Second Assignment: Genetic Algorithm

Riccardo Storchi

19/10/2024

Task 1: Tuning of Parameters and Description

In this task we had to solve a **Travelling Salesman Problem (TSP)**, which goal is to find, in a large solution space, the shortest path between the cities/locations that the salesman must visit in an efficient way as brute force algorithms are forbidden. Genetic algorithms are used to find solutions by mutating, combining, and selecting routes. This is done by minimizing or maximizing the fitness function (in this case we must **minimize**) until the best solution is found. In this task we have to test the algorithm with different parameters of population size and mutation rate:

- **Population size**: 10, 20, 50, 100
- **Mutation rate**: 0.1, 0.3, 0.6, 0.9

While the initial task was to run the algorithm one time for each of these values, I preferred to **run it ten times for each one of them and to make a boxplot of the fitness function** in order to contrast the randomness of the algorithm. This approach allows to see **statistics** of the fitness function and to choose an **average** one for coherency. Furthermore, I reported the **average execution time**, the **standard deviation**, and **mean fitness score**.

I decided to use a **pickle dump** to **save** the dictionary containing the **cities** with the corresponding coordinates. I did this because otherwise the cities are going to have other random coordinates. By doing this I can make coherent comparisons between the retrieved output data. I also saved every output retrieved in different **CSV files** in order to access the data and work on them without unnecessary runs of the algorithm.

Results obtained:

- **Population size of 10, mutation rate of 0.1**:

Parameter	**Value**
Average Execution Time	21.8689
Mean Fitness	652.2257
Standard Deviation	11.8929

Table 1: Summary of Results for Population size 10, Mutation rate 0.1

Output:

```
{
    'iteration': 5666,
    'fitness': np.float64(630.4599792974134),
    'cities': array([
        ['Istanbul', 'Amsterdam', 'Birmingham', 'Budapest', 'Sofia', 'Brussels',
        'Rome', 'Paris', 'Moscow', 'Munich', 'London', 'Warsaw', 'Barcelona',
        'Prague', 'Kyiv', 'Berlin', 'Milan', 'Minsk', 'Bucharest', 'Vienna']
    ], dtype='<U10'),
    'execution_time': 21.505290269851685
}
```

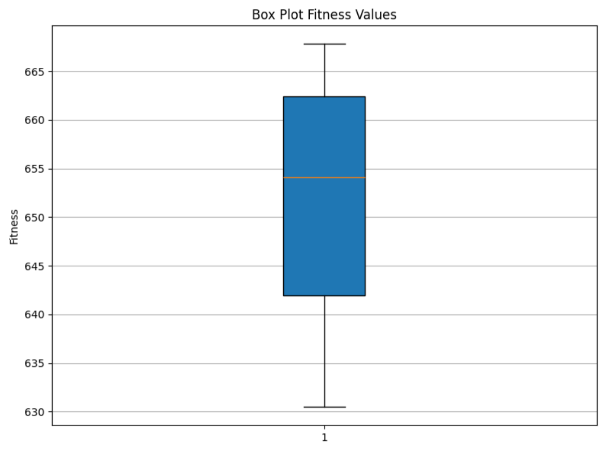

Figure 1: Box Plot of Fitness Values for Population size 10, Mutation rate 0.1

- **Population size of 10, mutation rate of 0.3**:

Parameter	Value
Average Execution Time	25.8736
Mean Fitness	608.0953
Standard Deviation	14.5660

Table 2: Summary of Results for Population size 10, Mutation rate 0.3

Output:

```
{
    'iteration': 350,
    'fitness': np.float64(621.8991473498029),
    'cities': array([
        ['Milan', 'Minsk', 'Moscow', 'Munich', 'Paris', 'Vienna', 'Barcelona',
        'Prague', 'Kyiv', 'Rome', 'Budapest', 'Birmingham', 'Amsterdam',
        'Istanbul', 'Sofia', 'Brussels', 'Warsaw', 'Bucharest', 'London', 'Berlin']
    ], dtype='<U10'),
    'execution_time': 24.632596731185917
}
```

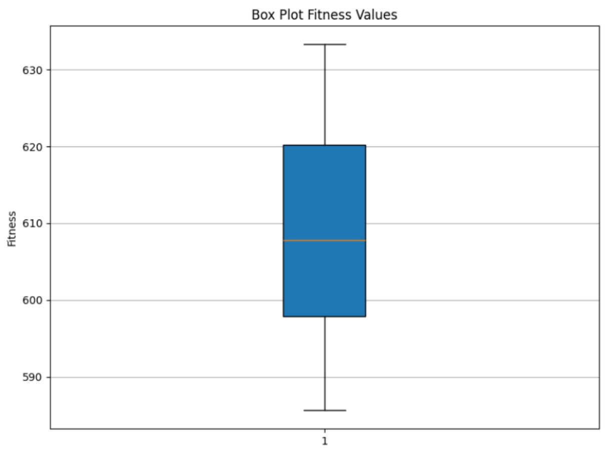

Figure 2: Box Plot of Fitness Values for Population size 10, Mutation rate 0.3

- **Population size of 10, mutation rate of 0.6**:

