Assignment 2
GENETIC ALGORITHM, GRAPH SEARCH & DATA SCIENCE - DV2618

Arnau Claramunt Soler - Oct 16, 2024

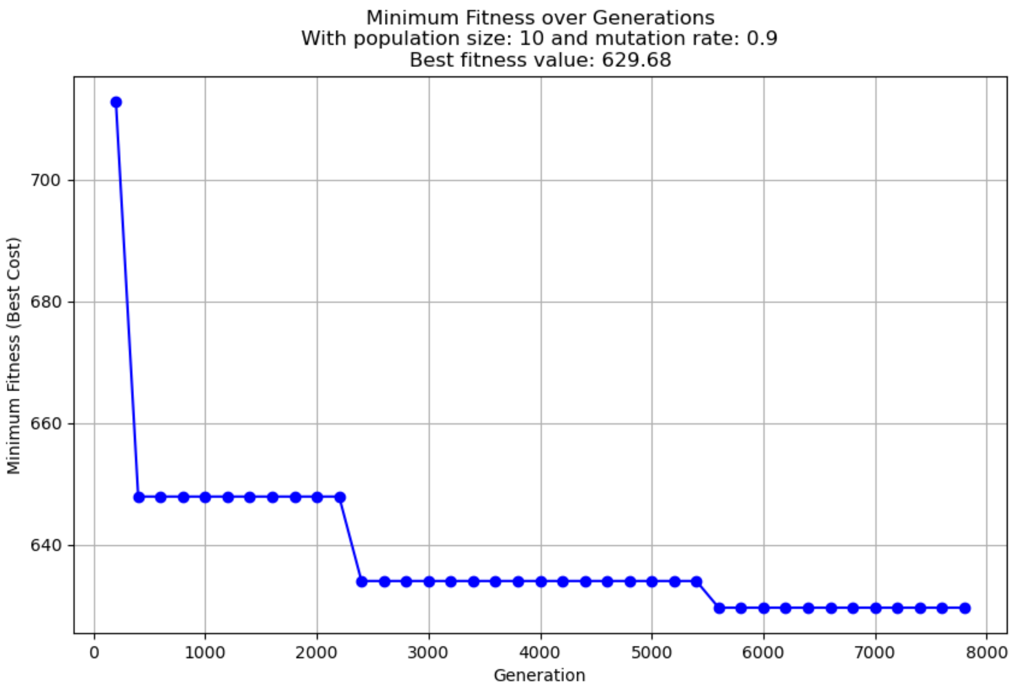
1 Task 1 - Traveling Salesman Problem, solved using GA

1.1 Results

To show the results, I plotted the generations vs minimum fitness values. And showing also the result at the end of the iteration of the algorithm with the minimum value. The number of iterations was fixed to 8000 for all the iterations.

Each result changes only one parameter at a time. First fixing the population size of 10 and changing the mutation rate.

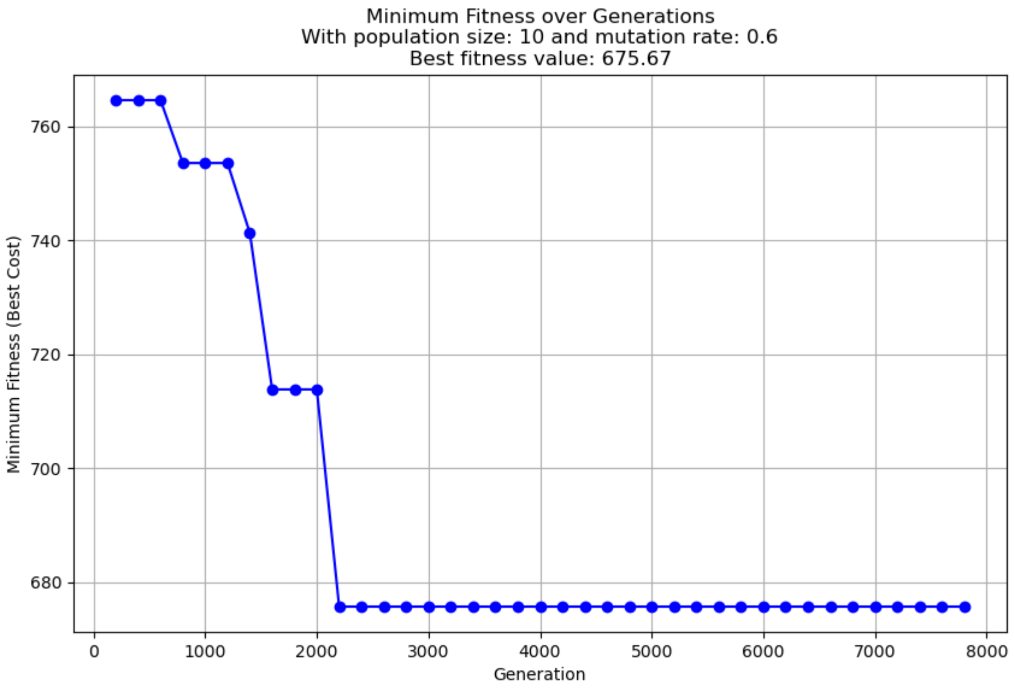
Population Size: 10
Mutation Rate: 0.9



This is the plot after the execution, with 8000 generations. Starts rapidly decreasing the fitness (cost of the path of the cities) and after 2000 generations it gets lower than 640. After the initial improvement, the fitness value remains relatively stable at around 630, particularly between generation 2300 and 5500. And due to the high mutation rate, I think it allows some more exploration. Finally, after generation 5500 it seems that it arrived to a local minima. Converging and having the final result of 629.68.

The progression over time is decreasing which shows the genetic algorithm is obtaining better results.

Population Size: 10
Mutation Rate: 0.6

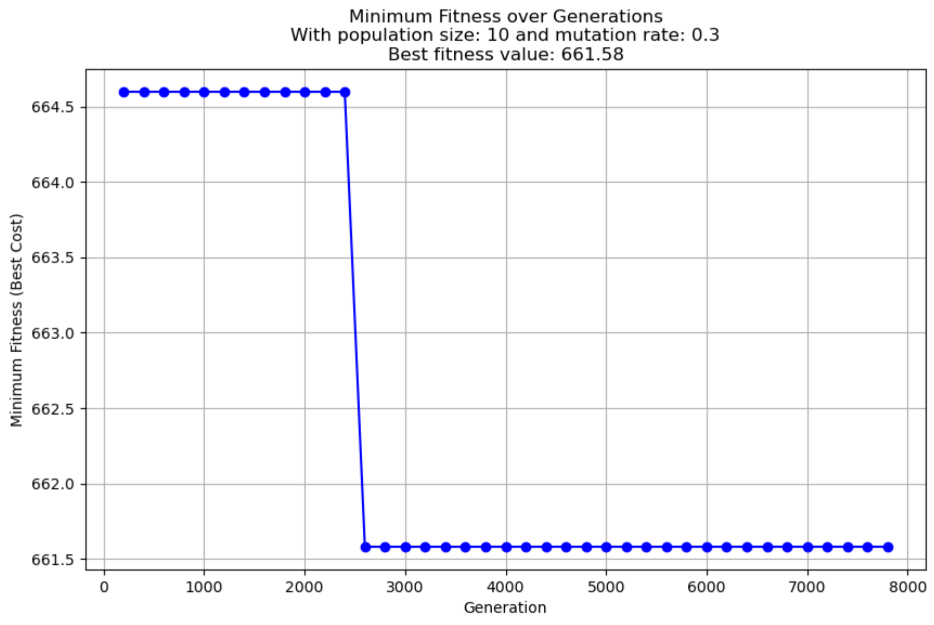


In this second execution, the decrease in fitness is slower in the beginning, staying at a cost higher than 700 for nearly 2000 generations, which in the previous mutation rate of 0.9 was already at 650. The most visual result is the stagnation of fitness at 675 after generation 2000. The final solution is 675.67 and was found way faster than the previous mutation rate.

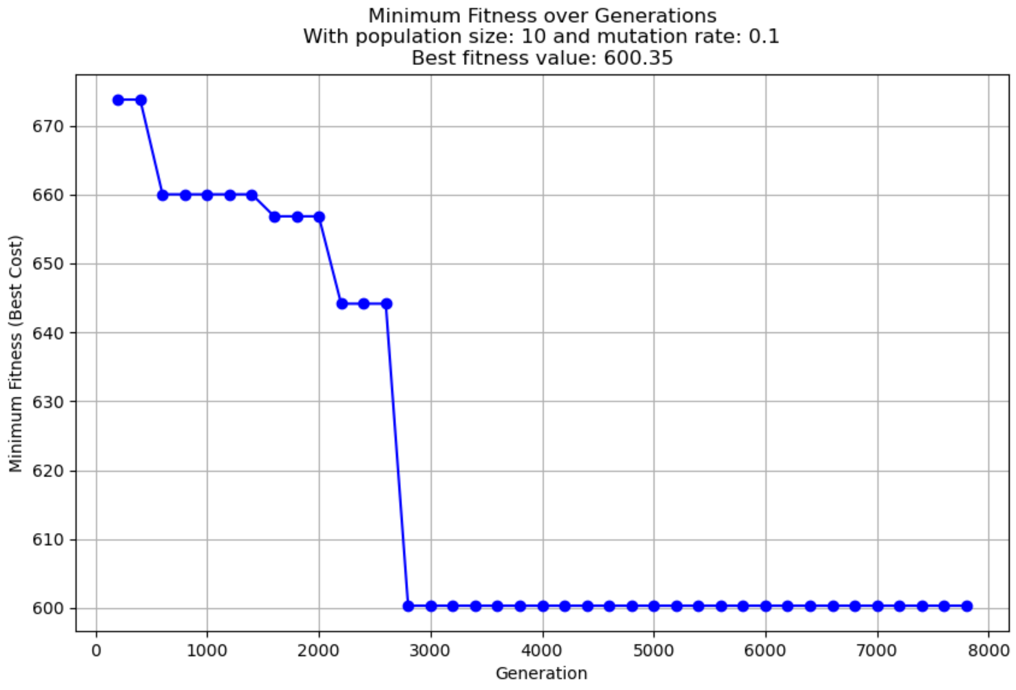
As with the previous run, the algorithm quickly settles into a stable state, which is expected given the small population size.

