Selection methods of Genetic Algorithms

Algorithms - A.Y. 2020/2021

Written by Francesco BARTOLI
Chiara BOETTI

Contents

1 Introduction: what is a Genetic Algorithm?

Genetic Algorithms (GA) are a subset of the deeper branch of Evolutionary Computation. They are stochastic search procedures based on the concepts of Mendelian genetics and Darwinian evolution. The theory of GAs was developed by John Holland at the University of Michigan in 1960 and he was aiming to:

(i) abstract and explain adaptive processes regarding natural selection and genetics;

(ii) design appropriate systems software that retain such mechanisms.

Essentially, a GA is a self-learning algorithm that uses past attempts to generate new and better attempts in order to solve a specific problem. It is composed by five distinct parts: initialization, fitness assignment, selection, crossover and, finally, mutation.
Once a population of individuals has been chosen, usually at random, it is used as initial input for the optimization problem. Then, we evaluate the fitness of each individual through an adaptation function and, with a selection mechanism, we pick some elements to be the parents of those of the next generation. Hence, we take crossover and mutations on such individuals so that, the next generation is finally formed. This process is repeated until a certain condition is satisfied: individuals keep evolving until the best possible answer is found. This is in line with the Darwinian Theory of "Survival of the Fittest".

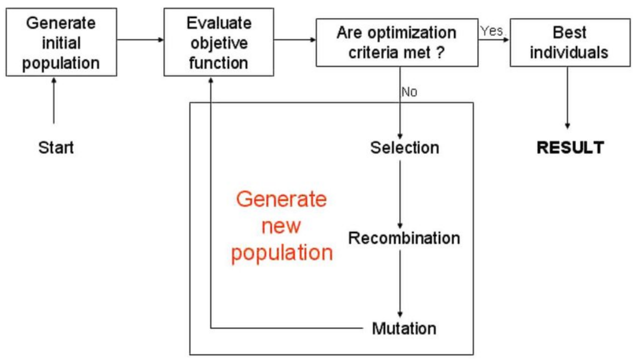

Figure 1: Scheme of a Genetic Algorithm

What are the motivations of using a GA rather than other Evolutionary Algorithms?
GAs are a valid approach in optimization problems which require an efficient and effective search of some adapting parameters: they provide "good enough" and "fast enough" solutions. For this reason, they are actually used to solve many different real-world problems, spacing in field like economics, computational science, bioinformatics, medicine, engineering, etc.
For example, they work well in multi-modal optimization, in which we have to find multiple optimum solutions, and they have been also used to solve the well-known combinatorial Traveling Salesman Problem.
In financial markets, GAs are commonly used to find the best combination of parameters in a trading rule whereas, in biology, they can help in DNA Analysis. Other applications concern the training of neural networks, the image processing and solving scheduling problems, such as an office timetable. Recently, agriculture systems have used GAs to determining optimal cropping patterns and nutrition combinations in greenhouses.

What about the limitations of GAs?
Unfortunately, GAs are quite computationally expensive. Calculating repeatedly fitness values and storing the whole population need computers with high processing power and capacity. Moreover, if not implemented properly, it may not converge to the optimal solution and, in case of convergence, there are no guarantees on the quality of such solution, since GAs are based on some stochastic assumptions.

Before moving to the next part, let us see the pseudocode of a general GA:

Algorithm 1 Genetic Algorithm

```
function GA($n\_gen, r\_cross, r\_mut$)
    choose an initial population of individuals
    while not satisfied and iteration $\leq$ n_gen do
        evaluate the fitness of all individuals
        if fitness satisfies the criteria then
            exit from the loop
        end if
        select parents for building mating pool
        for n times as the length(population) do
            select two parents from mating pool
            apply crossover operator on the two parents
            apply mutation operator on the child
            add the child to the new population
        end for
        replace old population with the new population
        iteration = iteration + 1
    end while
    return best solution
end function
```

As we can see, this algorithm requires to find the right parameters in order to describes properly the real process. In particular, we have to choose the correct population size, how to select and recombine parents and find the probabilities of crossover and mutation.
In this paper, we will only focus on the selection step.

