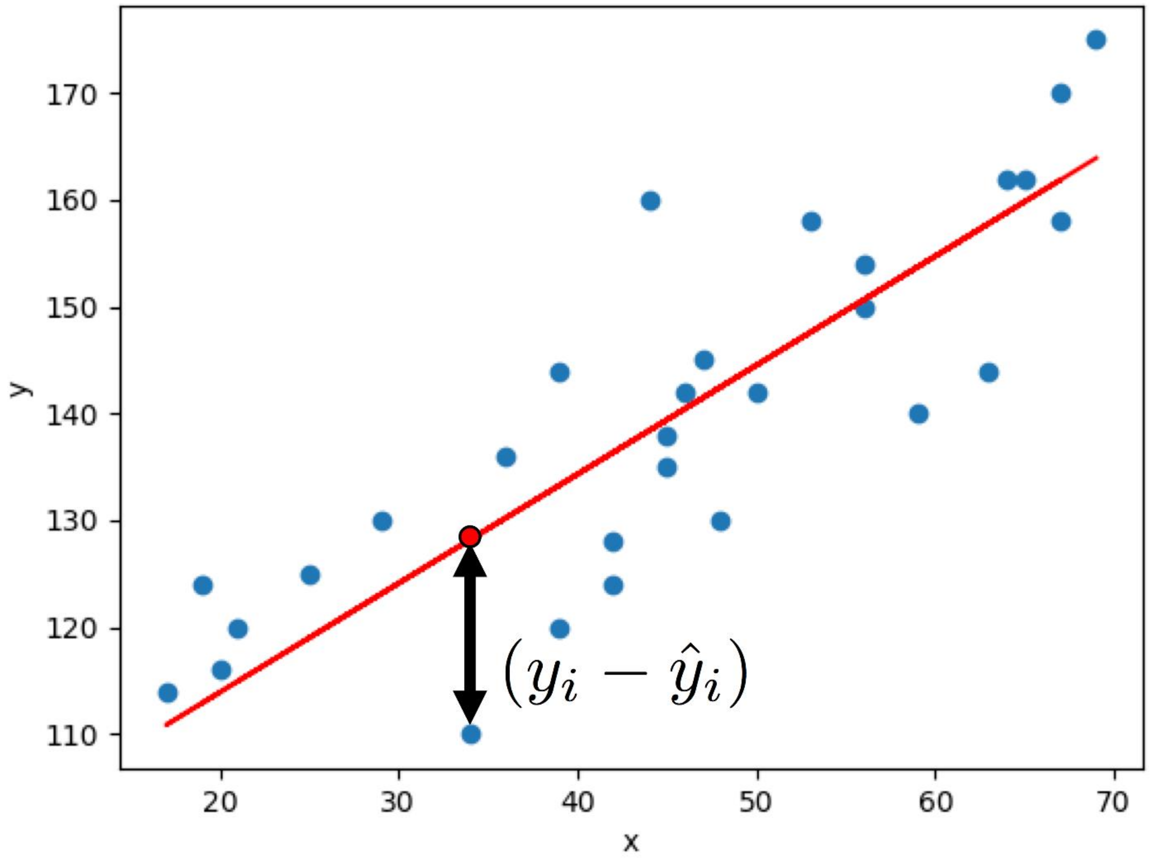
Computation - Final Project

Genetic algorithm



Jaime Resano Aísa

Spring 2023

Index

1. Introduction

1.1. Problem description

There is a training dataset available in CSV format, where each row represents an observation and the columns correspond to numerical input values, while the last column represents the outcomes. The input data consists of real numbers that have been normalized to facilitate their use.

1.2. Genetical algorithms

In this project, a genetic algorithm will be implemented, which consists of the following main steps: creation of the initial population, calculation of individuals' fitness, creation of offspring from the initial parents, mutation of the offspring, and random selection of individuals for the next generation.

Firstly, an initial population is generated, consisting of a set of random individuals. Each individual represents a possible solution to the problem being addressed.

Next, the fitness of each individual in the initial population is evaluated. Fitness is a measure of the quality of each solution in terms of how well it fits the established optimization criteria. This calculation is performed using an evaluation function that assigns a numerical value to each individual based on their performance.

Afterwards, initial parents are selected for children creation. Parents are selected in pairs based on their fitness, meaning that individuals with better fitness are more likely to be chosen for reproduction. Then a crossover function generates a pair of individuals from them.

Then, mutation is applied to the generated children. Mutation is a process that introduces random changes in individuals in order to explore new regions of the search space and prevent premature convergence towards a suboptimal solution. Mutation alters the characteristics of individuals, allowing for greater genetic diversity in the population.