Population Size: 10
Mutation Rate: 0.3

This execution looks more like a step function. but the results are good. It initially stays in a good initial value of 670 and stays stable around 2500 generations. And then there is a sharp drop in fitness, resulting with the value that will be the best found at 661.58, and continuing with this value the rest of the generations. In summary, the smaller mutation rate allows the algorithm to explore the solution space more conservatively. The slower initial progress suggests the mutation rate was not high enough to explore as much early on, but the algorithm still converged to a good solution by the end.



Population Size: 10
Mutation Rate: 0.1



Now we are in the smallest mutation rate of all, and the results didn't change as much as I expected, compared with the other executions. In this case it has more of a stair decrease starting at around 680 and with 2750 generations, it finally settles at 600 fitness score. With a best score of 600.35.

Overall, for this set of executions with population size of 10, which seems a small value and probably limits the genetic diversity. A small population means the algorithm doesn't explore many different solutions, which can lead to premature convergence. Basically, the population becomes too similar too quickly, and the algorithm gets stuck in a local minimum.

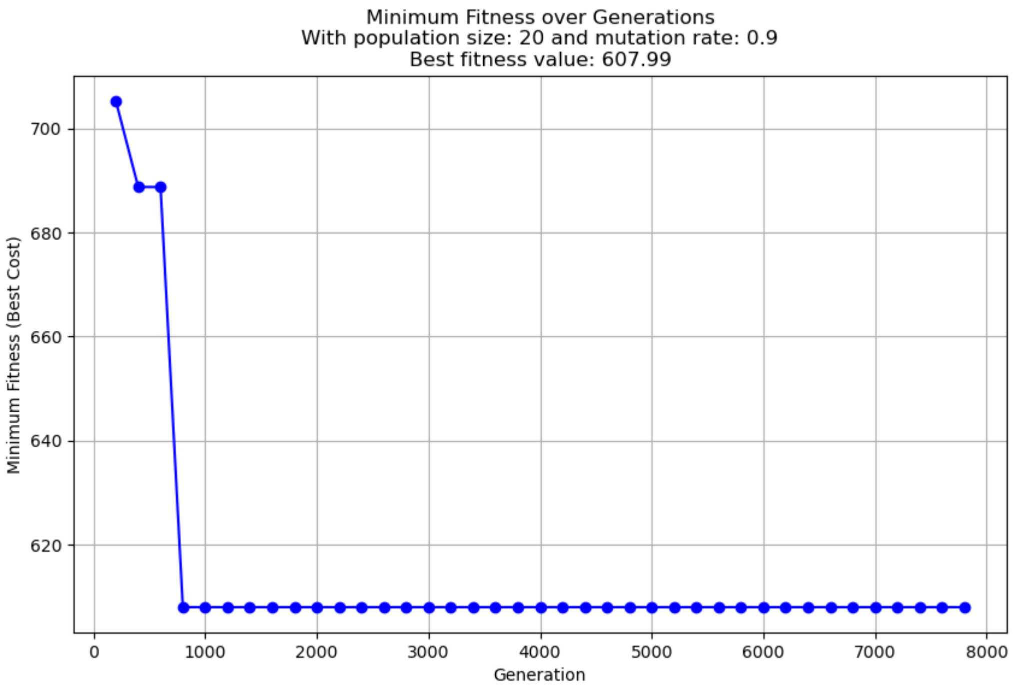
- **Higher mutation rates** (0.9) encouraged more exploration early on, allowing the algorithm to find lower fitness values but often getting stuck in local minima.
- **Moderate mutation rates** (0.6, 0.3) led to slower progress but more consistent behavior with fewer changes.
- **Low mutation rates** (0.1) resulted in more steady improvement and yielded the best final result. This approach required cautious exploration but ultimately obtained the minimum fitness.

In general, while a higher mutation rate favors early exploration, it can prevent the algorithm from settling into an optimal solution efficiently, whereas lower mutation rates allow for a more stable, refined search.

Now with population size of 20:

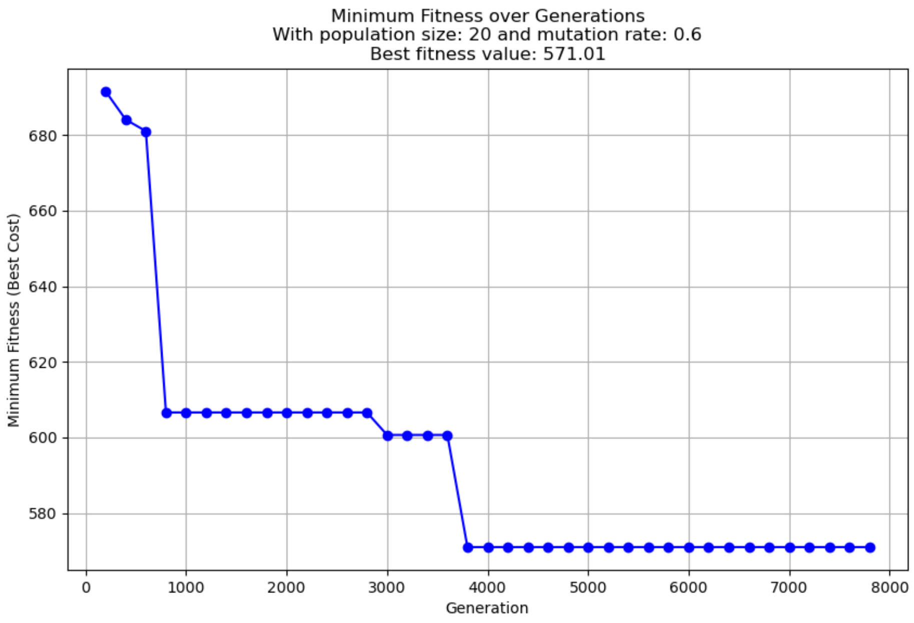
After explaining in detail the population size with value 10, I will analyze the results more superficially, because they might be similar, and can be repetitive.

Population Size: 20
Mutation Rate: 0.9



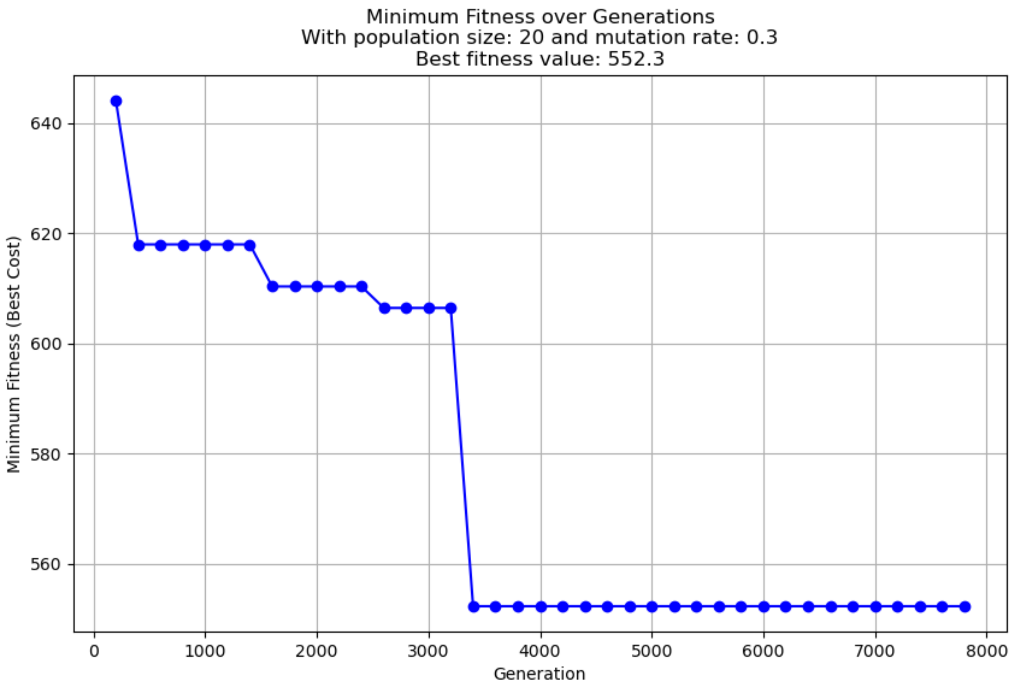
Overall, the plot, seems to be very stable in a local minima. Decreasing very fast in the first 800 iterations and dropping to 607, which stays constant for more than 7000 generations. Even with a high mutation rate, there was no mutation that helped get out of an early low result, and it seems that increasing the number of generations in this case, it's not going to work. The final best result is 607.99 which is not bad compared with all the values from population size 10, and this result was achieved fast.

