Object Oriented, Multiprocessing Genetic Algorithm for N variable optimization

Alessandro Brovelli

November 28, 2024

Abstract

An object oriented genetic algorithm was developed in Python. The scope of the code is to optimize (maximize or minimize) a $N \geq 1$ variable fitness function in a provided search space. It can employ a set of two selection, crossover and mutation methods, set by the user. The algorithm was tested on a 2 dimensional function (Styblinsky-Tang), for which parameters were tuned for optimal accuracy for each combination of evolutionary methods. To verify the stability around these optimized settings, a perturbative analysis was performed. Finally a speed test was carried out, to see how the different parameters affect the algorithm performance.

1 Algorithm description

1.1 General structure

The algorithm structure is divided in three classes. The first one, *Individual*, defines a single candidate solution, expressed as a genome of N binary strings (*chromosome*) of set length (*number of genes*). Each chromosome can be decoded into a decimal value the provided search space. The decoded genome will be the input of the N variable function used to evaluate the individual's fitness. Within itself, an individual can only mutate, given the mutation scheme and probability

A second class *Population* establishes a candidate solutions population (a list of individuals), and contains a method to create it given size, number of chromosomes and genes per chromosome. The other methods in this class allow the population to breed with a certain crossover method and probability, and to mutate by iterating through the individuals.

The main class, *Breeder*, inherits *Population* and defines the settings and procedures of the genetic algorithm. It includes methods to decode each genome, evaluate the population's fitness and to select worthy individuals for breeding. By creating an object, the user can set the problem size and type (maximize or minimize) and the various selection, crossover and mutation parameters and methods, which will be explained in detail in the next section. The whole algorithm can be executed by calling the method *start evolution*, which takes as input the fitness function, the search space, the error threshold and the generation number at which to stop if there is no convergence. If desired, at the end of the evolution the generational best fitness is plotted, alongside the fitness function in the search space with the final population scattered on it (if $N = 1$ or $N = 2$).

1.2 Selection

After evaluating the fitness of the current generation, the selection process begins. The aim is to choose the best individuals to reproduce, in order to evolve the population towards the optimal solution. Two selection methods are available: *tournament* and *entropy*. Each methods iterates through random batches of the population, in which an individual can be picked with set probability *ps*.

Tournament
A deterministic tournament selection method was employed. Only the individual with the best fitness is selected from each batch. This process is repeated until the selected population reaches the size of the initial one, which is therefore maintained constant through the generations.

Entropy
A more advanced selection is offered by the entropy method, inspired by simulated annealing optimization methods. Each individual in the current batch it is assigned a selection probability with the so called *Metropolis criterion* [1], which for a minimization problem would be

$$P(i) = \begin{cases} 1 & F_i \leq F_{old} \\ \exp\left(-\frac{\Delta F}{T}\right) & F_i > F_{old} \end{cases} \tag{1}$$

where F_i fitness of i^{th} individual, F_{old} best fitness of previous generation and $\Delta F = |F_i - F_{old}|$. The parameter T represents the *temperature* and it decreases through the generations with an exponential trend

$$T = T_0 \, \alpha^n \tag{2}$$

where T_0 is the initial temperature, n the generation number and $0 < \alpha < 1$. This method permits more exploration of the search space in the first genearations, when the temperature is higher, and more exploitation of the region where the solution will converge in the last generations. The selection iteration stops when the selected population reaches its former size or, to prevent being stuck in the loop when the temperature is too

low, when a maximum number of batch evaluations is reached. This logic allows the population size to decrease through the generations, by letting unfit individuals to freeze.

1.3 Crossover

Amongst the selected population, pairs of parents are chosen at random to reproduce. With a set probability (*pc*) they will either generate a pair of children and die or live and be part of the next generation. Each children chromosome will be the combination of the two same chromosomes of the parents, with a specific crossover method. In *one point* crossover the parents chromosomes are split at one random point, while in *two point* crossover they are split in three at two random points. They are then recombined as shown in figure 1

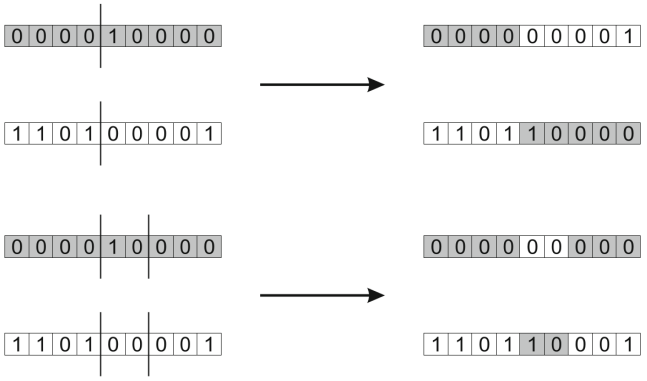

Figure 1: Illustration of one point (top) and two point (bottom) crossover [1]

1.4 Mutation

After the crossover, the children chromosomes can be mutated. This inserts a further random element in the evolution, allowing the population to explore the search space more efficiently and to prevent premature convergence. The mutation probability (*pm*) is set by the user and it is the probability that a gene in the chromosome will be flipped. The mutation method can be either *bit flip* or *swap*. In the first case, a random gene is chosen and its value is inverted, while in the second case two random genes are swapped.

2 Algorithm testing

The algorithm was tested on the 2 dimensional Styblinsky-Tang function

$$f(x,y) = \frac{1}{2}\left(x^4 - 16x^2 + 5x + y^4 - 16y^2 + 5y\right) \qquad (3)$$

which has a global minimum at $f(-2.903534, -2.903534) = -78.3323$. The search space was $(x, y) \in [-5, 5] \times [-5, 5]$, the convergence threshold was set to 10^{-9} and the maximum number of generations at 500. The plot [] shows the result for VVV OPTIMAL SETTINGS. These settings are the ones that give optimal accuracy for such combination of settings. They were found by tuning the algorithm, a procedure that will be described in the following section.

3 Parametric analysis

3.1 Tuning with multiprocessing

Tuning such an algorithm can be quite a time consuming task. For tournament selection, the total number of combinations of settings that can be tested are

$$2^2 \prod_{i=1}^{5} n_i \qquad (4)$$

where 2^2 are the combinations of the 2 available crossover and mutation methods and n_i are the lenghts of test parameter vectors which contain values of population size, number of genes and probabilities *ps*, *pc* and *pm*. For entropy selection, the number of test vectors will be 7, considering also T_0 and α. To reduce the computational time for this process, parallel computing with multiprocessing was employed. Every combination of settings was tested once, and the most promising ones (error on the fitness function lower than 10^{-4}) were retested 5 times. Only if every test yielded an error lower than the threshold, the settings were chosen as optimal.

3.2 Perturbative test

To verify the stability of the optimal settings, a perturbative analysis was carried out. The parameters were perturbed randomly with various maximum amplitudes, and the best fitness was recorded for each test. To speed up the process, multiprocessing was used here as well. VVV PLOTS AND STABILITY CONSIDERATIONS

3.3 Speed test

4 Conclusions and future developments

References

[1] J.E. Smith A.E. Eiben. *Introduction to Evolutionary Computing*. Springer, second edition, 2015.