Finally, individuals for the next generation are randomly selected. This selection is done using a random sampling process, where each individual has a probability of being selected proportional to its fitness. This process ensures that fitter individuals have a higher probability of being chosen for the next generation, but also allows the inclusion of less fit individuals to maintain genetic diversity.

This process is repeated certain number of times or until a desired result is achieved.

1.3. Goals

The purpose of this project is to apply a genetic algorithm to the training data to analyze the relationship between the input variables and the desired outcome.

- Implement a genetic algorithm that follows the steps of creating the initial population, calculating fitness, creating offspring from initial parents, mutating the offspring, and selecting individuals for the next generation.
- Develop an application with a graphical user interface that allows for the automatic execution of experiments. The graphical interface should be intuitive, configurable, and user-friendly, so that users can adjust algorithm parameters and define the problem to be solved.
- Design the application in a way that it can display and store an experiment log. The log should include relevant information about the conducted experiments.

2. Genetic algorithm

2.1. Implementation

All the code has been written in Python. The Numpy library has been used for all the operations of the genetic algorithm. The genetic algorithm has been implemented as a function called “run_experiment” that receives all the input parameters required for the algorithm and returns a “result” object that contains the results of the experiment.

The input datasets are stored as files in CSV (comma separated values) format. They are located in the “dataset” folder.

This is a list of all the variables that determine a result:

- Number pseudo random generator seed: an integer
- Input dataset: a csv file
- Individual count: a positive integer
- Iterations: a positive integer
- Crossover probability: a real from 0 to 1
- Mutation probability: a real from 0 to 1
- Progenitor selection method: a function
- Crossover method: a function
- Mutation method: a function

2.2. Functions

The genetic functions described in this section are stored in the “genetic_functions.py” file.

The individuals have been modeled as an array of real numbers representing the weights of the linear regression model. The initial population is initialized as a matrix array with random values.

The fitness function takes all the input data and the population array and returns an array of real numbers which represent the fitness of each individual. The implementation chosen is the square error since it is simple and well-known.

The progenitor selection functions are stored in the “ProgenitorSelectionFunction” class. They are a “NamedFunction“ instance which is basically a custom class that acts like a function that has a name. There are three progenitor selection functions implemented:

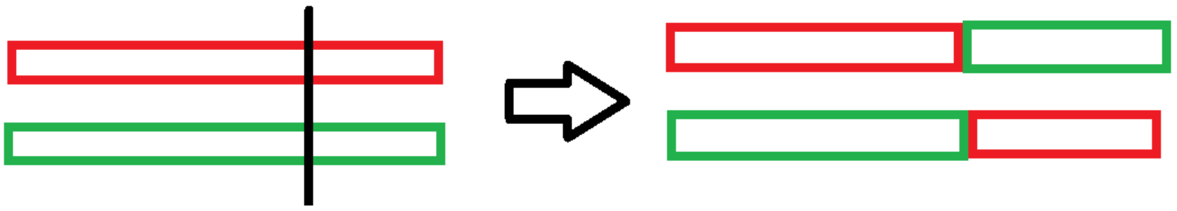
- Roulete: Progenitors are chosen with a weighted random selection based on its fitness score.
- Tournament with replacement: The best progenitors of a subgroup are chosen.
- Tournament without replacement: Same as the previous function, but individuals that have already been chosen cannot be chosen again.

The mutation functions are implemented in the same way. There are two mutation functions implemented:

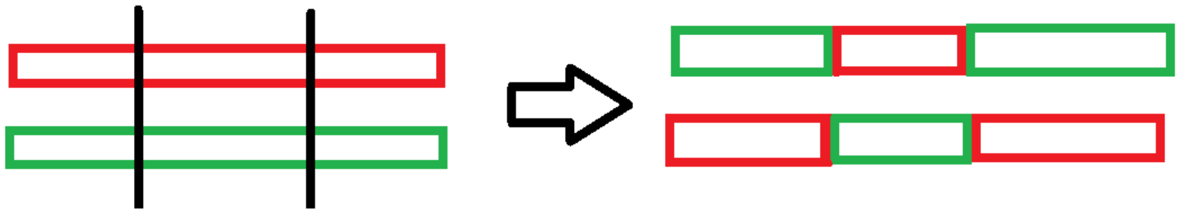
- Uniform: The mutated values are replaced by random values.
- Non-uniform: The individuals are taken in pairs and mutated by mixing their values. The mixing function is the arithmetic average but could be improved by taking into account the individual fitness.