Parameter	Value
Average Execution Time	60.9001
Mean Fitness	605.8568
Standard Deviation	18.6374

Table 3: Summary of Results for Population size 10, Mutation rate 0.6

Output:

```
{
    'iteration': 2226,
    'fitness': np.float64(630.8359604067214),
    'cities': array([
        ['Sofia', 'Berlin', 'Prague', 'Kyiv', 'Milan', 'Brussels', 'Moscow',
        'Munich', 'Vienna', 'Paris', 'Birmingham', 'Istanbul', 'Warsaw',
        'Bucharest', 'London', 'Rome', 'Amsterdam', 'Budapest', 'Barcelona', 'Minsk']
    ], dtype='<U10'),
    'execution_time': 91.88027834892272
}
```

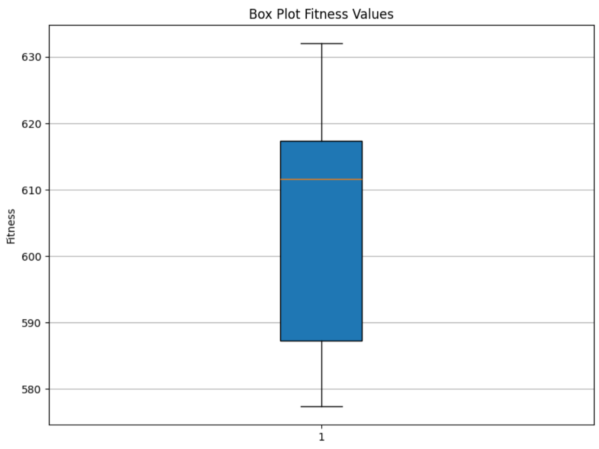

Figure 3: Box Plot of Fitness Values for Population size 10, Mutation rate 0.6

- **Population size of 10, mutation rate of 0.9**:

Parameter	Value
Average Execution Time	76.4989
Mean Fitness	595.2651
Standard Deviation	27.2850

Table 4: Summary of Results for Population size 10, Mutation rate 0.9

Output:

```
{
    'iteration': 7349,
    'fitness': np.float64(620.1695704141173),
    'cities': array([
        ['Budapest', 'Amsterdam', 'Paris', 'Munich', 'Moscow', 'London',
        'Istanbul', 'Bucharest', 'Warsaw', 'Birmingham', 'Rome', 'Barcelona',
        'Kyiv', 'Minsk', 'Vienna', 'Prague', 'Sofia', 'Milan', 'Brussels', 'Berlin']
    ], dtype='<U10'),
    'execution_time': 96.31691884994508
}
```

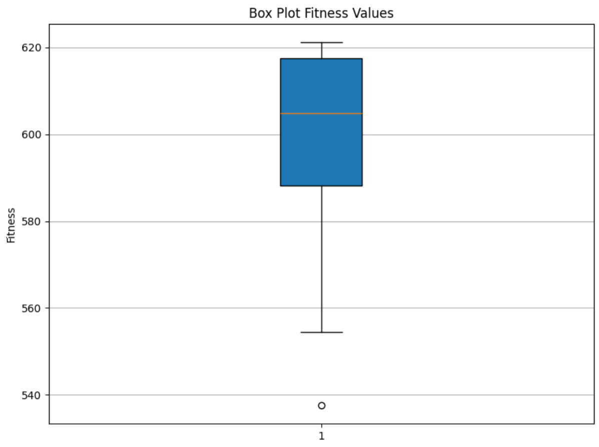

Figure 4: Box Plot of Fitness Values for Population size 10, Mutation rate 0.9

– **Population size of 20, mutation rate of 0.1**:

Parameter	Value
Average Execution Time	76.3502
Mean Fitness	576.6832
Standard Deviation	12.9946

Table 5: Summary of Results for Population size 20, Mutation rate 0.1

Output:

```
{
    'iteration': 4635,
    'fitness': np.float64(558.7257167369022),
    'cities': array([
        ['Brussels', 'Berlin', 'Barcelona', 'Minsk', 'Paris', 'Vienna', 'Rome',
        'Moscow', 'Munich', 'Bucharest', 'Warsaw', 'Birmingham', 'Amsterdam',
        'London', 'Istanbul', 'Budapest', 'Kyiv', 'Prague', 'Milan', 'Sofia']
    ], dtype='<U10'),
    'execution_time': 133.0600025653839
}
```

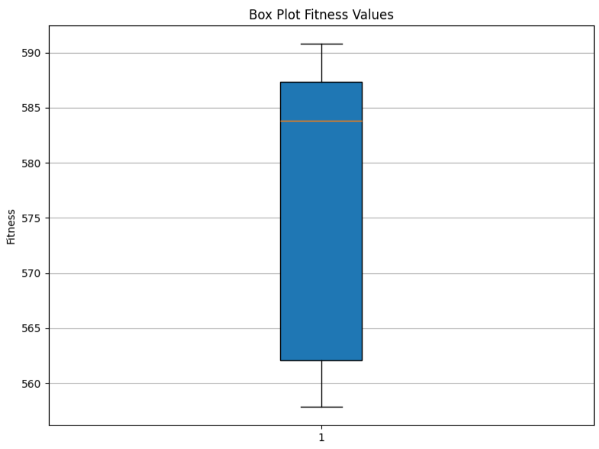

Figure 5: Box Plot of Fitness Values for Population size 20, Mutation rate 0.1

- **Population size of 20, mutation rate of 0.3**:

Parameter	**Value**
Average Execution Time	49.3846
Mean Fitness	595.2756
Standard Deviation	14.1912

Table 6: Summary of Results for Population size 20, Mutation rate 0.3

Output:

```
{
    'iteration': 7131,
    'fitness': np.float64(594.481703871397),
    'cities': array([
        ['Budapest', 'Brussels', 'Minsk', 'Prague', 'Kyiv', 'Milan', 'Bucharest',
        'Warsaw', 'London', 'Birmingham', 'Amsterdam', 'Istanbul', 'Sofia',
        'Berlin', 'Vienna', 'Moscow', 'Munich', 'Paris', 'Rome', 'Barcelona']
    ], dtype='<U10'),
    'execution_time': 49.05981779098511
}
```

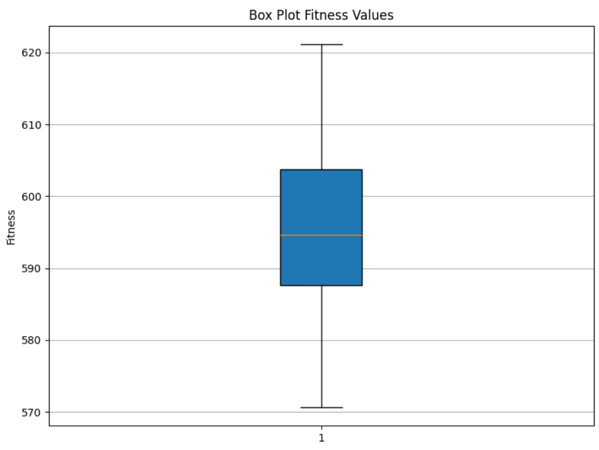

Figure 6: Box Plot of Fitness Values for Population size 20, Mutation rate 0.3

- **Population size of 20, mutation rate of 0.6**:

Parameter	Value
Average Execution Time	59.4098
Mean Fitness	598.2068
Standard Deviation	13.2472

Table 7: Summary of Results for Population size 20, Mutation rate 0.6

Output:

```
{
    'iteration': 2324,
    'fitness': np.float64(608.2663960195085),
    'cities': array([
        ['Bucharest', 'Istanbul', 'Amsterdam', 'Budapest', 'Prague', 'Kyiv', 'Minsk',
        'Paris', 'Rome', 'Milan', 'Sofia', 'Birmingham', 'London', 'Warsaw',
        'Moscow', 'Munich', 'Brussels', 'Vienna', 'Berlin', 'Barcelona']
    ], dtype='<U10'),
    'execution_time': 61.47695517539978
}
```

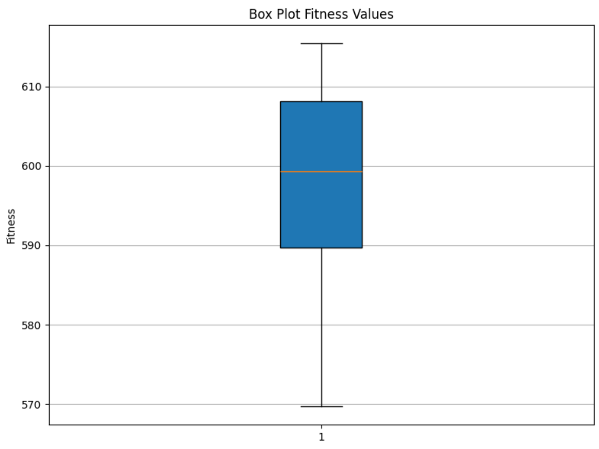

Figure 7: Box Plot of Fitness Values for Population size 20, Mutation rate 0.6

- **Population size of 20, mutation rate of 0.9**:

Parameter	Value
Average Execution Time	64.9067
Mean Fitness	590.1138
Standard Deviation	13.7827

Table 8: Summary of Results for Population size 20, Mutation rate 0.9

Output:

```
{
    'iteration': 2166,
    'fitness': np.float64(603.3255145894566),
    'cities': array([
        ['Bucharest', 'London', 'Amsterdam', 'Rome', 'Barcelona', 'Milan', 'Budapest',
        'Birmingham', 'Istanbul', 'Sofia', 'Prague', 'Berlin', 'Brussels', 'Vienna',
        'Moscow', 'Warsaw', 'Munich', 'Paris', 'Kyiv', 'Minsk']
    ], dtype='<U10'),
    'execution_time': 66.17883563041687
}
```