Population Size: 20
Mutation Rate: 0.6



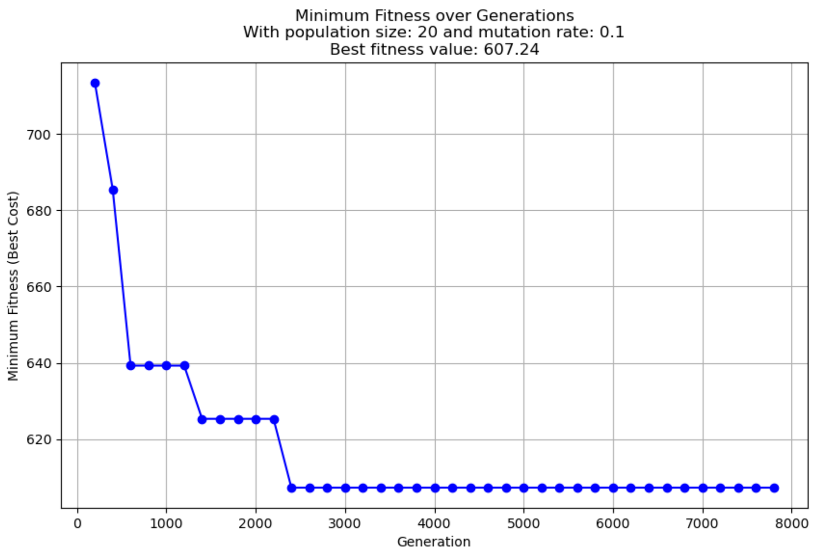
In this case the main difference is located between the generations 800 i 3500, when the fitness is stagnant at 610, and is after the 3000 generation when starts the improve and finally finds a good result with a value of 571.01

Population Size: 20
Mutation Rate: 0.3



This plot looks promising compared with all the other ones, where there is a slow but constant improvement since generation 300, and at low values. But everything changes when after generation 3100 an outstanding minimum value is found and it drops down to 552.3. Being the best result so far.

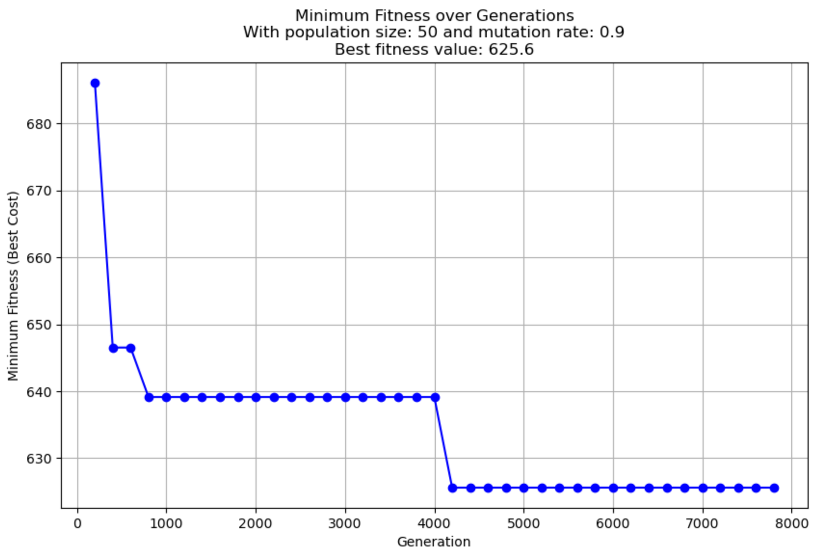
Population Size: 20
Mutation Rate: 0.1



For the last mutation rate value, now we have a rapid descent between generation 0 and 2100, and after, the minimum fitness is found staying in that value until the last generation. The final best fitness value is 607.24, which seemed better, after seeing the initial decrease, but didn't obtain better values than the previous parameters.

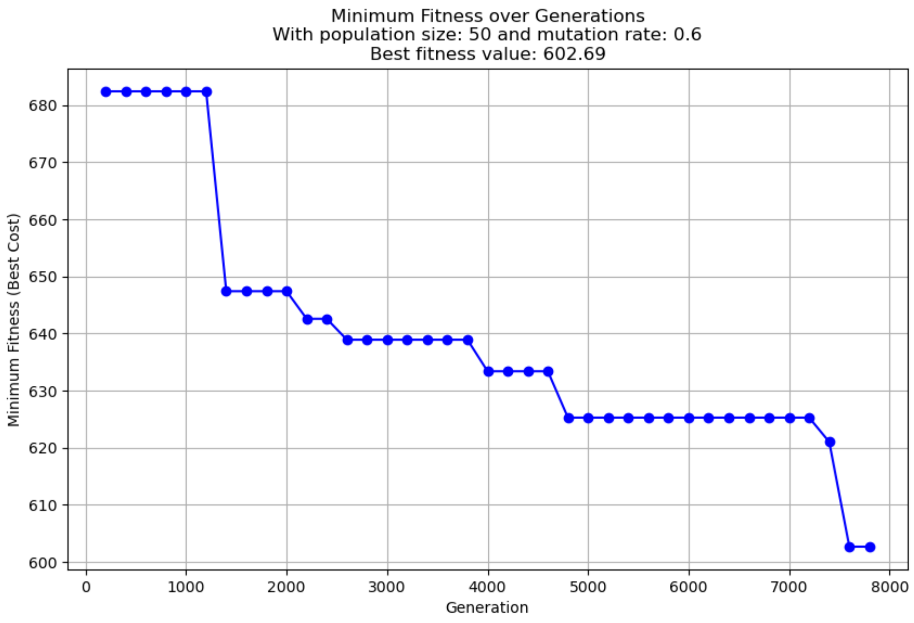
Now with population size of 50:

Population Size: 50
Mutation Rate: 0.9



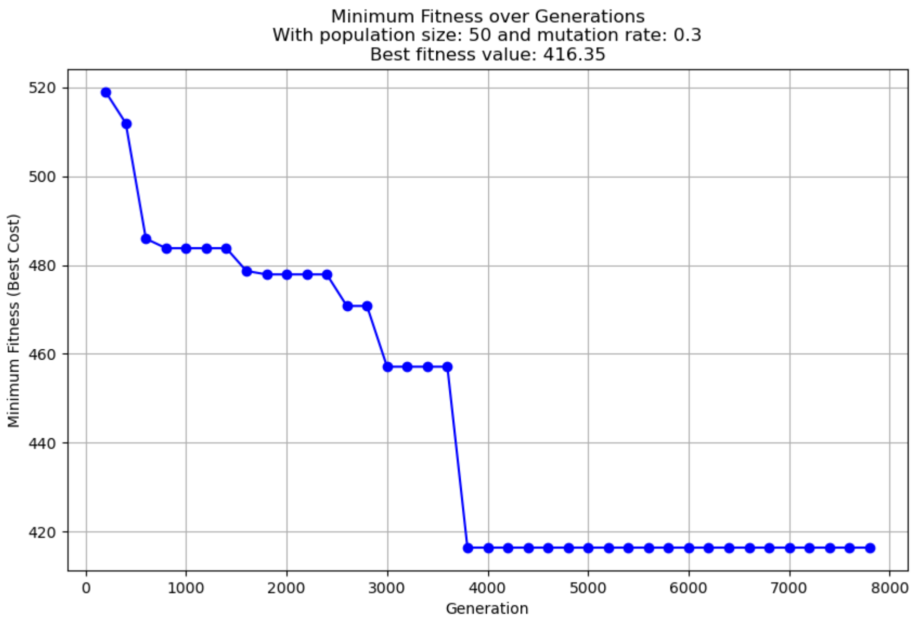
This plot has two main regions, the initial drop and continuous values at arround 640 (fitness value) before generation 4000, and afterwards the new best value is found being 625.6. Initially I thought it would look more like a stair pattern.

Population Size: 50
Mutation Rate: 0.6



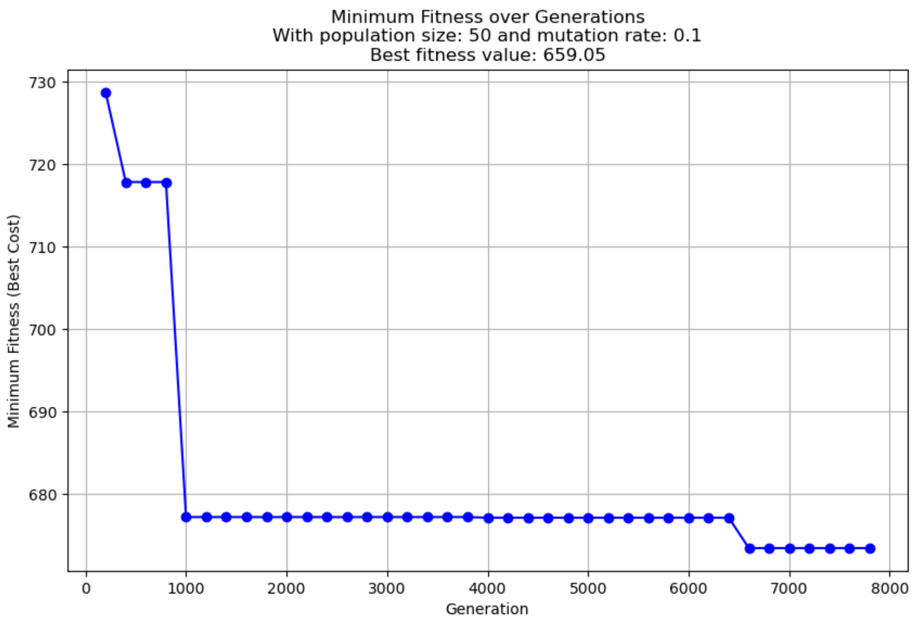
This execution is slowly decreasing towards a better score. Initially it stays a little bit more than 1000 generations in a "high" value, but after the generation 1200 every hundreds of generations the result gets better. Having intervals of 1000 generations of stagnation, but continuing to decrease, until the best fitness is achieved, with a value of 602.69.

Population Size: 50
Mutation Rate: 0.3



At a first glance this plot looks really good, making constant progress in decreasing the value of the cost. And the first time the 500 cost barrier is broken. It has to be said that the initial generations had a very good start, maybe caused by luck. But one thing is sure, after generation 3600 a remarkable decrease in the cost is made, and arrives at a value lower than 420. At the end the amazing minimum best fitness value is 416.35 and surpassing all the previous executions.