The crossover methods are the following ones:

- Single point: The pair of individuals are mixed by “cutting” them in a single point.



- Two points: There are two “cutting” points.



3. GUI program

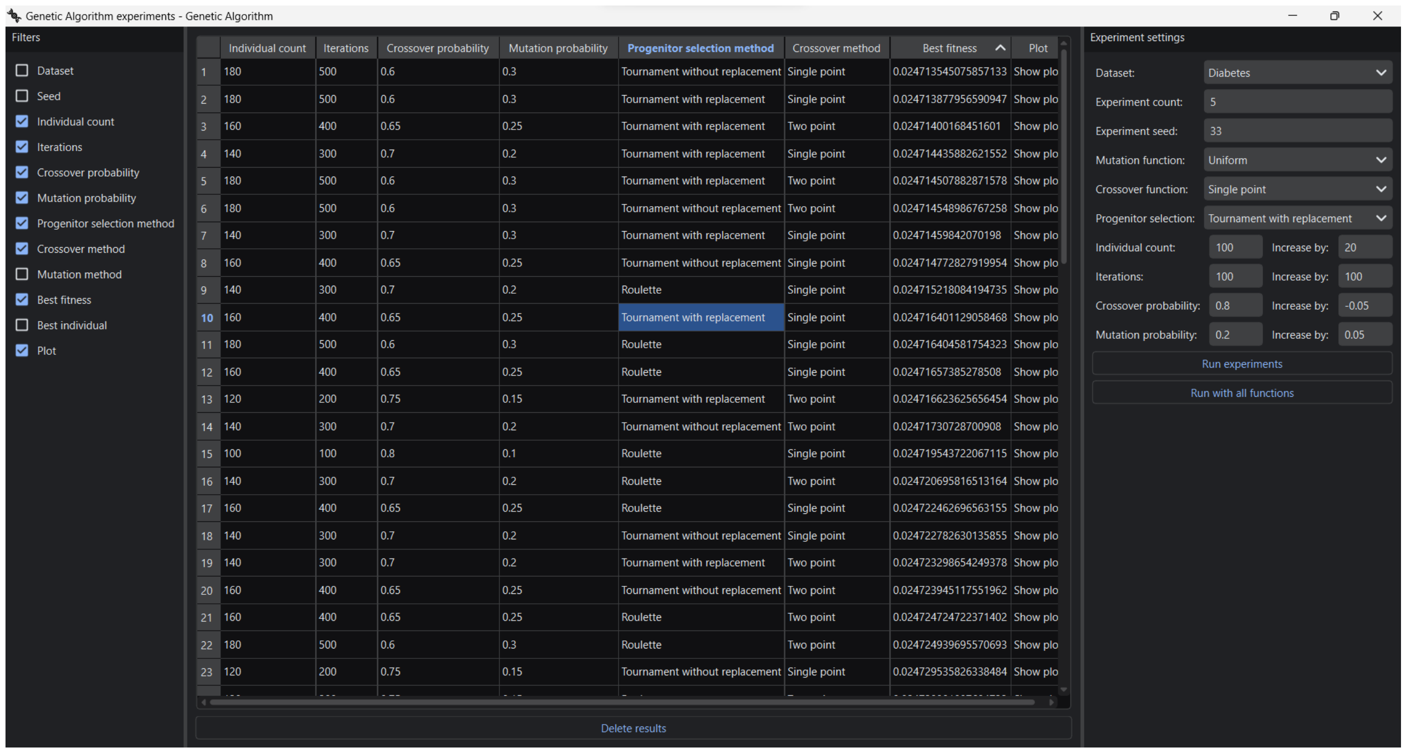
3.1. How to use

The source code of the program can be found in the GitHub repository. In order to run it, the library PySide6 and other dependencies must have been installed and be available in the current Python interpreter. Installing all dependencies can easily be carried out by executing the following command:

```
pip install -r requirements.txt
```

Note that the command needs to be run in a terminal that has opened the project root folder, there it is the “requirements.txt” file that lists all dependencies.