2 Selection Methods of Genetic Algorithms

A suitable evolution can happen only if we choose the best individuals. It is then necessary picking such elements in the correct way otherwise, without the selection step, a GA would be just a random method with different values each time and it would not work so well.
Just as one could expect, the basic idea behind selection methods is that individuals with an higher fitness value are more likely to being selected for the next generation. Although it could seem a simple concept, an efficient method is not at all trivial to implement. The behaviour of a GA is based on the balance between exploration and exploitation of the search space and it is carried on by selection, crossover and mutation. In particular, a strong selection pressure may cause a premature convergence to a local optimum whereas, with a low selection pressure, the results of GA could differ too much from one run to another.
Selection can occurs in two places:

- in the Parent selection, where only a few individuals of the current generation are allowed to reproduce;
- in the Survivor Selection, where we decide which offspring move into the next generation.

Later, we will only consider Parent selection methods but, for curiosity sake, we write just the few important aspects of the Survivor selection. Which selection method should be used depends upon the domain and the application of the problem.

The keyword to describe the Survivor selection is *replacement*: we have to determine which individuals are to be kept in the next generation and which are to be discarded. Hence, the population size is reduced from $n + l$ to n, where l represents the number of offspring of the previous generation.

This type of selections can be classified in:

I) Age-based Selection: each individual is kicked out of the population after a finite number of generations, even though it has a high fitness. Thus, when we decide which data structure is suitable for the implementation, we have to take in mind also the concept of age, i.e. a counter for generations of the element;

II) Fitness-Based Selection: the least fit individuals are replaced by the children. We can actually pick the least fit elements with some variations of the Parent selection methods that we will see in the next pages. Alternatively, one can use the Elite Preservation mechanism: a few best chromosomes are copied in the next generation or, symmetrically, an equal number of the worst solutions are eliminated. There is no risk in loosing the best solutions since they are the ones which survives in the next generation and so the performance of GA improves a lot. However, because of elitism, we could loose the diversity of the population and thus converge to local solutions.

Let us now come back to the Parent selection.
Here, we choose an individual of the population to be a parent for the next generation based on its fitness. This kind of selections are divided in three subcategories:

I) Fitness-based Selection: an element is chosen to become a parent according to its fitness value. The more popular is an individual in terms of fitness, the more likely is to be picked in the next generation. Note that this selection does not work when fitness can take a negative value. We will see the Fitness Proportionate selection in detail in a few lines;

II) Ordinal-based Selection: individuals are ranked according to their fitness values. It works with negative fitness values and it can be further divide in two types: Rank-based selection and Tournament selection.

   In a Rank-based selection, the population is sorted according to an objective value and the probability of being picked is related to such score. This approach is more robust than the fitness proportional one and it is mostly used in the final runs of GAs.

   We will also focus and implement the Tournament selection, which is extremely popular in literature and works well in every step of the GA;

III) Threshold-based Selection: a truncation threshold is used as parameter so that individuals below it do not produce offspring. We will see and implement the Truncation selection.

The trivial strategy in which there is no selection pressure is called Random Selection, since we pick parents randomly from the previous generation. As one can imagine, this method is usually avoided even though, in terms of complexity, is the best one: choosing at random an individual is basically a constant operation.

2.1 Fitness Proportionate Selection

The first type of selection we study is the Fitness Proportionate selection. It was introduced when GA were first developed and it is still very popular. With this method, the probability that an individual becomes a parent is proportional to its fitness, as one can easily guess by its name. In this way, fitter individuals have higher chance of propagating their features to the next generation. If we think at the fitness as measurement of better chromosomes, then individuals evolve better and better over time.
Due to its historical background and similarity with the physical roulette wheel, Fitness Proportionate is commonly known as Roulette Wheel selection.
Imagine a roulette wheel whose slots are weighted according to the elements' fitness values and throw a marble. We select the one which corresponds to the place where the marble stops. It is then trivial why individuals with bigger fitness are selected more times: the greater the piece of pie, the more likely is to stop there.

Let us now see how to implement this method. Instead of "building" a roulette wheel, we can just focus on the probabilities by normalizing fitness.
First of all, map the population as a contiguous segment from 0 to 1, such that each individual's section is proportional to its fitness. Then, generate a random number uniformly between 0 and 1. Select the individual whose segment spans the random number.
The mating pool is obtained by repeating this process as many times we need and by taking independent uniformly distributed random numbers.