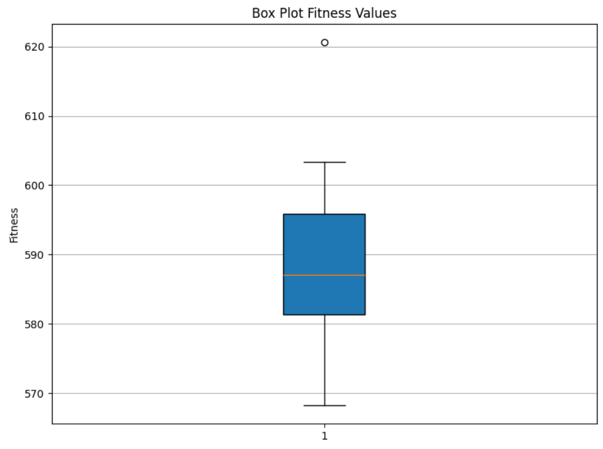

Figure 8: Box Plot of Fitness Values for Population size 20, Mutation rate 0.9

- **Population size of 50, mutation rate of 0.1**:

Parameter	**Value**
Average Execution Time	241.4917
Mean Fitness	575.3171
Standard Deviation	13.8532

Table 9: Summary of Results for Population size 50, Mutation rate 0.1

Output:

```
{
    'iteration': 6226,
    'fitness': np.float64(576.9118494134457),
    'cities': array([
        ['Paris', 'Barcelona', 'Minsk', 'Kyiv', 'Prague', 'Brussels', 'Rome',
        'Amsterdam', 'Budapest', 'Istanbul', 'Birmingham', 'Warsaw',
        'Bucharest', 'London', 'Moscow', 'Vienna', 'Munich', 'Milan',
        'Berlin', 'Sofia']
    ], dtype='<U10'),
    'execution_time': 123.64126324653624
}
```

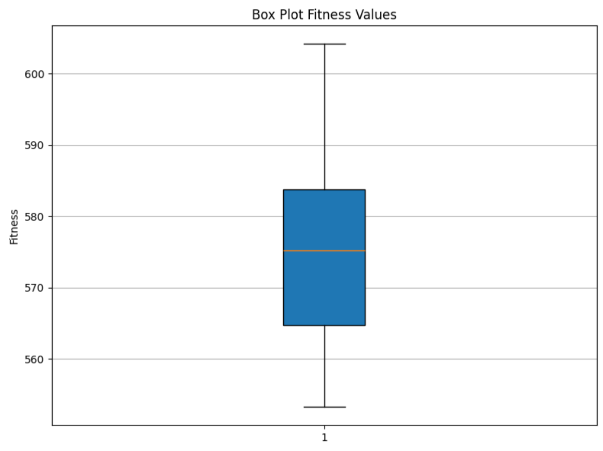

Figure 9: Box Plot of Fitness Values for Population size 50, Mutation rate 0.1

- **Population size of 50, mutation rate of 0.3**:

Parameter	**Value**
Average Execution Time	186.7700
Mean Fitness	578.3441
Standard Deviation	23.3791

Table 10: Summary of Results for Population size 50, Mutation rate 0.3

Output:

```
{
    'iteration': 3030,
    'fitness': np.float64(575.6454351607146),
    'cities': array([
        ['Munich', 'Paris', 'Vienna', 'Rome', 'Moscow', 'London', 'Istanbul',
        'Bucharest', 'Warsaw', 'Birmingham', 'Budapest', 'Brussels',
        'Amsterdam', 'Sofia', 'Milan', 'Barcelona', 'Prague', 'Kyiv',
        'Minsk', 'Berlin']
    ], dtype='<U10'),
    'execution_time': 401.8847532272339
}
```

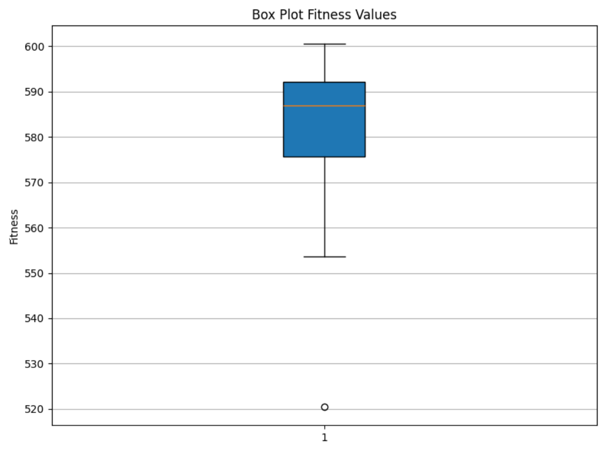

Figure 10: Box Plot of Fitness Values for Population size 50, Mutation rate 0.3

- **Population size of 50, mutation rate of 0.6**:

Parameter	Value
Average Execution Time	154.0989
Mean Fitness	585.0055
Standard Deviation	18.2449

Table 11: Summary of Results for Population size 50, Mutation rate 0.6

Output:

```
{
    'iteration': 6890,
    'fitness': np.float64(607.1688243000598),
    'cities': array([
        ['Milan', 'Sofia', 'Minsk', 'Moscow', 'Paris', 'Munich', 'Warsaw',
        'Bucharest', 'London', 'Birmingham', 'Amsterdam', 'Istanbul',
        'Vienna', 'Prague', 'Rome', 'Brussels', 'Budapest', 'Kyiv',
        'Barcelona', 'Berlin']
    ], dtype='<U10'),
    'execution_time': 155.28576016426086
}
```