This is a screenshot of the program home page:



Genetic Algorithm experiments - Genetic Algorithm

Filters

- ☐ Dataset
- ☐ Seed
- ✓ Individual count
- ✓ Iterations
- ✓ Crossover probability
- ✓ Mutation probability
- ✓ Progenitor selection method
- ✓ Crossover method
- ☐ Mutation method
- ✓ Best fitness
- ☐ Best individual
- ✓ Plot

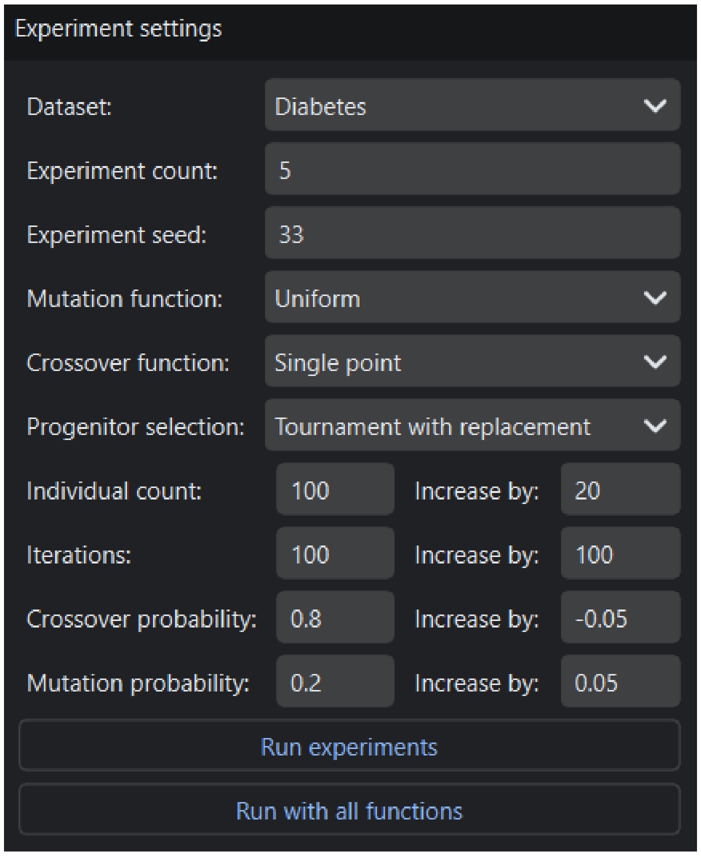
	Individual count	Iterations	Crossover probability	Mutation probability	Progenitor selection method	Crossover method	Best fitness	Plot
1	180	500	0.6	0.3	Tournament without replacement	Single point	0.024713545075857133	Show plo
2	180	500	0.6	0.3	Tournament with replacement	Single point	0.024713877956590947	Show plo
3	160	400	0.65	0.25	Tournament with replacement	Two point	0.02471400168451601	Show plo
4	140	300	0.7	0.2	Tournament with replacement	Single point	0.024714435882621552	Show plo
5	180	500	0.6	0.3	Tournament with replacement	Two point	0.024714507882871578	Show plo
6	180	500	0.6	0.3	Tournament without replacement	Two point	0.024714548986767258	Show plo
7	140	300	0.7	0.3	Tournament with replacement	Single point	0.02471459842070198	Show plo
8	160	400	0.65	0.25	Tournament without replacement	Single point	0.024714772827919954	Show plo
9	140	300	0.7	0.2	Roulette	Single point	0.024715218084194735	Show plo
10	160	400	0.65	0.25	Tournament with replacement	Single point	0.024716401129058468	Show plo
11	180	500	0.6	0.3	Roulette	Single point	0.024716404581754323	Show plo
12	160	400	0.65	0.25	Roulette	Single point	0.02471657385278508	Show plo
13	120	200	0.75	0.15	Tournament with replacement	Two point	0.024716623625656454	Show plo
14	140	300	0.7	0.2	Tournament without replacement	Two point	0.02471730728700908	Show plo
15	100	100	0.8	0.1	Roulette	Single point	0.024719543722067115	Show plo
16	140	300	0.7	0.2	Roulette	Two point	0.024720695816513164	Show plo
17	160	400	0.65	0.25	Roulette	Single point	0.024722462696563155	Show plo
18	140	300	0.7	0.2	Tournament without replacement	Single point	0.024722782630135855	Show plo
19	140	300	0.7	0.2	Tournament with replacement	Two point	0.024723298654249378	Show plo
20	160	400	0.65	0.25	Tournament without replacement	Two point	0.024723945117551962	Show plo
21	160	400	0.65	0.25	Roulette	Two point	0.024724724722371402	Show plo
22	180	500	0.6	0.3	Roulette	Two point	0.024724939695570693	Show plo
23	120	200	0.75	0.15	Tournament without replacement	Single point	0.024729535826338484	Show plo

Experiment settings

Dataset:	Diabetes		
Experiment count:	5		
Experiment seed:	33		
Mutation function:	Uniform		
Crossover function:	Single point		
Progenitor selection:	Tournament with replacement		
Individual count:	100	Increase by:	20
Iterations:	100	Increase by:	100
Crossover probability:	0.8	Increase by:	-0.05
Mutation probability:	0.2	Increase by:	0.05

There is a central widget which is a table that has all the results of the experiments. It is sortable by clicking on the header of the column that needs to be sorted. There are two dock widgets that are placed on the window sides and can be moved and resized by the user. There is a list of checkboxes that toggle the visibility of the table columns. This has been implemented due to the number of columns required.