Assuming that the population size is n and t_{el} is the number of times we execute the *while* loop with respect to the element el, then the pseudocode of this selection method is:

Algorithm 2 Fitness Proportionate selection

```
                                                          Cost   Number of times
 1: function FITPROP_SELECT(population)
 2:     compute sum of fitnesses                          c1     1
 3:     compute fitness_prob                              c2     1
 4:     for i = 1,2,...,n do                              c3     n+1
 5:         set random_prob                               c4     n
 6:         while element not added do                    c5     $\sum_{el=1}^{n} t_{el}$
 7:             if fitness_prob(element) > random_prob then   c6     $\sum_{el=1}^{n}(t_{el}-1)$
 8:                 add element to mating pool            c7     n
 9:                 exit from the while loop              c8     n
10:             end if
11:             element = element +1                      c9     $\sum_{el=1}^{n}(t_{el}-1)$
12:         end while
13:     end for
14:     return mating pool                                c10    1
15: end function
```

Hence, we have:

$$T(n) = c_1 + c_2 + c_3(n+1) + c_4 n + c_5 \sum_{el=1}^{n} t_{el} + c_6 \sum_{el=1}^{n} (t_{el} - 1) + (c_7 + c_8)\, n + c_9 \sum_{el=1}^{n} (t_{el} - 1) + c_{10}$$

where:

- $c_1, c_2 = O(n)$, because we have to go through all the elements in order to sum fitness values and compute their probabilities;
- $c_3, c_5, c_6 = O(1)$, because they are just boolean checks;
- $c_4 = O(1)$: looking at the documentation of the most commonly used algorithms to generate an uniform random between 0 and 1, we saw their complexity order is constant. Indeed, they must solve a integral whose primitive form is simple, so that they only involve arithmetic operations;
- c_7 depends on the implementation, but we can assume at worst $c_7 = O(n)$;
- $c_8, c_9, c_{10} = O(1)$.

In the 6th line, the one involving the *while* condition, we have to take care about the three possible cases. To see better what are the changes, we assume that c_7 is a constant operation.

<u>Best-case scenario:</u> $t_{el} = 1$, i.e. we have to compare the fitness probability only of the first element. This happens whether the random probability, that have been generated casually in

the 4th line, is smaller than the very first fitness, hence close to zero. In this case,

$$\begin{aligned}
T(n) &= c_1 + c_2 + c_3(n+1) + c_4 n + c_5 \sum_{el=1}^{n} t_{el} + c_6 \sum_{el=1}^{n} (t_{el} - 1) + (c_7 + c_8)\, n + c_9 \sum_{el=1}^{n} (t_{el} - 1) + c_{10} \\
&= c_1 + c_2 + c_3(n+1) + c_4 n + c_5 (n+1) + (c_6 + c_7 + c_8 + c_9)\, n + c_{10} \\
&= a + b \cdot n
\end{aligned}$$

and therefore the time complexity is $O(n)$.

Worst-case scenario: $t_{el} = el$, i.e. we have to compare the fitness probability of every element. This happens whether the random probability is close to one, hence greater than the last fitness, and so the "correct" element is at the end. In this case,

$$\begin{aligned}
T(n) &= c_1 + c_2 + c_3(n+1) + c_4 n + c_5 \sum_{el=1}^{n} t_{el} + c_6 \sum_{el=1}^{n} (t_{el} - 1) + (c_7 + c_8)\, n + c_9 \sum_{el=1}^{n} (t_{el} - 1) + c_{10} \\
&= c_1 + c_2 + c_3(n+1) + c_4 n + c_5 \frac{n^2+n}{2} + c_6 \frac{n^2-n}{2} + (c_7 + c_8)\, n + c_9 \frac{n^2-n}{2} + c_{10} \\
&= a + b \cdot n + c \cdot n^2
\end{aligned}$$

so that the time complexity is $O(n^2)$.

Average-case scenario: $t_{el} = \frac{el}{2}$, i.e. we have to compare the fitness probability of half of the elements. This happens when the random probability is near the mean value of the fitness probabilities. Since we have roughly the same computation of the Worst-case scenario, the time complexity is again $O(n^2)$.

Therefore, the time complexity of this algorithm is $O(n^2)$. This does not cause any relevant issues for relatively small dataset, but we could look for better algorithms.
For the moment, let us focus only on the inner *while* loop, i.e. the part which actually selects the parent. As one can easily see, the time complexity is just $O(n)$, supposing again that c_7 is a constant operation. As said before, we can interpret this algorithm as a roulette wheel, whose sectors are size proportional to the individuals' fitness values. Picking an element is equivalent to choosing randomly a point on the wheel. Then, it is like we are searching the right location to stop. In fact, the *while* loop is basically a searching algorithm. This means that we can actually reduce the computational cost from $O(n)$ to $O(\log n)$. The latter case corresponds to the binary search, where we assume the input population is already sorted with respect to the fitness. Thus, with that hypothesis, the time complexity of the whole algorithm is $O(n \log n)$.
Just for curiosity sake, we recall that the best sorting algorithm is the *Merge-Sort* algorithm, whose cost is $T(n) = O(n \log n)$. This is quite a positive news: if we sort the population according to the fitness values, then the time complexity of the adjusted Roulette Wheel Selection will still be $T(n) = O(n \log n)$.