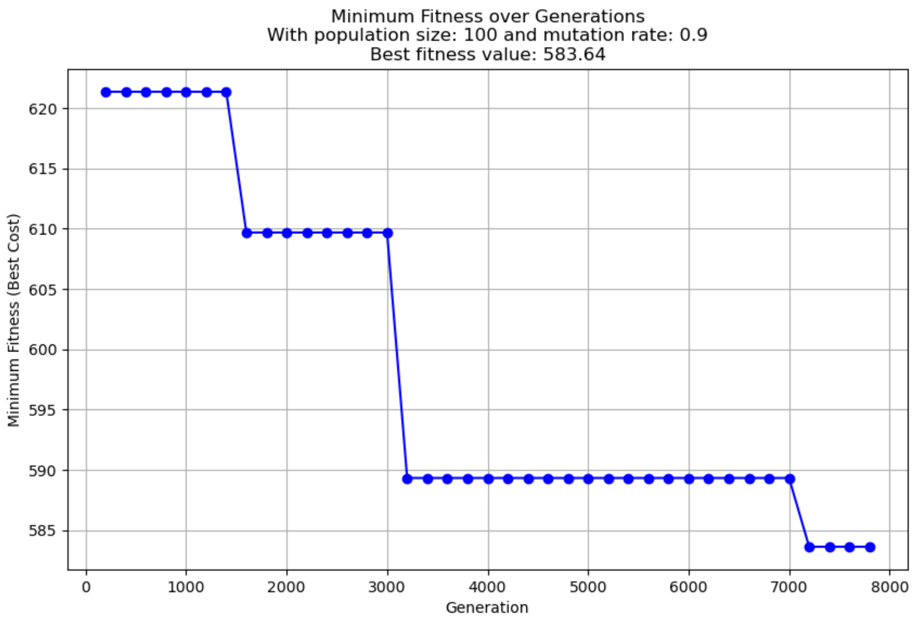
Population Size: 50
Mutation Rate: 0.1



In this part we can see a sharply drop approximately on the generation 800, which then nearly continues with a value of 660 for more than 5000 generations. Although after generation 6400 a lower value is found, ending in the best fitness being 659.05

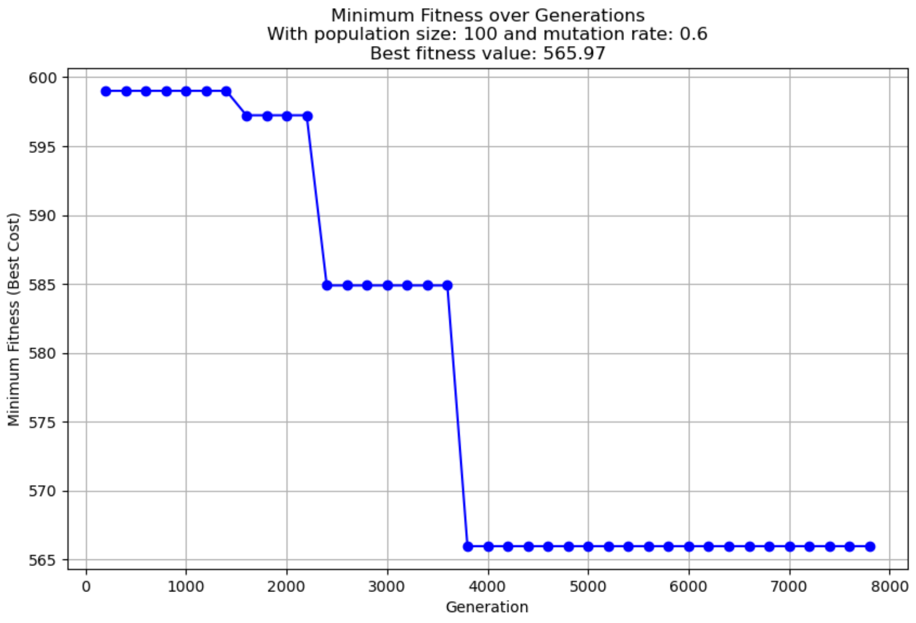
And last but not least the population size is 100. The ones that took more time to execute, and seemed to have better opportunities to find good results.

Population Size: 100
Mutation Rate: 0.9



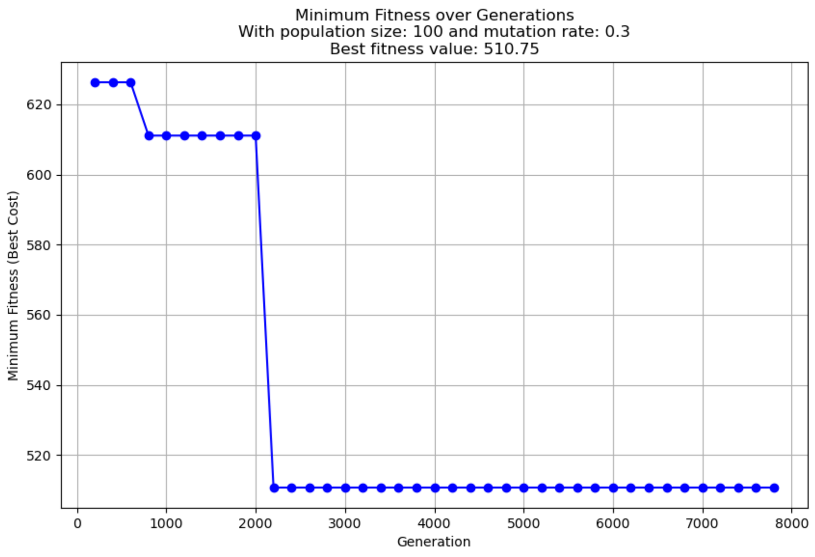
The plot has 3 major drops, the first one at the generation 1500, the second one at the generation 3000 and the last one at 7000. The barrier of fitness 600 is broken in the generation 3000, which is promising because there were still a lot of generations, but at the end it finished at the best value of 583.64

Population Size: 100
Mutation Rate: 0.6



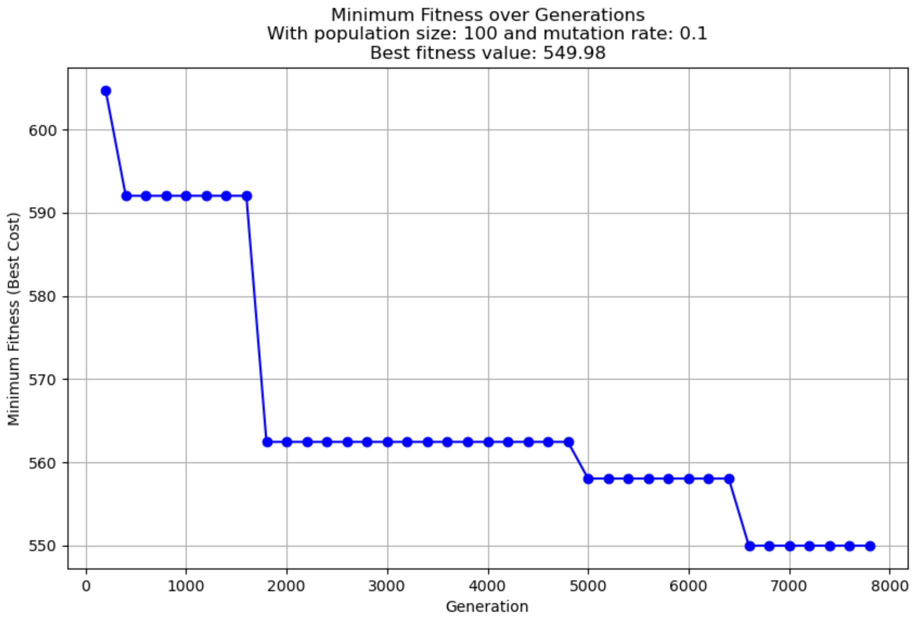
This plot is way better than the previous one with a mutation rate of 0.9. In this case, it starts constant but at a good initial value of approximately 598 and after generation 2100 decreases to 585, and stays there for more than one thousand iterations. And the last drop takes place at generation 3600 which obtains a good value, of 565.97 until the end of execution.

Population Size: 100
Mutation Rate: 0.3



This plot is another promising one, starting at a low value arround 630, which is better than the best fitness some executions have arrived, and does a big jump after iteration 2000 landing at a minimum value that is stagnated since the end of the execution. With a best fitness value of 510.75

Population Size: 100
Mutation Rate: 0.1



The last execution with the smallest mutation rate, decreases in 4 blocks. Making small improvements each time. Having a big stagnation between generation 2000 and 5000. Having a decreasing slope pattern present in many other results, analyzed previously. And after the last drop at generation 6500, the best and final fitness value is 549.98

1.2 Discussion

The executions with the results using the Genetic Algorithm on the TSP problem, revealed a clear pattern, which was pointed out during the analysis (the decreasing staircase) we have that GA consistently improves the solution quality over generations, this means that a shortest path is being found. This success is influenced largely by our change of parameters. Fixing all of them and only changing the population size and the mutation rate, which control the algorithm's exploration of all the solution space.

It has been tested the GA with various population sizes and also mutation rates to understand and see their impact. First smaller populations, that converged quickly, and often got stuck in a local minima. And on the other hand, higher mutation rates helped prevent premature convergence, but this also could lead to not optimal solutions. Regarding the lower mutation rates, they favored stability but limited the exploration. It was shown that the best results for small populations came with lower mutation rates, allowing more focused refinement.

