Introducing GeneAl: a Genetic Algorithm Python Library

In this post, I'll introduce GeneAl, a python library for solving optimisation problems with genetic algorithms (GA). And in the process, we'll get to know the theory behind them and see how they work under the hood with python examples.

Diogo Matos Chaves Follow
Jun 25 · 6 min read

Image by Arek Socha via Pixabay

Genetic algorithms (GA) are an optimization and search technique based on the principles of genetics and natural selection, in essence mimicking the natural evolution process that we observe in life. Their general principle is based on the concept of having an **initial** population composed of several individuals — with each representing a particular solution to the problem — and allow it to evolve to a state that maximizes its overall **fitness**, using three main operators: **selection**, **crossover** and **mutation**. We'll look into these aspects a bit more in detail below.

Genetic Algorithms are nothing short of fantastic, as they can be applied to many kinds of optimization problems and find solutions to complex functions for which we do not have a mathematical expression. This comes at a cost of computational complexity though, as, for large populations, we'll have to evaluate the fitness of all individuals at every generation. If the fitness function is expensive, the algorithm run will be slow.

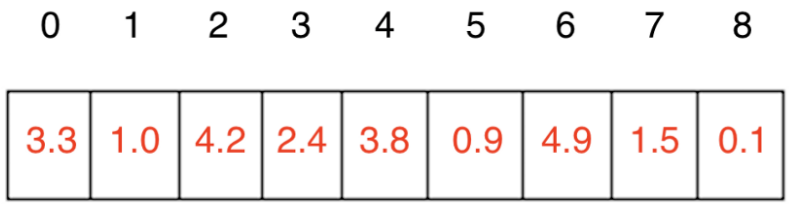
GA's can be divided into *Binary* and *Continuous*, depending on the type of problem we're optimizing for. Potentially all problems could be broken down as having their variables (*genes)* represented by binary strings, but in general, if the input space is real-valued, it makes more sense to use a *continuous* GA.

As there are fewer examples for continuous GA out there, the examples shown here will be for that version of GA.

Initialization

The search starts with a random population of *N* individuals. Each of those individuals corresponds to a *chromosome*, which encodes a sequence of genes representing a particular solution to the problem we're trying to optimize for. Depending on the problem at hand, the genes representing the

solution could be bits (0's and 1's) or continuous (real valued). An example of a real-valued *chromosome* representing a solution to a given problem with 9 variables (*genes*) is shown below.



Example of an individual's chromosome

```python
def initialize_population(pop_size, n_genes, input_limits):
    """
    Initializes the population of the problem according to the
    population size and number of genes.

    :param pop_size: number of individuals in the population
    :param n_genes: number of genes (variables) in the problem
    :param input_limits: tuple containing the minimum and maximum allowed

    :return: a numpy array with a randomly initialized population
    """

    population = np.random.uniform(
      input_limits[0], input_limits[1], size=(pop_size, n_genes)
    )

```

```
17      return population
```

initialization.py hosted with ❤️ by GitHub view raw

Fitness

The *fitness* of each individual is what defines what we are optimizing for, so that, given a *chromosome* encoding a specific solution to a problem, its fitness will correspond to how well that particular individual fares as a solution to the problem. Therefore, the higher its fitness value, the more optimal that solution is.

After all, individuals have their fitness score calculated, they are sorted, so that the fittest individuals can be selected for crossover.

```
1   def fitness_function(individual):
2       """
3       Implements the logic that calculates the fitness
4       measure of an individual.
5
6       :param individual: chromosome of genes representing an individual
7       :return: the fitness of the individual
8       """
9
10      raise NotImplementedError
```

fitness_function.py hosted with ❤️ by GitHub view raw

Selection

Selection is the process by which a certain proportion of individuals are selected for mating between each other and create new offsprings. Just like in real-life natural selection, individuals that are fitter have higher chances of surviving, and therefore, of passing on their genes to the next generation. Though versions with more individuals exist, usually the selection process matches two individuals, creating pairs of individuals. There are four main strategies:

pairing: This is perhaps the most straightforward strategy, as it simply consists of pairing the top fittest chromosomes two-by-two (pairing odd rows with even ones).

random: This strategy consists of randomly selecting individuals from the mating pool.

roulette wheel: This strategy also follows a random principle, but fitter individuals have higher probabilities of being selected.

tournament: With this strategy, the algorithm first selects a few individuals as candidates (usually 3), and then selects the fittest individual. This option has the advantage that it does not require the individuals to be sorted by fitness first.