From a more general point of view, with the Fitness Proportionate selection, the probability of choosing an individual depends directly on its fitness and, in theory, all elements can be put in the mating pool. However, there could be problems of premature convergence to a local optimum: if there exist few individuals with prominent values, we can loose the diversity of the population. In this case, the degree for which better individuals are favoured, also known as Selective Pressure, is large.
On the other hand, if individuals' fitness are all very similar to each other, then there is almost no dispersion and Fitness Proportionate selection is just a random selection. This is the case of low Selective Pressure.
Finally, Fitness Proportionate selection algorithm provides a zero bias, but it does not guarantee minimum spread.

2.2 Rank Selection

Some drawbacks of the previous selection method are that it does not work with negative fitness values and it can lead to a premature convergence of the GA to a local optimum. Rank selection methods has been introduced to prevent these problems and they are supported by a simple idea: removing the dependence of fitness values by ranking individuals. Let us see it better.
In the selection step of every GA, we consider a population of length n that has been evaluated: each individual is associated to a certain number, which is referred to as fitness. We can use it to sort individuals and assign a particular score: the best individual has rank n, the second one has $n - 1$ and so on, until the rank 1 is assigned to the worst one. Then, the probability of choosing a certain element depends only on its rank, as its name suggests.

The population can be ranked either in a linear way, i.e. by increasing order of fitness, from 1 to n, or in a non-linear way, for instance by using a non-linear distribution such as the exponential one. Respectively, we talk about Linear Rank selection and Non-Linear Rank selection.
In a general Rank selection, each element el has probability of being selected given by the expression:

$$p(el) = \frac{rank(el)}{\sum_{el=1}^{n} rank(el)}$$

In the particular case of Linear Rank selection, $rank(el)$ is just the position of el and so $\sum_{el=1}^{n} rank(el)$ is the sum of the first n integers:

$$p(el) = \frac{2 \cdot rank(el)}{n \cdot (n+1)}$$

Since $1 \leq rank(el) \leq n$, then:

$$\frac{2}{n \cdot (n+1)} \leq p(el) \leq \frac{2}{n+1}$$

In the Non-Linear Rank selection, a common choice for computing rank-probabilities is given by the expression:

$$p(el) = exp\left\{\frac{-rank(el)}{c}\right\} \qquad \text{with} \qquad c = \frac{2 \cdot n \cdot (n+1)}{6 \cdot n + n + 1}$$

In the next pseudocode, we assume to rank and compute the rank-probabilities inside the selection algorithm. Of course, we could implement this part outside but, in that case, we should still take into consideration it in analysing the costs of the algorithm.
Here, the time complexity is:

$$\begin{aligned} T(n) = {} & c_1 + c_2 + c_3 + c_4(n+1) + (c_5 + c_6)n + c_7 \sum_{el=1}^{n} t_{el} + (c_8 + c_9) \sum_{el=1}^{n} (t_{el} - 1) \\ & + (c_{10} + c_{11})n + c_{12} \sum_{el=1}^{n} (t_{el} - 1) + c_{13} \end{aligned}$$

where t_{el} is the number of times we execute the *while* loop with respect to the element el.

Algorithm 3 Rank selection

```
                                                   Cost    Number of times
function RANK_SELECT(*population*)
    sort population by fitness                     c1      1
    assign *rank*                                  c2      1
    compute $p$ of every element                   c3      1
    for i = 1,2,...,n do                           c4      n+1
        set $r$ random probability                 c5      n
        set $s = 0$                                c6      n
        while element not added do                 c7      $\sum_{el=1}^{n} t_{el}$
            $s = s + p(el)$                        c8      $\sum_{el=1}^{n} (t_{el} - 1)$
            if $s > r$ then                        c9      $\sum_{el=1}^{n} (t_{el} - 1)$
                add element to mating pool         c10     n
                exit from the while loop           c11     n
            end if
            element = element +1                   c12     $\sum_{el=1}^{n} (t_{el} - 1)$
        end while
    end for
    return mating pool                             c13     1
end function
```

We notice that:

- c_1 depends the sorting procedure;
- c_2 depends on our ranking method, but we can assume it is a set of arithmetic computations done for every element of the population, thus $O(n)$;
- c_3 as before, it is just an arithmetic computation done for every element of the populations, hence $O(n)$;
- $c_4, c_7, c_9 = O(1)$, because they are just boolean checks;
- $c_5 = O(1)$, same consideration as for the Fitness Proportionate algorithm: generate an uniform random is constant;
- $c_6, c_8, c_{11}, c_{12}, c_{13} = O(1)$ because arithmetic computations or constant operations;
- c_{10} depends on the implementation, but we can assume at worst $c_{10} = O(n)$.