In the case with moderate population sizes, we observed an improvement in the performance compared to the smaller ones. The genetic diversity was higher and allowed a better exploration, but also reflected an earlier convergence, which continued to be an issue at other higher mutation rates. The optimal results were achieved with a balance of exploration and not taking the risk, using moderate mutation rates.

Finally, larger populations have provided the most significant improvements. The diversity, thanks to the high population, prevented an early stagnation, as seen in the results, and the mutation rates had a more pronounced effect. The mutation rate of 0.3 consistently achieved better results, not being a high value like 0.9 or 0.1 with less change of mutations. We have that higher mutation rates led to an initial fast improvement but afterwards failed to maintain its good progress, while on the other hand, the lower rates ended up in slower but better improvements.

Is true that larger populations offer greater potential on exploration, but they require more time to converge, making them more time inefficient. Finding the right balance between the population size and the mutation rate is crucial for obtaining good results, and leveraging the GA algorithm.

1.3 Conclusions

The Genetic Algorithm (GA) has proved based on the results, to be a powerful algorithm for solving the famous problem of the TSP. Based on his nature of the natural selection through evolution makes it a good option. And being able not to use an heuristic, because there is not a good one for this problem, thus the algorithm A* would not be suitable.

The presented results, highlighted the importance of the parameter tuning. The population size and the mutation rate considerably impacted the GA's performance. For instance, larger populations increased the diversity but required more execution time. And the mutation rate controlled the balance between exploration and the diversity. Finding the right combination of values was key for achieving better results.

As explained in the results and discussion, we see a clear correlation between the population size and the mutation rate, on the performance. Where larger populations are the ones that provide greater genetic diversity, and reduce the chances of getting stuck in the beginning.

We found that a combination of a moderate population size and a mutation rate of 0.3 consistently produced the best results in our TSP experiments, showing that it is important to balance the approach we are taking. With the best result being with the population size of 50.

In summary, the Genetic Algorithm successfully found good paths, as seen with the minimum value of the fitness function. Proving to be a powerful algorithm for solving the TSP problem. It has to be clarified, that the effectiveness lies down to the parameter tuning, concretely on the population size and mutation rate. If we consider similar problems or real world applications, it's fundamental that we spend more time on parameter tuning, if we want to achieve optimal or feasible results. The ability of GAs to balance diversity and exploration, makes them well-suited for solving complex problems like this TSP.

2 Task 2 - Fitness function for the optimization problem using GA

The function that has to be implemented is based on the mathematical equation:

$$f(x, y, z) = 2 \cdot x \cdot z \cdot e^{-x} - 2 \cdot y^3 + y^2 - 3 \cdot z^3 + \frac{\cos(x \cdot z)}{1 + e^{-(x+y)}}$$

Implementation of the fitness function:

```python
# Define the fitness function
def fitness_function(x, y, z):
  '''
  computes the value of the equation given inputs x, y, and z
  '''
  # the objective is to maximize

  first_term = 2 * x * z * np.exp(-x)
  second_term = -2 * (y ** 3)
  third_term = y ** 2 -3 * (z ** 3)
  fourth_term = np.cos(x * z) / (1 + np.exp(-(x + y)))

  return first_term + second_term + third_term + fourth_term
```

The fitness function we have to implement receives 3 parameters, which are the 3 variables that the mathematical function uses. And first of all, it's clear that we have to maximize, making the GA obtain better results for the three variables.

To make things easier I divided the function in the 4 terms that take part, and used numpy for the math functions. Notice that e I have used exp and the values of cos, are expected to be in radians.

To do the exponentiation I have used ** and the other parts are just basic mathematical operations.

This fitness function helps the algorithm to improve over time, giving feedback if the parameters are better and improving over generations. Also it is worth saying that the function might have a lot of local minima, making it hard to optimize. At the end of the execution we obtained:

Best solution found: x = 3.4226, y = 1.3450, z = -1.7779

That is a similar result as the first execution that was given in the *Task_2* file.

This task shows the importance of having a good fitness function, and how to help the GA algorithm towards a better solution.

3 Task 3 - GA for solving a Maze

This task has the objective to find a path of a maze, making a mouse reach the destination or food with a shortest and acceptable path. The GA has to find a path by optimizing the problem automatically. Using a 2D matrix representing the maze with the possible passes being 1 and the obstacles 100.

The core fitness function:

```python
def fitness_eval(self):
    pos = START
    visited_positions = set()  # track visited positions to avoid loops
    valid_steps = 0
    penalty = 0  # introduce penalty for invalid moves
    last_direction = None  # Track the last direction

    reverse_moves = {'R': 'L', 'L': 'R', 'U': 'D', 'D': 'U'}
    distance_to_goal = np.abs(START[0] - GOAL[0]) + np.abs(START[1] - GOAL[1])

    for direction in self.chromosome:
        # penalize if the mouse moves in the exact opposite direction of its last move (backtracking)
        if last_direction and reverse_moves[last_direction] == direction:
            penalty += (penalty_value * (1 + (distance_to_goal - valid_steps) / distance_to_goal)) # higher penalty if further from goal
            break

        new_pos = move(pos, direction)
        # check if the new position is a valid move in the maze
        if possible_move(maze, new_pos):
            if new_pos not in visited_positions:
                pos = new_pos
                visited_positions.add(pos)
                valid_steps += 1
                last_direction = direction
            else:
                penalty += (penalty_value * (1 + (distance_to_goal - valid_steps) / distance_to_goal))
                break
        else:
            break  # stop evaluation after invalid move

    # Manhattan distance to goal
    distance_to_goal = np.abs(pos[0] - GOAL[0]) + np.abs(pos[1] - GOAL[1])

    # reward progress made towards the goal (closer to the goal means lower fitness)
    progress_reward = (len(maze) + len(maze[0])) - distance_to_goal

    fitness = distance_to_goal * pond_dist_goal # the lower the better
    fitness -= valid_steps * pond_valid_steps  # reward for valid steps
    fitness -= progress_reward
    fitness += penalty

    return fitness
```

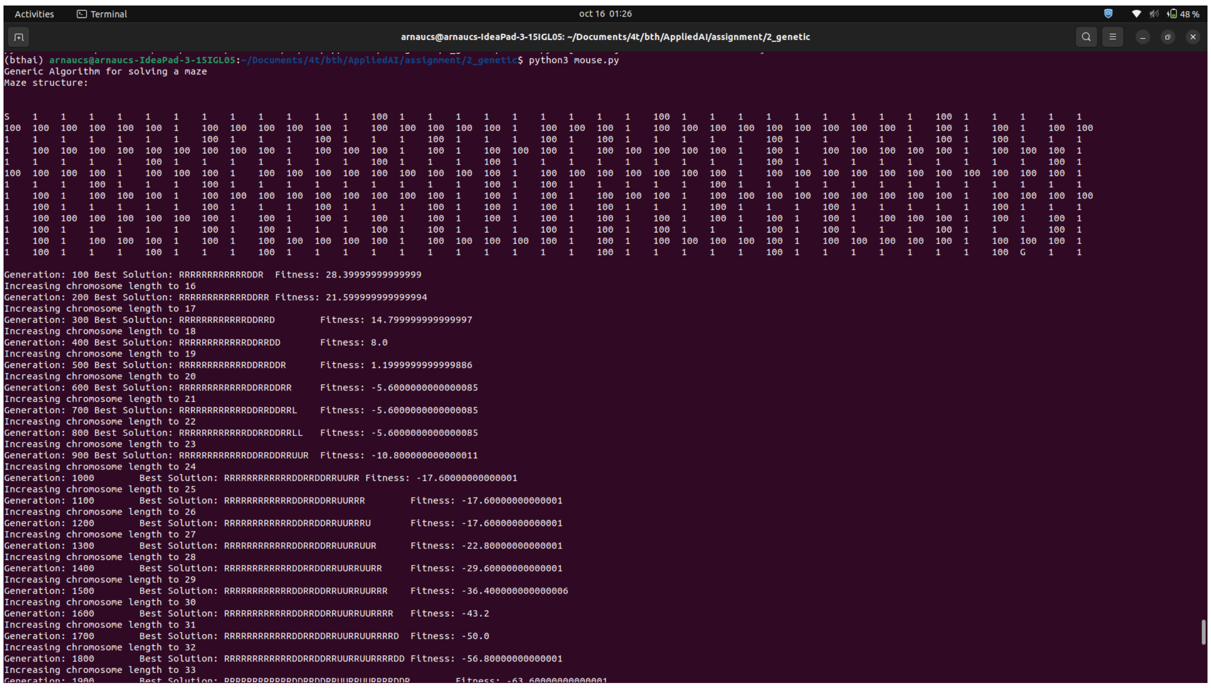
- The `fitness_eval` function is central to evaluating how well each "mouse" or individual performs in navigating the maze. It calculates a fitness score that reflects the distance from the goal, the number of valid steps taken, and any penalties for bad decisions. The goal is to minimize this fitness score, with lower values indicating better paths.
- This is a detailed explanation of each component of the function:
  - `pos = START`: Initializes the starting position of the mouse at the maze's entrance.
  - `visited_positions = set()`: A set is used to track all the positions that the mouse has visited, ensuring that loops (where the mouse revisits the same position) are avoided. Loops would otherwise artificially inflate the fitness by making paths longer than necessary.
  - `valid_steps = 0`: This variable tracks the number of valid steps the mouse takes. A valid step is a move to a new, unvisited position within the maze's boundaries and not blocked by an obstacle.
  - `penalty = 0`: Introduces a penalty for invalid or inefficient moves, such as moving into a wall or revisiting a position, helping guide the algorithm towards more optimal paths.
  - `last_direction = None`: Stores the last direction taken by the mouse. This helps the algorithm detect backtracking, which can be penalized.
  - `reverse_moves = 'R': 'L', 'L': 'R', 'U': 'D', 'D': 'U'`: A dictionary mapping each direction to its reverse, allowing the algorithm to easily detect backtracking.
  - `distance_to_goal = np.abs(START[0] - GOAL[0]) + np.abs(START[1] - GOAL[1])`: Calculates the Manhattan distance from the start to the goal. This distance is used to measure how far the mouse still has to go, with lower distances indicating better performance.
- Chromosome Iteration:
  - The algorithm loops through each direction in the mouse's chromosome, which represents the sequence of moves it will attempt.
- Backtracking Penalty: If the mouse moves in the reverse direction of its last move, a penalty is applied, discouraging backtracking.
- New Position Calculation (`new_pos = move(pos, direction)`): For each direction, the new position is calculated based on the current position and the movement direction.
- `possible_move(maze, new_pos)`: Checks if the move is valid, i.e., within the maze's bounds and not blocked by a wall or obstacle.
- If the move is valid and the position hasn't been visited before, the mouse moves to the new position, and it is added to `visited_positions`. Each valid step increases `valid_steps`.
- Penalties for Invalid Moves: If the mouse revisits a position or moves into an invalid space, a penalty is added to the fitness score to discourage inefficient movement.
- Final Distance to Goal: After iterating through the chromosome, the remaining distance to the goal is recalculated, determining how close the mouse got to the target.
- `progress_reward = (len(maze) + len(maze[0])) - distance_to_goal`: Reflects progress towards the goal. The closer the mouse gets, the higher the reward.

