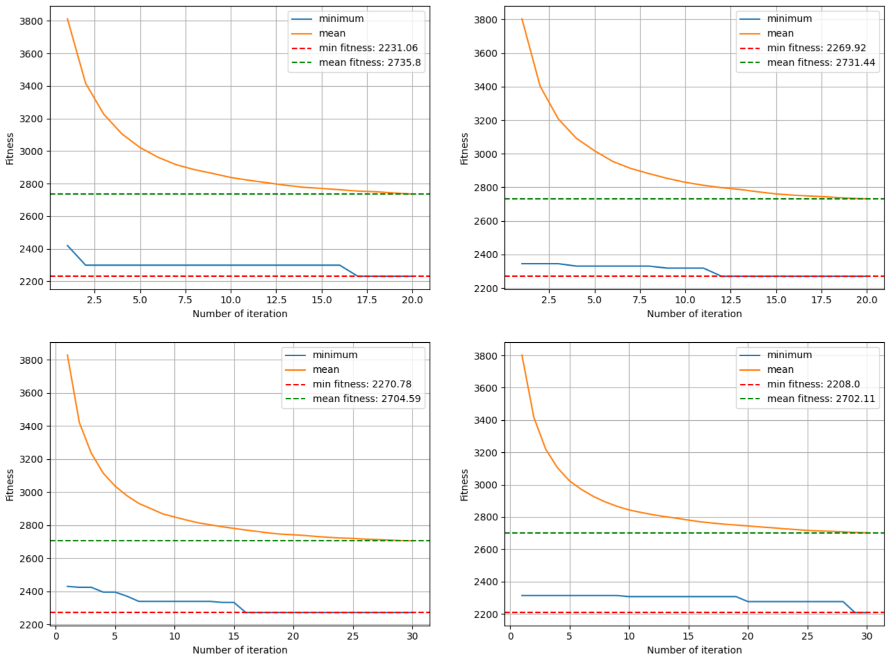

Figure 3.4: Dataset: A-n80-k10