What can we say about the computational cost in the Worst, Average and Best scenarios? Before starting the analysis, we observe that:

(i) in line 2 we sort the population with respect to fitness values. As we have said before, the *Merge-Sort* algorithm is the best in terms of complexity and it costs $O(n \log n)$;

(ii) lines 3 and 4 are linear since we have to go through the whole population in order to assign ranks and compute the rank-probabilities: the computational costs is $O(n)$;

(iii) from line 5 up to the end, we have the same procedure as the Roulette Wheel selection. This is reasonable and also historically coherent: Rank selection has been proposed as an improvement of that selection method. Hence, we have $O(n)$ in the Best-case scenario and $O(n^2)$ in the other two, still assuming that adding an element to the mating pool is constant.

Therefore, we can conclude that $T(n) = O(n \log n)$ in the Best-case scenario and $O(n^2)$ in Average and Worst-case scenarios.

... Are we sure?

Recall what we have said at the end of the previous section: in the Fitness Proportionate selection the *while* loop is a searching algorithm and, if population is already sorted, then we could use a binary search to reduce the time complexity of the whole algorithm to $O(n \log n)$. In fact, we are exactly in this situation: rank probabilities are displaced according increasing values. Thus, instead of implementing the naive search version, we can perform a binary search and obtain that the time complexity of the adjusted Rank selection is $T(n) = O(n \log n)$ in all the three scenarios.

A few final comments. The Rank selection approach is more robust than the proportional fitness one. In particular, all the elements have a chance of being selected and hence it is likely to prevent too quickly convergences.
On the other hand, it is quite unfair to fitter individuals: it does not matter how big is the difference between two individuals as long as they are adjacent, according to their fitness values. To think at it differently, let us go back to the image of the roulette wheel. In the Fitness Proportionate selection, the piece of pie of an element was as wide as its fitness value. Here, instead, we do not care if such piece of pie is much bigger or just a little bigger than another. In the Rank selection we focus only on which is the largest one, which is the second largest one and so on and we pick parents just looking at their ranks.
For these reasons, it is mostly used when fitness values are all very similar to each other, situation that actually happens at the final runs of GAs.

2.3 Tournament Selection

It is one of the simplest methods of selection and it is based on a quite intuitive approach. First we choose a random set of k individuals from the total population and then we create a tournament among them, based on their fitness values. The winner is the one with the best fitness and it is added into the mating pool. This process ends when the mating pool is full. For instance, suppose $k = 2$. Then the Tournament selection picks two entities, compares their fitness and selects the best one to be able to reproduce.
With this method, the selection pressure can be easily controlled by changing the tournament size. In particular, a bigger k corresponds to a bigger selection pressure: if the tournament is done among k elements, then the competition is higher.

In our implementation, we consider a population of length n and a tournament pool of k individuals, with $2 \le k \le n$, so that one can choose k according to the specific problem.
Here, the pseudocode is:

Algorithm 4 Tournament selection

```
                                                              Cost    Number of times
 1: function TOURNAMENT_SELECT(population, k_tournament_pool)
 2:     for i = 1,2,...,n do                                  c1      n+1
 3:         set best                                          c2      n
 4:         for j = 2, ..., k_tournament_pool do              c3      n·(k+1)
 5:             select element                                c4      n·k
 6:             if fitness(element) > fitness(best) then      c5      n·k
 7:                 element = best                            c6      n·k
 8:             end if
 9:         end for
10:         add best to mating pool                           c7      n
11:     end for
12:     return mating pool                                    c8      1
13: end function
```

Our time complexity is described by the function

$$T(n) = c_1(n+1) + c_2 n + c_3(n(k+1)) + (c_4 + c_5 + c_6)nk + c_7 n + c_8$$

where:

- $c_1, c_3, c_5 = O(1)$, because they are just boolean checks;
- $c_2, c_6 = O(1)$, we assign a value to the best;
- $c_4 = O(1)$: we select an element and it can be done at random;
- c_7 depends on the implementation, but we can assume at worst $c_7 = O(n)$;
- $c_8 = O(1)$.