- Fitness Calculation:
  - `fitness = distance_to_goal * pond_dist_goal`: The base fitness score is proportional to the distance remaining to the goal, with smaller distances leading to lower scores.
  - `fitness -= valid_steps * pond_valid_steps`: The fitness score is adjusted by rewarding valid steps taken, with more valid steps resulting in a better score.
  - `fitness -= progress_reward`: Progress towards the goal reduces the fitness score, incentivizing paths that get closer to the goal.
  - Penalties: Any accumulated penalties are added back to the fitness score, with more penalties leading to a worse fitness score.

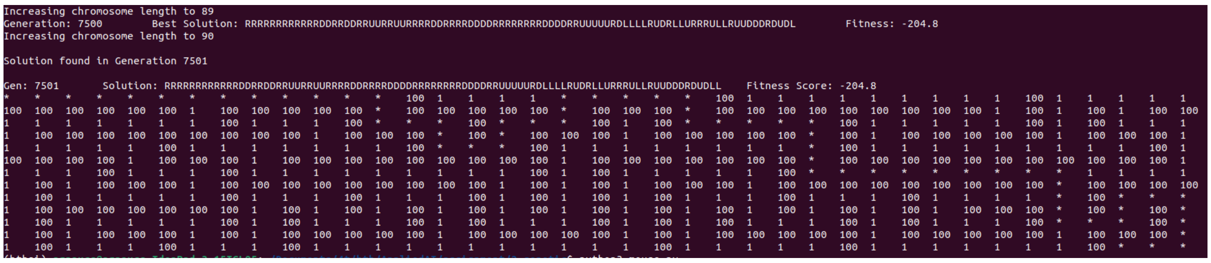
All the scaling factors (starting with pond_) are defined at the beginning of the code, so they can be changed fast, and easy for parameter tuning.

In summary, the `fitness_eval` function evaluates the quality of each path by combining distance to goal, valid steps, and penalties, guiding the genetic algorithm to produce better paths over generations, with lower fitness scores and more efficient movement.

Execution and final result


Activities
Terminal
oct 16 01:26
48 %
arnaucs@arnaucs-IdeaPad-3-15IGL05: ~/Documents/4t/bth/AppliedAI/assignment/2_genetic

```
(bthai) arnaucs@arnaucs-IdeaPad-3-15IGL05:~/Documents/4t/bth/AppliedAI/assignment/2_genetic$ python3 mouse.py
Generic Algorithm for solving a maze
Maze structure:


S    1    1    1    1    1    1    1    1    1    1    1    1    100  1    1    1    1    1    1    1    1    1    100  1    1    1    1    1    1    1    1    1    100  1    1    1    1    1
100  100  100  100  100  100  1    100  100  100  100  100  1    100  100  100  100  100  1    100  100  100  1    100  100  100  100  100  100  100  100  100  1    100  1    100  1    100  100
1    1    1    1    1    1    1    100  1    1    1    100  1    1    1    100  1    1    1    100  1    100  1    1    1    1    1    100  1    1    1    1    1    100  1    100  1    1    1
1    100  100  100  100  100  100  100  100  100  1    100  100  100  1    100  1    100  100  100  1    100  100  100  100  100  1    100  1    100  100  100  100  100  1    100  100  100  1
1    1    1    1    1    100  1    1    1    1    1    1    1    100  1    1    1    100  1    1    1    1    1    1    1    1    1    100  1    1    1    1    1    1    1    1    1    100  1
100  100  100  100  1    100  100  100  1    100  100  100  100  100  100  100  100  100  1    100  100  100  100  100  100  100  1    100  100  100  100  100  100  100  100  100  100  100  1
1    1    1    100  1    1    1    100  1    1    1    1    1    1    1    1    1    100  1    100  1    1    1    1    1    100  1    1    1    1    1    1    1    1    1    1    1    1    1
1    100  1    100  100  100  1    100  100  100  100  100  100  100  100  100  1    100  1    100  1    100  100  100  1    100  100  100  100  100  100  100  100  100  1    100  100  100  100
1    100  1    1    1    1    1    100  1    1    1    100  1    1    1    100  1    100  1    100  1    100  1    1    1    100  1    1    1    100  1    1    1    1    1    100  1    1    1
1    100  100  100  100  100  100  100  1    100  1    100  1    100  1    100  1    100  1    100  1    100  1    100  1    100  1    100  1    100  1    100  100  100  1    100  1    100  1
1    100  1    1    1    1    1    100  1    100  1    1    1    100  1    100  1    1    1    100  1    100  1    100  1    1    1    100  1    100  1    1    1    100  1    1    1    100  1
1    100  1    100  100  100  1    100  1    100  100  100  100  100  1    100  100  100  100  100  1    100  1    100  100  100  100  100  1    100  100  100  100  100  1    100  100  100  1
1    100  1    1    1    100  1    1    1    100  1    1    1    1    1    1    1    1    1    1    1    100  1    1    1    1    1    100  1    1    1    1    1    1    1    100  G    1    1

Generation: 100 Best Solution: RRRRRRRRRRRRDDR  Fitness: 28.39999999999999
Increasing chromosome length to 16
Generation: 200 Best Solution: RRRRRRRRRRRRDDRR Fitness: 21.599999999999994
Increasing chromosome length to 17
Generation: 300 Best Solution: RRRRRRRRRRRRDDRRD        Fitness: 14.799999999999997
Increasing chromosome length to 18
Generation: 400 Best Solution: RRRRRRRRRRRRDDRRDD       Fitness: 8.0
Increasing chromosome length to 19
Generation: 500 Best Solution: RRRRRRRRRRRRDDRRDDR      Fitness: 1.1999999999999886
Increasing chromosome length to 20
Generation: 600 Best Solution: RRRRRRRRRRRRDDRRDDRR     Fitness: -5.6000000000000085
Increasing chromosome length to 21
Generation: 700 Best Solution: RRRRRRRRRRRRDDRRDDRRL    Fitness: -5.6000000000000085
Increasing chromosome length to 22
Generation: 800 Best Solution: RRRRRRRRRRRRDDRRDDRRLL   Fitness: -5.6000000000000085
Increasing chromosome length to 23
Generation: 900 Best Solution: RRRRRRRRRRRRDDRRDDRRUUR  Fitness: -10.800000000000011
Increasing chromosome length to 24
Generation: 1000        Best Solution: RRRRRRRRRRRRDDRRDDRRUURR Fitness: -17.60000000000001
Increasing chromosome length to 25
Generation: 1100        Best Solution: RRRRRRRRRRRRDDRRDDRRUURRR        Fitness: -17.60000000000001
Increasing chromosome length to 26
Generation: 1200        Best Solution: RRRRRRRRRRRRDDRRDDRRUURRRU       Fitness: -17.60000000000001
Increasing chromosome length to 27
Generation: 1300        Best Solution: RRRRRRRRRRRRDDRRDDRRUURRUUR      Fitness: -22.80000000000001
Increasing chromosome length to 28
Generation: 1400        Best Solution: RRRRRRRRRRRRDDRRDDRRUURRUURR     Fitness: -29.60000000000001
Increasing chromosome length to 29
Generation: 1500        Best Solution: RRRRRRRRRRRRDDRRDDRRUURRUURRR    Fitness: -36.400000000000006
Increasing chromosome length to 30
Generation: 1600        Best Solution: RRRRRRRRRRRRDDRRDDRRUURRUURRRR   Fitness: -43.2
Increasing chromosome length to 31
Generation: 1700        Best Solution: RRRRRRRRRRRRDDRRDDRRUURRUURRRRD  Fitness: -50.0
Increasing chromosome length to 32
Generation: 1800        Best Solution: RRRRRRRRRRRRDDRRDDRRUURRUURRRRDD Fitness: -56.80000000000001
Increasing chromosome length to 33
Generation: 1900        Best Solution: RRRRRRRRRRRRDDRRDDRRUURRUURRRRDDR        Fitness: -63.60000000000001
```



```
Increasing chromosome length to 89
Generation: 7500        Best Solution: RRRRRRRRRRRRDDRRDDRRUURRUURRRRDDRRRRDDDDRRRRRRRRDDDDRRUUUUURDLLLLRUDRLLURRRULLRUUDDDRDUDL        Fitness: -204.8
Increasing chromosome length to 90

Solution found in Generation 7501

Gen: 7501       Solution: RRRRRRRRRRRRDDRRDDRRUURRUURRRRDDRRRRDDDDRRRRRRRRDDDDRRUUUUURDLLLLRUDRLLURRRULLRUUDDDRDUDLL    Fitness Score: -204.8
*    *    *    *    *    *    *    *    *    *    *    *    *    100  1    1    1    1    *    *    *    *    *    100  1    1    1    1    1    1    1    1    1    100  1    1    1    1    1
100  100  100  100  100  100  1    100  100  100  100  100  *    100  100  100  100  100  *    100  100  100  *    100  100  100  100  100  100  100  100  100  1    100  1    100  1    100  100
1    1    1    1    1    1    1    100  1    1    1    100  *    *    *    100  *    *    *    100  1    100  *    *    *    *    *    100  1    1    1    1    1    100  1    100  1    1    1
1    100  100  100  100  100  100  100  100  100  1    100  100  100  *    100  *    100  100  100  1    100  100  100  100  100  *    100  1    100  100  100  100  100  1    100  100  100  1
1    1    1    1    1    100  1    1    1    1    1    1    1    100  *    *    *    100  1    1    1    1    1    1    1    1    *    100  1    1    1    1    1    1    1    1    1    100  1
100  100  100  100  1    100  100  100  1    100  100  100  100  100  100  100  100  100  1    100  100  100  100  100  100  100  *    100  100  100  100  100  100  100  100  100  100  100  1
1    1    1    100  1    1    1    100  1    1    1    1    1    1    1    1    1    100  1    100  1    1    1    1    1    100  *    *    *    *    *    *    *    *    *    1    1    1    1
1    100  1    100  100  100  1    100  100  100  100  100  100  100  100  100  1    100  1    100  1    100  100  100  1    100  100  100  100  100  100  100  100  100  *    100  100  100  100
1    100  1    1    1    1    1    100  1    1    1    100  1    1    1    100  1    100  1    100  1    100  1    1    1    100  1    1    1    100  1    1    1    1    *    100  *    *    *
1    100  100  100  100  100  100  100  1    100  1    100  1    100  1    100  1    100  1    100  1    100  1    100  1    100  1    100  1    100  1    100  100  100  *    100  *    100  *
1    100  1    1    1    1    1    100  1    100  1    1    1    100  1    100  1    1    1    100  1    100  1    100  1    1    1    100  1    100  1    1    1    100  *    *    *    100  *
1    100  1    100  100  100  1    100  1    100  100  100  100  100  1    100  100  100  100  100  1    100  1    100  100  100  100  100  1    100  100  100  100  100  1    100  100  100  *
1    100  1    1    1    100  1    1    1    100  1    1    1    1    1    1    1    1    1    1    1    100  1    1    1    1    1    100  1    1    1    1    1    1    1    100  *    *    *
(bthai) arnaucs@arnaucs-IdeaPad-3-15IGL05:~/Documents/4t/bth/AppliedAI/assignment/2_genetic$ python3 mouse.py
```