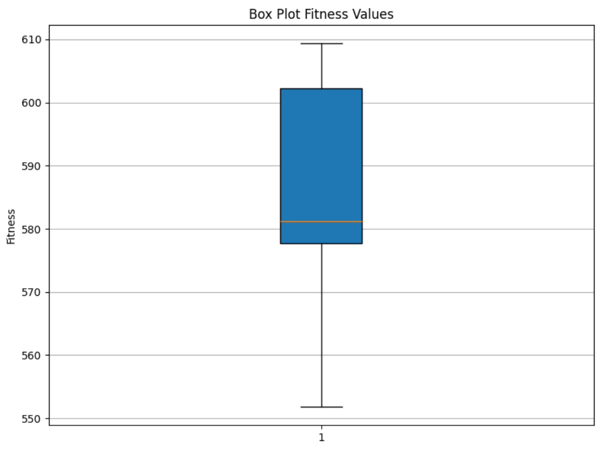

Figure 11: Box Plot of Fitness Values for Population size 50, Mutation rate 0.6

- **Population size of 50, mutation rate of 0.9**:

Parameter	Value
Average Execution Time	174.6518
Mean Fitness	584.1979
Standard Deviation	7.2642

Table 12: Summary of Results for Population size 50, Mutation rate 0.9

Output:

```
{
    'iteration': 5585,
    'fitness': np.float64(590.8957094763584),
    'cities': array([
        ['Paris', 'Prague', 'Minsk', 'Kyiv', 'Vienna', 'Berlin', 'Sofia',
        'Budapest', 'Milan', 'Brussels', 'Moscow', 'Warsaw', 'Bucharest',
        'London', 'Birmingham', 'Istanbul', 'Amsterdam', 'Munich', 'Rome',
        'Barcelona']
    ], dtype='<U10'),
    'execution_time': 180.8912508487701
}
```

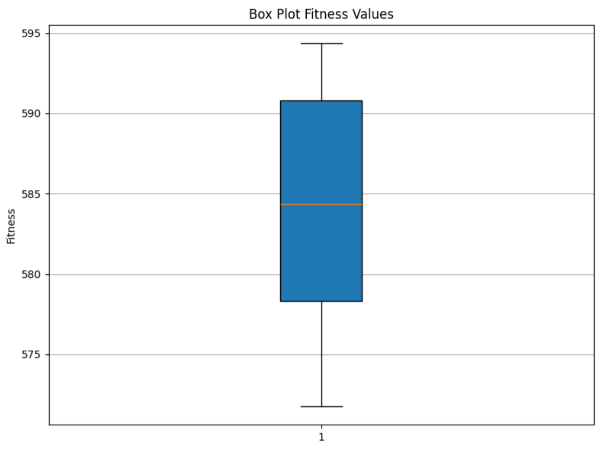

Figure 12: Box Plot of Fitness Values for Population size 50, Mutation rate 0.9

- **Population size of 100, mutation rate of 0.1**:

Parameter	**Value**
Average Execution Time	1194.6624
Mean Fitness	577.6858
Standard Deviation	16.1486

Table 13: Summary of Results for Population size 100, Mutation rate 0.1

Output:

```
{
    'iteration': 1705,
    'fitness': np.float64(579.8362553270352),
    'cities': array([
        ['Milan', 'Berlin', 'Barcelona', 'Vienna', 'Sofia', 'Rome', 'Budapest',
        'Istanbul', 'Birmingham', 'Amsterdam', 'Brussels', 'Prague', 'Kyiv',
        'Paris', 'Minsk', 'Moscow', 'Warsaw', 'Munich', 'Bucharest', 'London']
    ], dtype='<U10'),
    'execution_time': 606.2447619438171
}
```

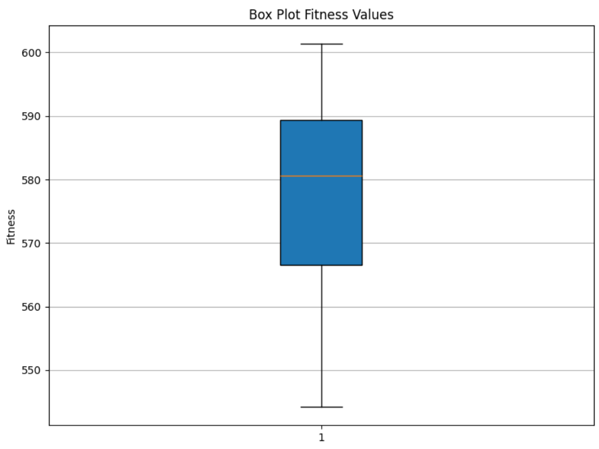

Figure 13: Box Plot of Fitness Values for Population size 100, Mutation rate 0.1

- **Population size of 100, mutation rate of 0.3**:

Parameter	**Value**
Average Execution Time	539.0293
Mean Fitness	580.8850
Standard Deviation	11.2966