Hence, the time complexity of this algorithm is $O(n^2)$ in the Worst-case scenario, i.e. when $c_7 = O(n)$.

Note that some operations depend on the value k, the number of comparisons. Usually, this selection method is implemented with a tournament among two individuals and, of course, this changes the complexity of the algorithm. In fact, if we are using a data structure for which adding an element is constant, then the time complexity will just be $O(n)$. However, if for some reasons the number of comparisons is the number of elements in the population, then the complexity would be again $O(n^2)$.
Therefore, when c_7 is constant, we can say that $T(n)$ is between $O(n)$ and $O(n^2)$ or, more precisely, $T(k,n) = O(k \cdot n)$

Why one should use a bigger k instead of a smaller one?
Differently from the Fitness Proportionate, the Tournament selection does not depend strictly on the fitness. Thus, with a small number of comparisons, some "poor" elements can actually make it into the mating pool, creating a genetic difference in the population.

In terms of complexity, we have slightly improved the selection algorithm, but we could still look for a better situation, since this could cause some issues when using huge datasets. Finally, other than decreasing the computing time, the Tournament selection works on parallel architectures and even with negative fitness values. These features makes such method extremely popular in literature.

2.4 Truncation Selection

The Truncation selection is a very simple technique that sorts the candidates of the population according to their fitness and, then, only a certain portion p of the fittest individuals is selected and allowed to reproduce. It is less used in practice than other techniques, except when the population is too large and a mass selection is needed. We use an additional parameter, say truncation threshold, to indicate the proportion of the total elements to be selected as parents for the next generation. In general, it varies from 10% to 50%.

As always, we assume the population size is n, so that the pseudocode is:

Algorithm 5 Truncation selection

```
                                                          Cost    Number of times
1: function TRUNCATION_SELECT($population, threshold$)
2:     sort population by fitness                          c1          1
3:     take percentage above threshold as mating pool      c2          1
4:     return mating pool                                  c3          1
5: end function
```

The time complexity is easy to compute:

$$T(n) = c_1 + c_2 + c_3$$

where:

- c_1 depends the sorting procedure;
- c_2 depends on the implementation, for instance one should access to the population and read all the elements above the truncation threshold. Thus, at worst, $c_2 = O(n)$;
- $c_3 = O(1)$.

Then, the time complexity of the Truncation selection depends on how we sort the population. For instance, if we use the *Merge-Sort* algorithm, which is the best one we can hope for, the cost of this selection method is $T(n) = O(n \log n)$.

Therefore, Truncation selection is the fastest algorithm among the ones discussed here, but it also the least dispersive. This drawback is due to the fact that a consistent percentage of the population is not allowed to reproduce. Because only the very best elements are taken, the proposed final solutions could just be local optima and not always the best answers.

3 Implementation

GAs are a really popular tool for solving optimization problems because of their ability to finding a good solution in a relative small time. On the other hand, they are expensive enough in terms of memory space. Not only input parameters can be quite large, but a GA needs also a lot of space in order to improve the population as the time goes on. Therefore, choosing a suitable data structure is extremely important for the success of the algorithm.
Before entering in further details, let us first see what are the main features that a GA population must satisfies.

The very first step in the GA is the initialization process. We can think at the population as a set of chromosomes. Practically, it is a subset of solutions and it is often defined as a two dimensional matrix of dimensions (population size)x(chromosome size). These elements are updated as generation increases and they lead, possibly, to some optimal solutions.
The initial population, which corresponds to the first generation, is usually created randomly or heuristically. In particular, it has been observed that:

1. diversity of the solutions leads to optimality, which is usually the case of random elements;
2. with a heuristic initialization, population has similar individuals and very little diversity.

Hence, the best practice in initialization is not choosing one instead of the other, but combining them: seeding the population with some initial good solutions and filling the rest with random ones.
Another aspect that one should consider is that keeping a diversity in the population prevents the Premature Convergence. We have already discussed about it in the previous sections, but let us make it clearer.
In Evolutionary Algorithms, we talk about Premature Convergence when the algorithm converges before the global optimum solution is reached. For instance, this happens when it is stuck in some local minimum and offspring are not better than the parents. It is typically caused by:

(i) loss of diversity, i.e. results are all very similar to each others;

(ii) too strong selective pressure;

(iii) too much exploitation of the current generation.