It successfully found the goal in 90 steps (the same as chromosomes). Also, if the code is executed multiple times, you can get better results like 75 steps, or worst cases with more than 110. Or rarely not a solution in less than 200 steps, which I control stopping the execution if that happens.

Regarding the GA algorithm, it effectively solved the maze problem. The code performed as expected, correctly penalizing poor decisions like reversing direction, and making sure that the valid moves were rewarded. A key success was the dynamic adjustment of mutation rates based on fitness stagnation and proximity to the goal, so it was not stuck on a local minima.

Overcoming the final part of the maze, which is the final closed section, is challenging to arrive there, because the closest point to the goal, is outside, the walls, and for entering that final section, you have to move away and increase the fitness function. This part is where I had more complications on this task. Moreover, I tried with fixed chromosome length, or also being able to decreased, but the results didn't find the goal.

Also, this task was more challenging than the others, due to the freedom of implementation, and because of the long waitings after starting to run the code. Parameter tuning was hard, because sometimes it was a subtle change but wasn't visible in a couple minutes.

In conclusion, there was a balance in exploration and exploitation, adapting parameters based on the performance of previous generations. And successfully evolving a generation of lost mice who only wanted to arrive at the goal.

4 Task 4 - Regression for Data

This task requires generating data points using numpy. Then use a regression model to predict the y data. And progressively changing the prediction models, such as a polynomial model in b) section, a three-layer neural network in c).

After the three models, do a comparison using the mean squared error (MSE), and finally plot the data points and curve of predictions of the a) and b) models.

a) Linear Regression model to predict data using sklearn

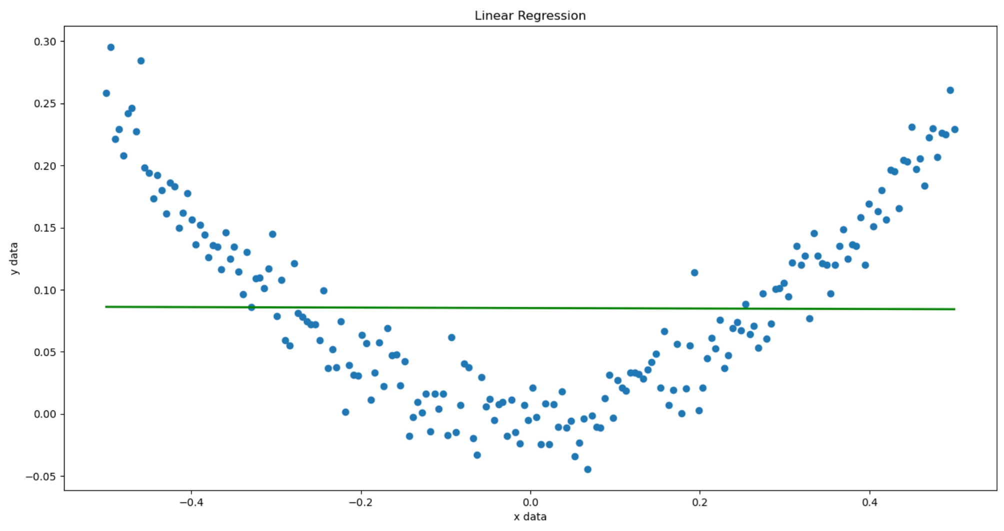
The main part of the code is this:

```
reg = LinearRegression()
reg.fit(x_data, y_data)
print("The linear model is: Y = {:.5} + {:.5}x".format(reg.intercept_[0],
    reg.coef_[0][0]))

predictions = reg.predict(x_data)
# and plot the line
```

Using sklearn is comfortable and fast, after having the data, we just have to fit the line, and we get the equation of the line.

Afterwards, we can predict the values, and thus plot the linear regression line on top of the values.

The result as expected is a bad fitting, because the data seems quadratic and the line is linear.



b) Polynomial model with highest power of 2 to predict y data

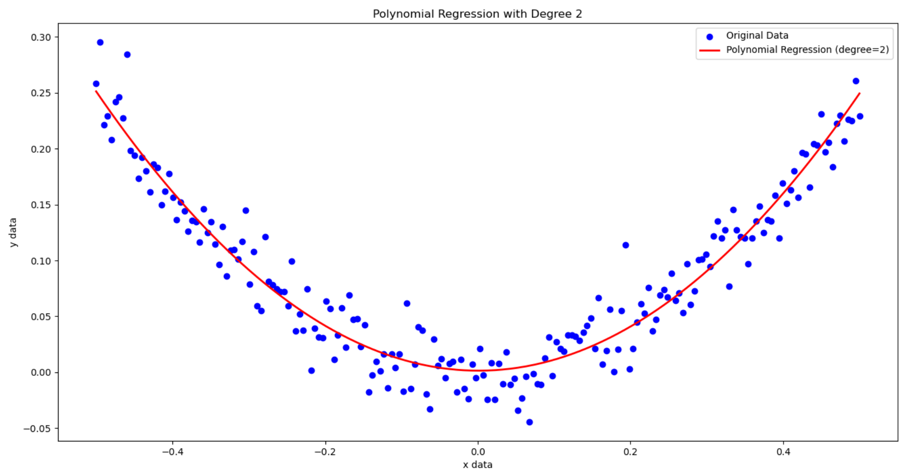
```python
from sklearn.preprocessing import PolynomialFeatures

# polynomial transformation (degree 2)
poly_features = PolynomialFeatures(degree=2)
x_poly = poly_features.fit_transform(x_data)

# fit the Linear Regression model on the polynomial data
poly_reg_model = LinearRegression()
poly_reg_model.fit(x_poly, y_data)

# make predictions using the polynomial model
y_poly_pred = poly_reg_model.predict(x_poly)

# plot ...
```

In this main part of the code, I implemented the polynomial of degree 2, using the `PolynomialFeatures` from sklean preprocessing library. The important part is realizing that now we are using both a linear term and a quadratic term, so it will be useful to fit the curve based on the nature of the data.

First we had to transform the data on the x variable, transforming it to a polynomial with terms up to degree 2, the fit_transform creates a new feature matrix, which contains three columns. The first one is the intercept values, the second the original x, and the third is x^2.

After that, we apply the `LinearRegression()` like in the previous section, but now the input data includes higher-order terms so it's non-linear.

Finally we do the prediction, to compute the predicted y values, and everything is ready to plot.