On the other dock, there are the experiment running options. The user can customize all of the experiment parameters. There has been implemented two modes of running experiments: the single experiment that runs the algorithm certain amount of times and the “combination mode”.


Experiment settings

Dataset:	Diabetes		
Experiment count:	5		
Experiment seed:	33		
Mutation function:	Uniform		
Crossover function:	Single point		
Progenitor selection:	Tournament with replacement		
Individual count:	100	Increase by:	20
Iterations:	100	Increase by:	100
Crossover probability:	0.8	Increase by:	-0.05
Mutation probability:	0.2	Increase by:	0.05

The first mode runs the algorithm as many times as the “experiment count” input says. There are some parameters that are customizable and vary on each experiment. These parameters have an increase field that is applied on each experiment. The increase can also be negative. This is useful for testing how does the results change depending on different parameters.

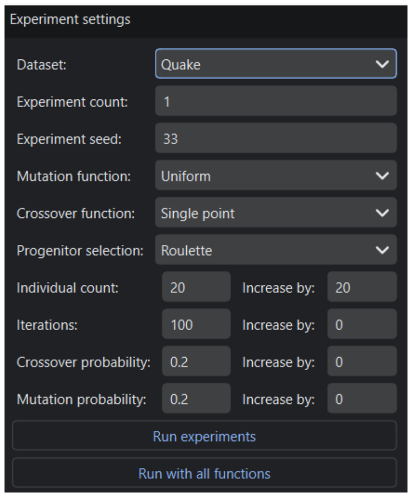
The “combination mode” runs the experiments with the initial configuration but the functions are changed on each round. All of the possible combinations between the functions are tested. Since there are 3 progenitor selection functions, 2 mutation functions and 2 crossover functions, there are 3 * 2 * 2 = 12 possibilities. The experiments are also run as many times as the “experiment count” specifies. This feature shows how the results change depending on the functions chosen.

There is a plot for each result that shows the average fitness of each generation and the best fitness of it.

4. Conclusions

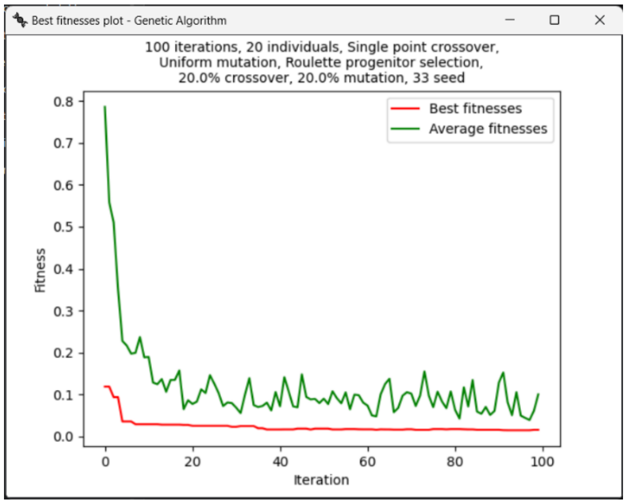
It is extremely hard to extract blunt conclusions about which parameters are better or worse than others since there are a lot of variables that affect the result of an experiment.

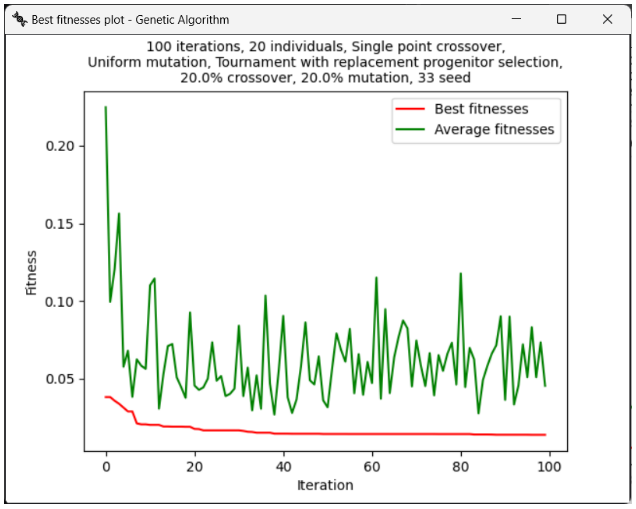
For example, let's try to find out which is the best progenitor selection method. This are the experiment settings:

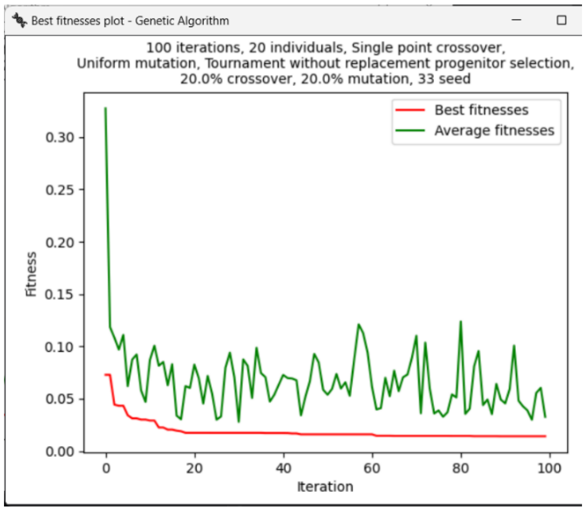


Experiment settings

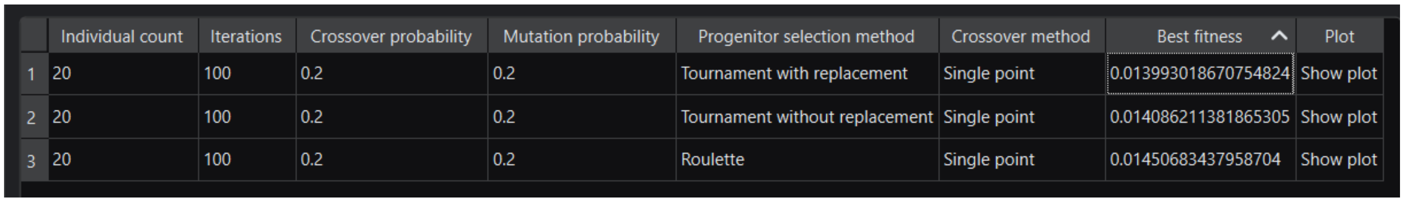
Dataset:	Quake		
Experiment count:	1		
Experiment seed:	33		
Mutation function:	Uniform		
Crossover function:	Single point		
Progenitor selection:	Roulette		
Individual count:	20	Increase by:	20
Iterations:	100	Increase by:	0
Crossover probability:	0.2	Increase by:	0
Mutation probability:	0.2	Increase by:	0





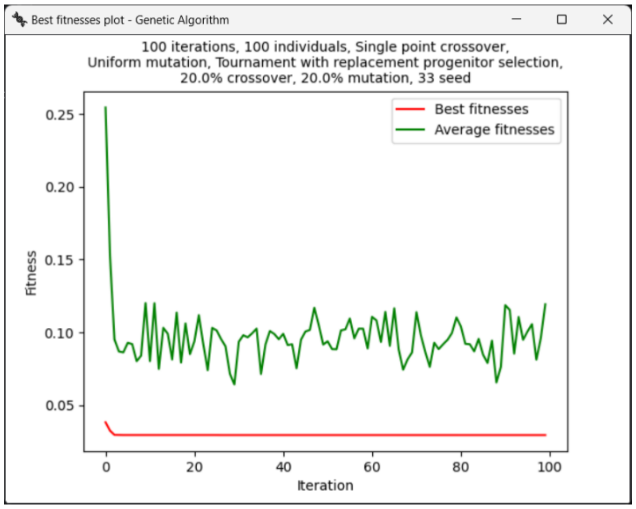


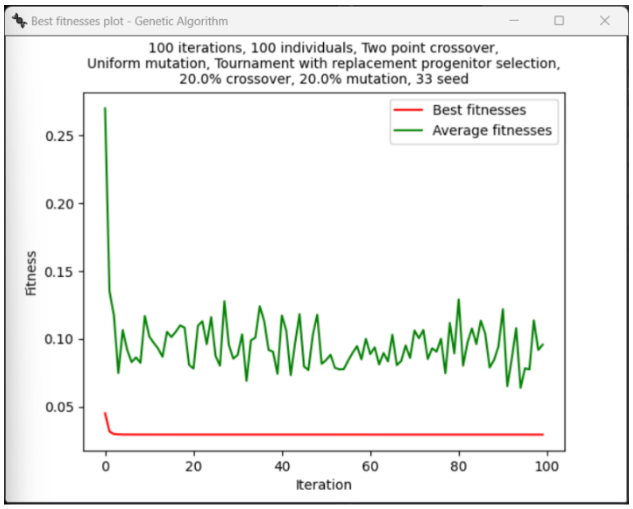
In this case the roulette is worse than the tournament functions. The tournament functions are less stable but achieve a better result way faster. The fitness average stays between 0.05 and 0.1 quickly, whereas in the roulette case It requires more iterations. The tournament functions achieve 0.014 best fitness, but the roulette experiment reaches 0.0145.