Table 14: Summary of Results for Population size 100, Mutation rate 0.3

Output:

```
{
    'iteration': 9682,
    'fitness': np.float64(587.3067749754663),
    'cities': array([
        ['Barcelona', 'Brussels', 'Vienna', 'Rome', 'Munich', 'Paris', 'London',
        'Amsterdam', 'Birmingham', 'Bucharest', 'Warsaw', 'Moscow', 'Minsk',
        'Kyiv', 'Berlin', 'Prague', 'Milan', 'Budapest', 'Istanbul', 'Sofia']
    ], dtype='<U10'),
    'execution_time': 701.2134182453156
}
```

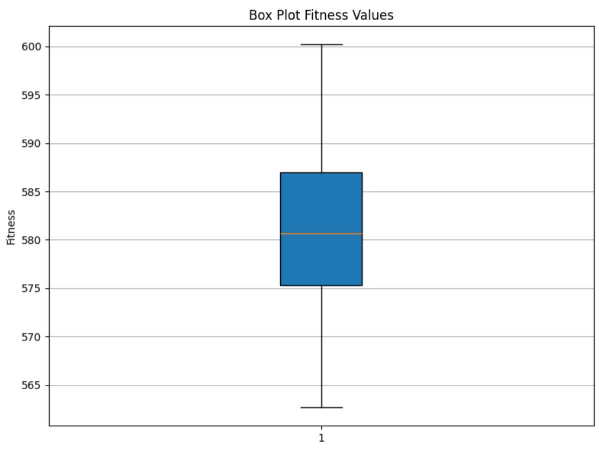

Figure 14: Box Plot of Fitness Values for Population size 100, Mutation rate 0.3

- **Population size of 100, mutation rate of 0.6**:

Parameter	Value
Average Execution Time	298.3275
Mean Fitness	565.0690
Standard Deviation	12.7553

Table 15: Summary of Results for Population size 100, Mutation rate 0.6

Output:

```
{
    'iteration': 5381,
    'fitness': np.float64(574.136597638938),
    'cities': array([
        ['Budapest', 'Sofia', 'Berlin', 'Barcelona', 'Milan', 'Amsterdam',
        'Birmingham', 'London', 'Prague', 'Kyiv', 'Brussels', 'Rome',
        'Vienna', 'Minsk', 'Paris', 'Munich', 'Moscow', 'Bucharest',
        'Istanbul', 'Warsaw']
    ], dtype='<U10'),
    'execution_time': 278.0203351974488
}
```

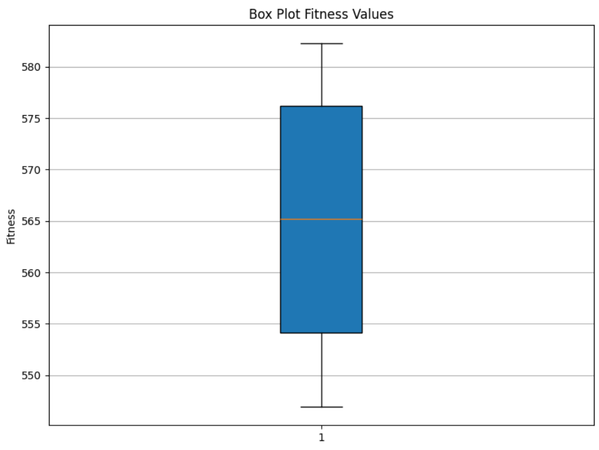

Figure 15: Box Plot of Fitness Values for Population size 100, Mutation rate 0.6

- **Population size of 100, mutation rate of 0.9**:

Parameter	Value
Average Execution Time	315.5008
Mean Fitness	568.6143
Standard Deviation	19.2909

Table 16: Summary of Results for Population size 100, Mutation rate 0.9

Output:

```
{
    'iteration': 7638,
    'fitness': np.float64(552.612844834584),
    'cities': array([
        ['Sofia', 'Rome', 'Birmingham', 'Bucharest', 'Warsaw', 'London',
        'Budapest', 'Istanbul', 'Amsterdam', 'Moscow', 'Brussels', 'Vienna',
        'Milan', 'Berlin', 'Barcelona', 'Prague', 'Kyiv', 'Minsk', 'Paris',
        'Munich']
    ], dtype='<U10'),
    'execution_time': 320.1178410053253
}
```