A python implementation for the *roulette wheel* strategy is shown on the snippet below.

```
def select_parents(self, selection_strategy, n_matings, prob_intervals):
    """
    Selects the parents according to a given selection strategy.
    Options are:

    roulette_wheel: Selects individuals from mating pool giving
    higher probabilities to fitter individuals.

    :param selection_strategy: the strategy to use for selecting parents
    :param n_matings: the number of matings to perform
    :param prob_intervals: the selection probability for each individual in
     the mating pool.
    :return: 2 arrays with selected individuals corresponding to each parent
    """

    ma, pa = None, None

    if selection_strategy == "roulette_wheel":

        ma = np.apply_along_axis(
```

```
            lambda value: np.argmin(value > prob_intervals) - 1, 1, np.random.rand(n_mat
        )
        pa = np.apply_along_axis(
            lambda value: np.argmin(value > prob_intervals) - 1, 1, np.random.rand(n_mat
        )

    return ma, pa

```

selection.py hosted with ❤️ by GitHub view raw

Crossover

This is the step where new offsprings are generated, which will then replace the least fit individuals in the population. The idea behind crossing over individuals is that, by combining different genes, we might produce even fitter individuals, which will be better solutions to our problem. Or not, and in that case, those solutions won't survive to the next generations.

In order to perform the actual crossover, each of the pairs coming from the selection step are combined to produce two new individuals each, which will both have genetic material from each of the parents. There are several different strategies for performing the crossover, so for brevity, we'll only discuss one of them.

Supposing we have a problem defined by 9 variables, if we have 2 parents and we choose randomly the crossover gene as index 3, then each of the offsprings will be a combination of each parent, as shown in the diagram below.