	Individual count	Iterations	Crossover probability	Mutation probability	Progenitor selection method	Crossover method	Best fitness	Plot
1	20	100	0.2	0.2	Tournament with replacement	Single point	0.013993018670754824	Show plot
2	20	100	0.2	0.2	Tournament without replacement	Single point	0.014086211381865305	Show plot
3	20	100	0.2	0.2	Roulette	Single point	0.01450683437958704	Show plot

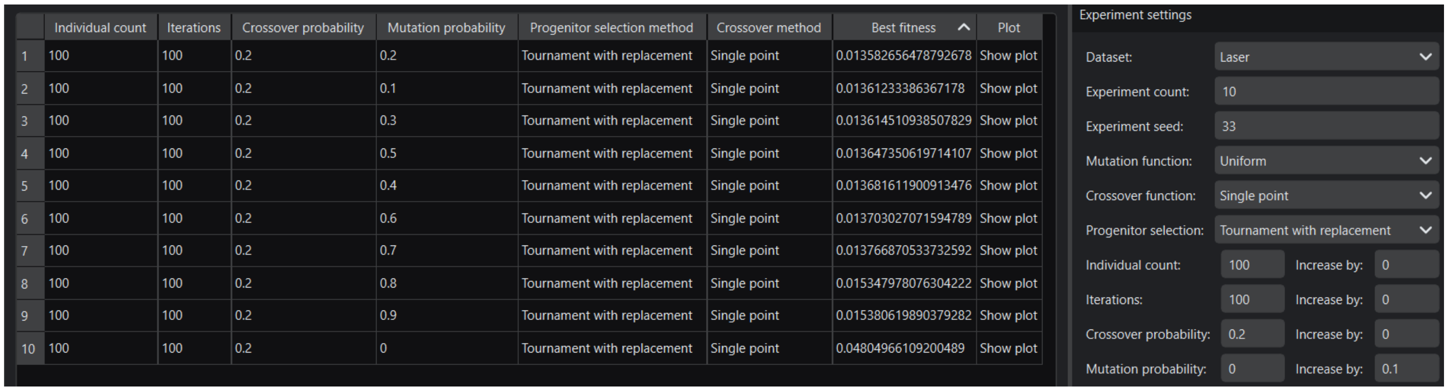
Let's try to do the same experiment with the crossover functions.





In this case, there are no noticeable differences between functions. However, this is just a single comparison that uses certain parameters. Is perfectly feasible that a different set up can show a difference between functions.

The mutation probability is an important factor that determines how important is the mutation function. This are the results of the experiment:



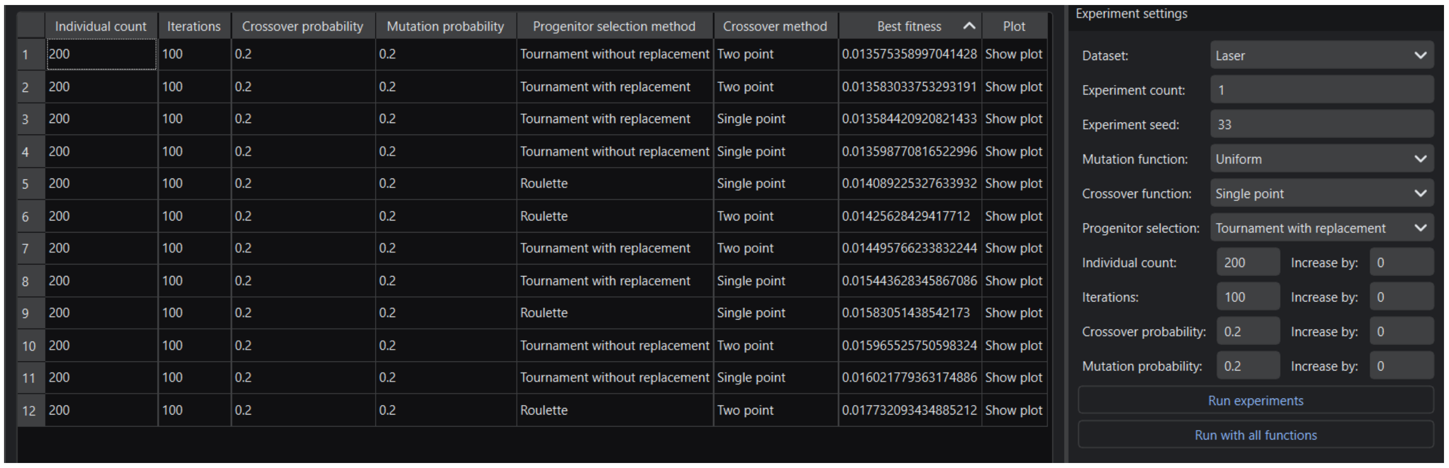
	Individual count	Iterations	Crossover probability	Mutation probability	Progenitor selection method	Crossover method	Best fitness	Plot
1	100	100	0.2	0.2	Tournament with replacement	Single point	0.013582656478792678	Show plot
2	100	100	0.2	0.1	Tournament with replacement	Single point	0.01361233386367178	Show plot
3	100	100	0.2	0.3	Tournament with replacement	Single point	0.013614510938507829	Show plot
4	100	100	0.2	0.5	Tournament with replacement	Single point	0.013647350619714107	Show plot
5	100	100	0.2	0.4	Tournament with replacement	Single point	0.013681611900913476	Show plot
6	100	100	0.2	0.6	Tournament with replacement	Single point	0.013703027071594789	Show plot
7	100	100	0.2	0.7	Tournament with replacement	Single point	0.013766870533732592	Show plot
8	100	100	0.2	0.8	Tournament with replacement	Single point	0.015347978076304222	Show plot
9	100	100	0.2	0.9	Tournament with replacement	Single point	0.015380619890379282	Show plot
10	100	100	0.2	0	Tournament with replacement	Single point	0.04804966109200489	Show plot

Experiment settings

Dataset:	Laser		
Experiment count:	10		
Experiment seed:	33		
Mutation function:	Uniform		
Crossover function:	Single point		
Progenitor selection:	Tournament with replacement		
Individual count:	100	Increase by:	0
Iterations:	100	Increase by:	0
Crossover probability:	0.2	Increase by:	0
Mutation probability:	0	Increase by:	0.1

The results are remarkable since the optimal mutation value seems to be at 0.2 and increasing it causes worse results.

Finally let's try all possible combinations of the functions. The dataset chosen is the Laser. The results are sorted by fitness in ascending order, so the best results are on top.



	Individual count	Iterations	Crossover probability	Mutation probability	Progenitor selection method	Crossover method	Best fitness	Plot
1	200	100	0.2	0.2	Tournament without replacement	Two point	0.013575358997041428	Show plot
2	200	100	0.2	0.2	Tournament with replacement	Two point	0.013583033753293191	Show plot
3	200	100	0.2	0.2	Tournament with replacement	Single point	0.013584420920821433	Show plot
4	200	100	0.2	0.2	Tournament without replacement	Single point	0.013598770816522996	Show plot
5	200	100	0.2	0.2	Roulette	Single point	0.014089225327633932	Show plot
6	200	100	0.2	0.2	Roulette	Two point	0.01425628429417712	Show plot
7	200	100	0.2	0.2	Tournament with replacement	Two point	0.014495766233832244	Show plot
8	200	100	0.2	0.2	Tournament with replacement	Single point	0.015443628345867086	Show plot
9	200	100	0.2	0.2	Roulette	Single point	0.01583051438542173	Show plot
10	200	100	0.2	0.2	Tournament without replacement	Two point	0.015965525750598324	Show plot
11	200	100	0.2	0.2	Tournament without replacement	Single point	0.016021779363174886	Show plot
12	200	100	0.2	0.2	Roulette	Two point	0.017732093434885212	Show plot

Experiment settings

Dataset:	Laser		
Experiment count:	1		
Experiment seed:	33		
Mutation function:	Uniform		
Crossover function:	Single point		
Progenitor selection:	Tournament with replacement		
Individual count:	200	Increase by:	0
Iterations:	100	Increase by:	0
Crossover probability:	0.2	Increase by:	0
Mutation probability:	0.2	Increase by:	0

As stated previously, it is very hard to draw blunt conclusions due to the number of variables that affect the experiment. In this case the optimal result is at about 0.0135, but it can be improved by using more iterations and individual count. However, this makes the execution a lot slower.

The project has been completed successfully since all the goals have been completed. There are still a lot of features that could be added.

For example, the performance of the program could be improved by using a “Just in time” compiler like Numba which could speed up a lot the NumPy code.