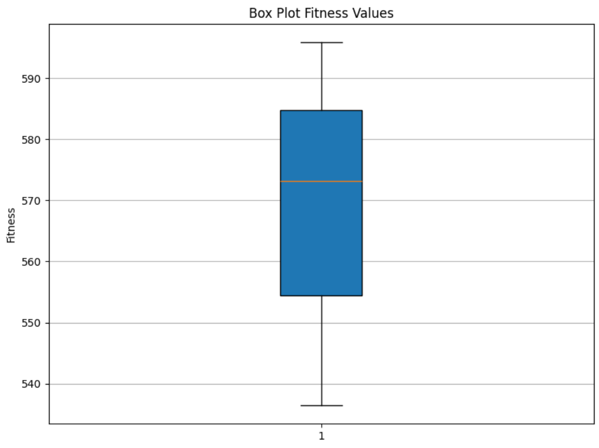

Figure 16: Box Plot of Fitness Values for Population size 100, Mutation rate 0.9

Key Insights from Experimental Results

The parameter tuning of the genetic algorithm shows clear trends. With a population size of **10** and mutation rate of **0.1**, the algorithm achieved the **best balance between mean fitness** (652.2) and **execution time** (21.9 seconds), demonstrating **efficiency** in fitness optimization and speed. **Increasing the mutation rate** to 0.3 or higher led to a **decline in fitness** and a **substantial increase in computational time**, without noticeable benefits. For population size of 20, mutation rates from 0.1 to 0.9 showed only marginal improvements in fitness while execution times ranged from 49 to 76 seconds, indicating diminishing returns. With a population size of 50 or 100, the mean fitness values stagnated around 575-580, while execution times escalated drastically to over 200-500 seconds. Ultimately, **larger population sizes and higher mutation rates failed to improve fitness meaningfully and significantly increased computational demands. The most effective combination was a small population (10) and a low mutation rate (0.1), optimizing both fitness results and computational efficiency**.

Task 2: Evaluation through Fitness Function

The fitness function is:

```python
def fitness_function(x, y, z):
    a = 2 * x * z * np.exp(-x)
    b = -2 * y * y * y
    c = y * y
    d = -3 * z * z * z
    e = np.cos(x * z) / (1 + np.exp(-(x + y)))

    fitness = a + b + c + d + e

# In the algorithm there was an error: sometimes the fitness function was not in the
    acceptable range.
#In order to avoid it, I put a really small fitness function that can't be considered as
     the best in any case as all the other are above 1.

    if fitness <= 0:
        fitness = 0.001
    return fitness
```

This is the fitness function that I implemented based on the function provided in the directive.
I applied some **changes** to the code **to address the problem of a negative fitness function** and of the **best fitness function**:

- To solve the problem for negative values of fitness function I put an **if** statement in the code that checks if the fitness value is **negative or zero**, and in that case, **sets it to 0.001** as the **fitness score**. Another implementation could have been the use of the absolute value of the fitness function.
- I modified the code responsible for **selecting the best fitness function** to ensure it consistently picks the **highest fitness value** at each generation. Additionally, it now prints the **current best fitness** value, as the algorithm originally did before my changes. **Previously, the code would always select the last fitness function, regardless of whether it was the best**.

I made this adjustment because the primary goal of a Genetic Algorithm **(GA)** is to optimize the fitness function (maximizing it in our case). If the purpose of the exercise was simply to identify the best fitness function for the final generation (which may not necessarily be the best one), the original code would have been sufficient. However, if the aim is to consistently **maximize the fitness function**, and find the best one considering all the generations, the approach I implemented is the most suitable.

Task 3: Genetic Algorithm for Maze Solving

This Python code implements a **genetic algorithm (GA)**, a technique developed before the advent of **Convolutional Neural Networks (CNNs)**, to solve a maze where open paths are represented by 1 and walls by 1000.

The algorithm defines key parameters, including: **Population size, Mutation rate, Fitness evaluation criteria**

The maze's **start and end positions** are also specified. A visualization function displays the solution path.

The genetic algorithm operates on a **population of potential paths**, referred to as **individuals**, each represented by a **series of moves**. Initial individuals are generated by exploring valid paths from the start until they hit a wall or reach the end.

To **prevent backtracking**, the algorithm imposes movement constraints that ensure paths evolve in a forward direction.

Fitness Evaluation and Selection:

A **fitness function** evaluates each individual's path based on:

- **Path length**
- **Proximity to the endpoint** (rewarding shorter paths and penalizing longer ones)

Selection is performed using **roulette wheel selection**, allowing fitter individuals to contribute more to the next generation.

Crossover and Mutation:

Crossover combines pairs of parent paths to create offspring, while **mutation** introduces variability to prevent stagnation. I designed an **adaptive mutation rate** inspired by the learning rate used in CNNs, which emerged around 1990.

The formula used is:

$$\text{MUTATION\_RATE} = \text{INITIAL\_MUTATION\_RATE} \cdot \exp(-k \cdot \text{gen}) \tag{1}$$

The approach begins with a **higher mutation rate** that **keeps decreasing** for every generation. To **prevent a vanishing mutation rate*** when the algorithm reaches a higher number of generations, I imposed a condition: after **300 generations**, the **mutation rate becomes 0.01**.

Population Reinitialization and Convergence:

This strategy emphasizes the significance of the **top 10 parents** through **Elitism**, as crossover between them is likely to yield better results than mutation when they are close to the solution.

Additionally, the algorithm incorporates a **population reinitialization mechanism** to address situations where it may become stuck at a specific point, as increasing the mutation rate alone may not resolve stagnation. This careful tuning of parameters ensures that the algorithm effectively converges toward optimal solutions.

The final result is visualized to illustrate the path found through the maze, showcasing the algorithm's performance.