Premature Convergence can be avoided by increasing population size, replacing similar individuals and using uniform crossover and segmentation of elements with similar fitness.
Finally, the population size should not be neither too large, as it might affect the speed of GA, nor too small, as it might not be good enough for the mating pool. An optimal population size is chosen through several trials and errors.

Keeping in mind these considerations, we decide to generate a population of random integer from 0 to 255 and convert them in a sequence of 9 bits. Thus, our population is only composed by 0 and 1. This was also motivated by the fact that genes and chromosomes can be coded as binary sequences. Moreover, since we only focused on the selection step and ignored the evaluation, we generate a random fitness number for every element.

We have implemented our selection methods in Python and the data structure that we have chosen to store our data is the NumPy array. This is actually quite reasonable: the NumPy library is well-known for yielding to specific high-performance numerical computations, especially when working with multidimensional variables. Looking at the online documentation, Python is a high-level dynamic language, easy to use but slower than a low-level language such as C. In fact, NumPy implements the multidimensional array structure in C and provides a convenient Python interface, so that it brings together high performance and simple usage.
Back to NumPy arrays, they are homogeneous blocks of data organized in a multidimensional finite grid. Data is stored in a contiguous block of memory in RAM. This leads to more efficient use of CPU cycles and cache. However, differently from lists, all the elements have to share the same data type which, in our case, is the integer numbers type. Since we work with a two dimensional array, we will also refer to it as matrix.
Once we have generated the population matrix and the fitness vector, we could also define another two dimensional array containing both chromosomes and fitness values with the `numpy.column_ stack` method. Actually, this is not at all necessary: thanks to the their implementation, we can just refer to elements by using their index position. In practice, we use this relationship to connect individuals with their respective fitness values.

Another interesting aspect that we have considered while implementing our algorithms was "how to add the selected element to the mating pool". This fact concerns the first three selection methods that we have studied: Fitness Proportionate, Rank and Tournament selection. As said before, the cost of adding an element to the mating pool depends on the data structure. In particular, using arrays, it depends on the position in which the individual is added, too. For instance, think for a moment at lists. They supports random access, hence the position of an element is relevant. The append operation is constant (except in the case list needs to be expanded), since we are just adding a new element in the last position: no changes of indexes. However, if we want to insert such element in any other position, it becomes a linear operation because indexes have to be adjusted. It is then natural to add the selected chromosome at the end of the list.
Now, using arrays is still a little bit different. NumPy arrays are stored as one contiguous block of memory, hence appending matrices within a loop is not computationally efficient: a new array has to be created at each iteration and that takes time. Thus, we solve this issue by using lists. In every step of the for loop, we append the selected individual to a list and wait until the end. This gives us a list of one dimensional NumPy arrays, Then, we stack them all vertically in the mating pool matrix with the `numpy.vstack` method. Of course, because of this last command, the speed of three algorithm decreases: converting a list into a NumPy array takes time. This represents also an example of trade-off between memory usage and amount of time.

Let us now have an insight in the `SelectWithArray.ipynb` file and add a few more comments about these algorithms. To measure the execution time spent in processing each method, we use `timeit` method, which is provided in Python library. In doing our analysis, we mainly focus on the time efficiency, trying different population sizes. We will evaluate the efficiency in terms of selecting the fittest individuals in the final section, with a simple example of GA.

1. Fitness Proportionate selection. To compute the element-wise ratio of these arrays, we do not need to write explicitly a for loop: we can just performs computations array-by-array instead of component-by-component. The theory behind this operation follows by how the NumPy arrays are implemented, i.e. using C loops. We note also that it is at least 4 times faster that the component-by-component way (of course, depending on the machine).

   All the operations in the for loops are constant, so we have to count only how many times we are doing them.

2. Linear Rank selection. We have implemented the naive version, i.e. without the binary search. Same considerations as above, even if it seems to be faster than the previous selection method.

3. Tournament selection. Again, operations are constant in time but repeated $k \cdot n$ times. This has a significant impact on the result and indeed it seems to be the slowest method.

4. Truncation selection. The complexity of the algorithm is dominated by the complexity of sorting, which is $O(n \log n)$. We have specified "reverse = True", which means individuals are sorted in a decreasing way with respect to their fitness: from the biggest to the smallest. Hence, we have to return the mating pool of the first elements of the sorted population. This algorithm is by far the fastest one.

With NumPy arrays, we can use the index of individuals in the population matrix to keep track of them and connect them to the fitness vector. However, we realize that, up to the selection step, the position of an element is not relevant: we pick a parent just according to its fitness. Let us change data structure: since we would work with unordered elements, we can just define our population through a set. With this kind of data structure, adding a new element or finding it is always constant. We expect, then, an improvement in terms of time performance.