Now the result, correctly predicts and has a curve shape. Making it much more accurate and better option than linear regression, without adding a lot of complexity.

c) Three-layer Neural Network (using Tensorflow)

```python
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Build the neural network model
model = Sequential()

# Input layer and hidden layer (6 neurons)
model.add(Dense(units=6, activation='relu', input_shape=(1,)))

# Output layer (1 neuron)
model.add(Dense(units=1))

# Compile the model (Mean Squared Error loss for regression, Adam optimizer
    )
model.compile(optimizer='adam', loss='mean_squared_error')

# Train the model
history = model.fit(x_train, y_train, epochs=100, batch_size=10,
    validation_split=0.2)

# Evaluate the model on test data
test_loss = model.evaluate(x_test, y_test)
print(f"Test Loss (MSE): {test_loss}")

# Make predictions
predictions = model.predict(x_data)

# plot ...
```

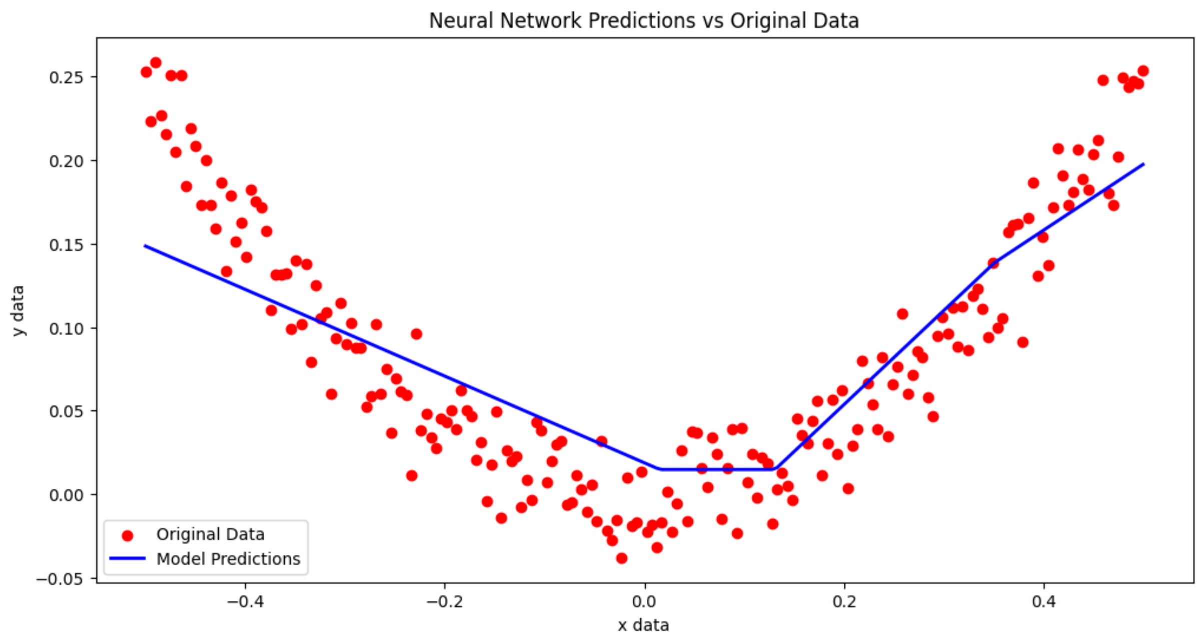
First of all the neural network consists of an input layer, a hidden layer with 6 nodes and an output layer. And the training data will be used to fit the NN while the testing data will be used to evaluate how well the model generalizes the unseen data.

The model architecture is created using the Sequential class, which allows to stack layers sequentially. So the layers are added one after another in sequence, we add them using the .add(). We include a fully connected layer to the NN wich serves both as input layer and hidden layer. The input has one dimension, which is the value of x, indicated in the `input_shape=(1,)`. The input layer simply passes the input data to the next layer, then the hidden layer with 6 neurons. About the activation function, I decided that ReLU would work fine, and it helps the hidden layer having non-linearity, so it can learn complex patterns.

Additionally, the output layer contains a single neuron, which outputs the predicted y value. Then the model is compiled with the Adam optimizer, which is for gradient optimization, and adapting the learning rate during training, and the loss function of Mean Squared Error.

The model is trained on the training data using the fit method, and after completion is evaluated on the test set using evaluate, which calculates the MSE on the unseen test data. And finally train the model using predict to generate predictions for the entire dataset x_data. Including the plot of the results at the end.

This was the result:



At first glance, the fitting looks okay. But there are sections of straight line, like it was trying to overfit, and comparing with the polynomial, it's not that accurate, but we will see which is better in the next section.

d) Comparison mean squared errors of three models

```
# Linear Regression:
mse = mean_squared_error(y_data, predictions)
print(f"Mean Squared Error (MSE) Linear Regression: {mse}")


  # Result:
  # Mean Squared Error (MSE) Linear Regression: 0.006122


# Polynomial degree 2
mse_poly = mean_squared_error(y_data, y_poly_pred)
print(f"Mean Squared Error (MSE) for Polynomial Regression: {mse_poly}")

  # Result:
  # Mean Squared Error (MSE) for Polynomial Regression: 0.000406

# Neural Network
test_loss = model.evaluate(x_test, y_test)
print(f"Test Loss (MSE): {test_loss}")

  # Result:
  # Test Loss (MSE): 0.001521
```

Table with the summary of the values:

Model	**Mean Squared Error (MSE)**
Linear Regression	0.006122
Polynomial Regression (degree 2)	0.000406
Neural Network	0.001521

Table 1: Mean Squared Error (MSE) for Different Models

First let's talk about the linear Regression: it has the highest MSE making it the worst, which is expected since the data follows a quadratic (non-linear) pattern, and if the model it's linear can't model these relationships effectively. The model, underfits the data.

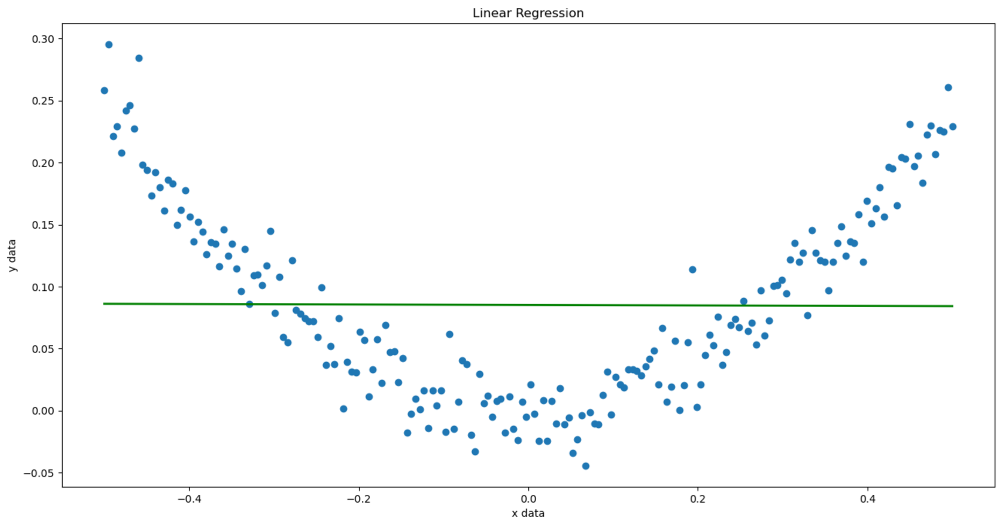
The second worst model is the Neural Network in terms of MSE. But it's important to consider that they are highly flexible and can model non-linear patterns. Adding the complexity in terms of parameter tuning. The results could not be that good because of the limited complexity of only 1 hidden layer with 6 neurons, or in general a small NN.

The clear winner is the Polynomial Regression, it achieves the lowest MSE of all three models. It's specifically designed to model quadratic relationships, and the data has this pattern. This model fits the data successfully.

For more complex real-world problems, or if we don't know the nature of the data, I would use Neural Networks, after seeing these results. But it's worth knowing that simpler models like linear or quadratic, can do a good job with less complexity.

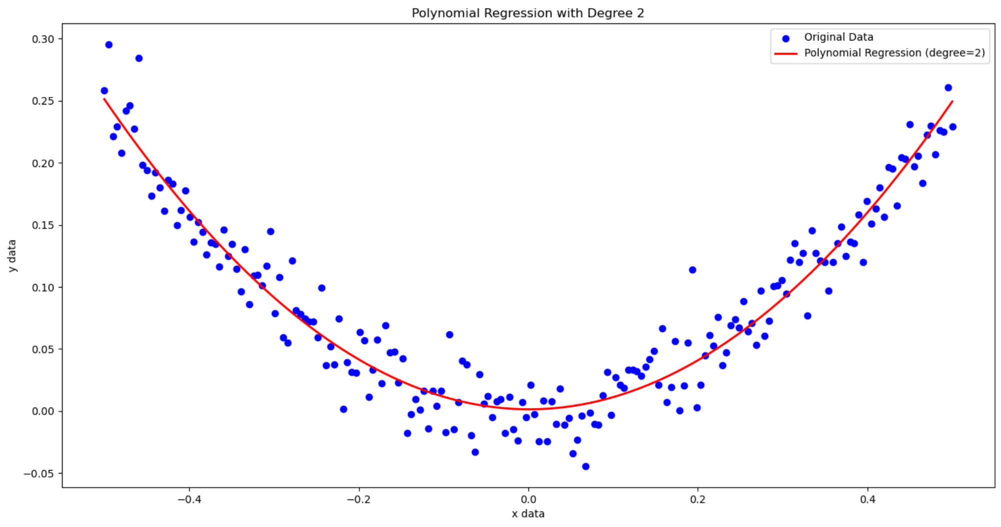
e) Plotting the results of the first two models

The result of the Linear Regression model is this:



The resulting fitted line is what was expected, linear, but because of the curved pattern of the data, it does very poorly in predicting.

The result of the Polynomial model with power of 2 is:



This model effectively captures the curvature of the data and provides a clear improvement over a simple linear regression model for this type of problem. This shows that knowing the nature of the data, following a quadratic pattern, we can fit it with polynomials up to 2.