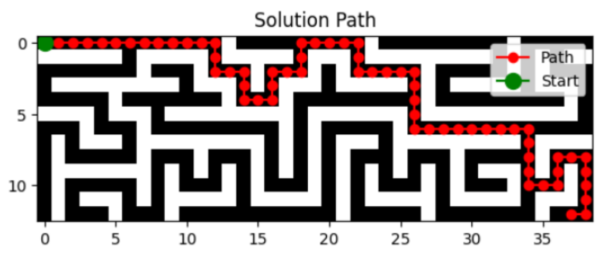

Figure 17: Example of an execution: best solution for the maze reached

Task 4: Evaluation of Regression Models

At the beginning, I import the necessary libraries to handle the data and models. Then, I generate the dataset, which is based on a quadratic relationship with added noise, and split it into **training (80%)** and **testing (20%)** sets, as requested. Next, I define and evaluate three different models:

1. **Linear Regression**: This model is simple and assumes a linear relationship between the input and output. The code includes four steps: **defining the linear regression model**, **fitting** it to the training data, **making predictions** on the test set, and calculating the **Mean Squared Error (MSE)** to assess its performance. As expected, the linear model struggles because it cannot capture the underlying quadratic relationship in the data.

2. **Polynomial Regression (Degree 2)**: The polynomial model is more flexible than the linear one. It transforms the original feature by adding polynomial terms up to degree 2. The transformed feature set consists of:

   - The original feature x (degree 1),
   - The squared version of the feature x^2 (degree 2),
   - A constant term for the intercept (bias term), which is 1.

   By adding these polynomial terms to the feature set, the model can better capture the quadratic nature of the data. After fitting the model, I calculate and print the MSE to measure its performance. As expected, **the polynomial model performs better than the linear one**, as it aligns more closely with the underlying data structure.

3. **Neural Network**: I implemented a simple **neural network (NN)** with **three layers**: an **input** layer, a **hidden layer** with **6 neurons** and **ReLU** activation, and an **output layer**. The model is trained similarly to the other models, with predictions made on the test set, and its MSE is calculated. Although the neural network is more flexible than both linear and polynomial regression, it doesn't perform as well in this case because it lacks sufficient complexity (i.e., it has too few layers) and is trained on a relatively small dataset. This leads the network to **perform similarly to the polynomial model**, but it still requires more data and tuning to reach its full potential.

At the end, I print the MSE for all three models to compare their performances. The **polynomial regression model performs the best**, as it accurately models the quadratic nature of the data. The **linear model performs poorly**, as expected, due to its inability to capture non-linear relationships. The **neural network**, although **flexible**, **needs more layers and a larger dataset to improve its performance** and prevent overfitting.

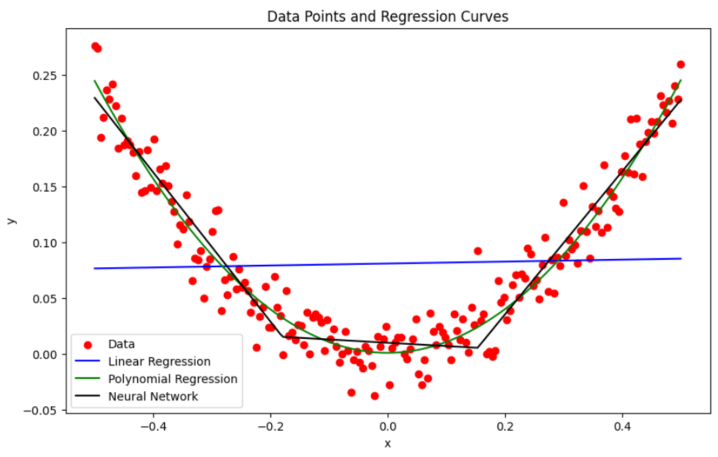

Figure 18: Plot of the regression curves of the models

References

[1] Ramez Shendy, "Traveling Salesman Problem (TSP) using Genetic Algorithm (Python)," available at: https://medium.com/aimonks/traveling-salesman-problem-tsp-using-genetic-algorithm-fea640713758.

[2] ShendoxParadox, "TSP-Genetic-Algorithm," available at: https://github.com/ShendoxParadox/TSP-Genetic-Algorithm?source=post_page-----fea640713758-------------------------------- .

[3] Muhammad Talha, "Maze Solver(Genetic Algorithm)," available at: https://medium.com/@muhammadtalha1735164/maze-solver-genetic-algorithm-2bfca68897.

[4] Nitin Choubey, "A-Mazer with Genetic Algorithm," available at: https://www.researchgate.net/publication/233786685_A-Mazer_with_Genetic_Algorithm.

[5] argonaut, "What are Genetic Algorithms?," available at: https://www.youtube.com/watch?v=XP2sFzp2Rig&t=361s.

[6] scikit-learn.org, "Linear Models," available at: https://scikit-learn.org/stable/modules/linear_model.html#ridge-regression-and-classification.

[7] Tensorflow, "Documentation," available at: https://www.tensorflow.org/api_docs/python/tf/keras.

[8] OpenAI, "ChatGPT," available at: https://chatgpt.com/.

[9] Politecnico di Milano, previous knowledge acquired during the lectures of the Artificial Neural Network course.

[10] Blekinge Tekniska Högskola , lecture material.