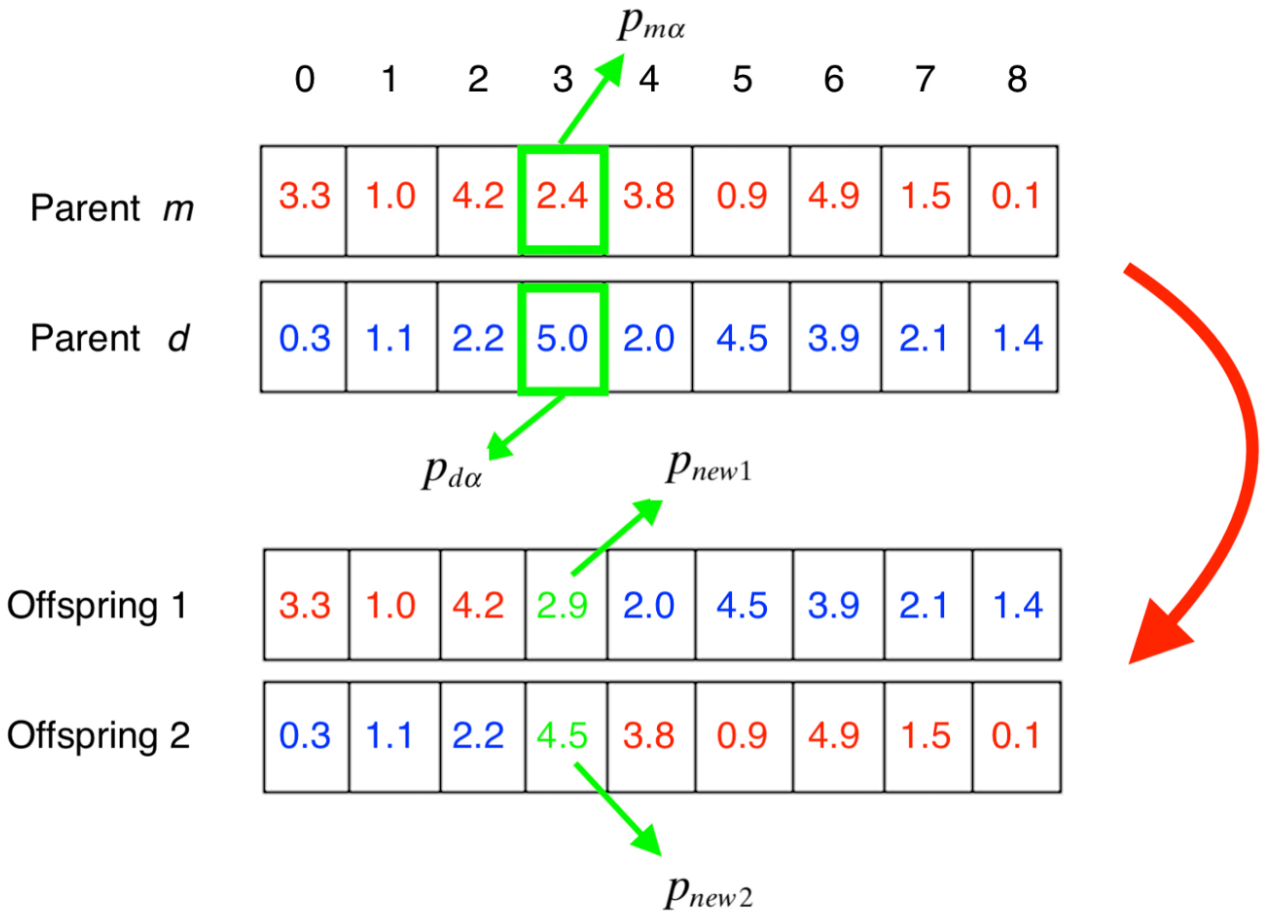


Diagram showing how parents are crossed over to generate new offspring

The crossover gene of each offspring is calculated according to the rule given by:

$$p_{new1} = p_{m\alpha} - \beta[p_{m\alpha} - p_{d\alpha}]$$
$$p_{new2} = p_{d\alpha} + \beta[p_{m\alpha} - p_{d\alpha}]$$

Equation for calculating new crossover genes

Where β will be a random number between 0 and 1. The python code for the crossover is given below.

Mutation

Mutation is the process by which we introduce new genetic material in the population, allowing the algorithm to search a larger space. If it were not for mutation, the existing genetic material diversity in a population would not increase, and, due to some individuals "dying" between generations, would actually be reduced, with individuals tending to become very similar quite fast.

In terms of the optimization problem, this means that without new genetic material the algorithm can converge to local optima before it explores an enough large size of the input space to make sure that we can reach the global optimum. Therefore, mutation plays a big role in maintaining diversity in the population and allowing it to evolve to *fitter* solutions to the problem.

The most simple way we can do this is, given a certain *mutation rate*, to randomly choose some individuals and some genes and assign a new random number to those positions. This is exemplified in the diagram and code snippet below.

Mutation of two genes in an individual

Solver

Now it's time to tie it all together. Using the operators that we defined above, the algorithm can now solve the problem, with the actual main cycle of the algorithm being implemented in just a few lines of code. The flowchart of the algorithm, as well as an example implementation in python are shown below.

Flowchart of the Genetic Algorithm

Introducing GeneAl

GeneAl is a python library implementing Genetic Algorithms, which can be used and adapted to solve many optimization problems. One can use the provided out-of-the-box solver classes — **BinaryGenAlgSolver** and **ContinuousGenAlgSolver** — , or create a custom class which inherits from one of these, and implements methods that override the built-in ones. It also has support for solving the (in)famous **Travelling Salesman Problem**.

For brevity, we'll only see how to use the continuous version — keeping in line with this post — , but for more details, check out the README of this project.

The first thing would be to install the package, which can be made through `pip`, as such:

```
pip install geneal
```

After the installation completes, one is ready to use it. So let's see how we can use the **ContinuousGenAlgSolver** class.

As a bare minimum, the class requires the user to provide the number of genes present in the problem, as well as to provide the custom defined fitness function. For convenience, the library provides some default fitness functions that can be used for testing.

With the initialization done above, the class will solve the problem using default values for all the parameters. If we wish to have more control over the algorithm run, we will want to adjust these, and that can be done as shown below:

Finally, this class allows the user to specify the type of problem — if the possible values are integers or floats — , as well as the variables' limits, in

order to limit the search space.

This completes this short introduction to the library. If you want to know more, check out the GitHub repository, which has more information :)

Thanks for reading!

Genetic Algorithm Python Optimization Artificial Intelligence