From now on, we refer to the `SelectWithDict.ipynb` file.
We use the same initialization as before: chromosomes as binary sequences of 9 bits, zeros and ones, with random fitness values. Here, we construct the relationship between the two quantities by creating a list of lists. Indeed, looking at the external one, the first element is the chromosome and the second is the related fitness. In particular, the chromosome is defined through the internal list of 0 and 1. We store this population of size n in the dictionary data structure of Python where, as keys, we simply take the first n integers and as values the lists. Of course, this type of codification can be different according to the implementation we consider: the choice of the keys values is totally arbitrary as long as we consider hashable variables.
Another way of building our population dictionary is by using elements as keys and fitness as values. In this kind of implementation, however, we have to turn chromosomes list into, for instance, tuples or strings, which are immutable and hence hashable. Moreover, recall that a Python dictionary is a collection of distinct elements. Thus, in adding key-elements, we should take care of repetitions. If such chromosome is already present, then we have to update properly its fitness value.
In our code, we have chosen the first type of implementation: in the population dictionary, each key is an integer and each value is a list. This is mainly motivated by the fact that we are working in the GA context. Population needs to evolve and so, when returning the mating pool, chromosomes should be mutable. For completeness, we have also written the code to define the population dictionary of the second case, where keys are chromosomes tuple.

We measure execution times of our selection algorithms with the `timeit` method and, again, we just focus on the time efficiency.

1. Fitness Proportionate selection. Both retrieving and appending a value in a dictionary are close to O(1) and, in the Worst-case scenario, we have to repeat it n^2 times.

2. Linear Rank selection. Same considerations, but here we have also a sorting procedure. It is interesting that now Linear Rank is slower than the Fitness Proportionate selection,

differently from the array implementation. This could be due to the sorting. In the previous case, we sorted the fitness array and then we rearranged the population matrix by means of the sorted indexes. Here, we also sort fitness values but then we have to define with a Python for loop the sorted population dictionary.

3. Tournament selection. Here, we access to a value using the respective key in dictionary $k \cdot n$ times. We can notice practically a difference in time respect to the NumPy array method.

4. Truncation selection. The only computation that affects this algorithm is the sorting. Moreover, we notice that the time performance is again better than the one with arrays. Even though we need a loop to pick the elements, the constant operations, such as appending and using keys to retrieve values, do not give a relevant contribution to the total execution time.

In general, we see that the time efficiency of this selection methods is certainly better with the dictionary implementation. This is a positive aspect and confirms our previous assumption about a possible improvement.
However, dictionaries are worse than NumPy arrays in terms of memory usage and this feature is not at all irrelevant. We have already said that GAs are expensive as they need a lot of space to store all the population. Thus, when we have to decide which data structure is more suitable for the implementation, we prefer NumPy Arrays. Moreover, in studying a problem with an Evolutionary Algorithm, it could be necessary to use an ordered collection of items, and hence we need an practical way to track position of elements. In a similar situation, then, dictionaries are quite useless.

Just for curiosity sake, we have also tried our four selection methods with the list data structure of Python, referencing to `SelectWithList.ipynb` file. Quite surprisingly, we notice better time performances than the ones with NumPy array. The only exception is the Tournament Selection, where list are still faster, but with a little difference respect the others. We could explain this fact because of additional time taken by `numpy.vstack` method used to return a NumPy matrix.

4 Application and conclusion

We now present an application of GA for study a very simple optimization problem. In this way, we have a roughly measure of the efficiency of our selection methods. The complete GA comprehends both mutation and crossover operators and the evaluation of the new population to provide fitness values at every generation. Since we want to focus only on the performances of the selection algorithms, we choose a trivial problem, so that the transformation of the dataset is simple to write and control. See the `OptProblemGA.ipynb` file.

With the same initialization presented above, we generate our population as a list of 0 and 1. We evaluate individuals' fitness at each generation by means of an objective function. This latter is the one we want to optimize and it is the negative sum.
We implement the first three selection methods that we have studied: Fitness Proportionate, Linear Rank and Tournament. The Truncation selection has not been implemented because, in this example, we want to keep a constant population size and so this property would not be satisfied by such method.
We decide to use the dictionary data structure implementation we have presented above. Indeed, we work with a relative small population and an easy optimization problem, so we do not have many issues in terms of memory usage.

The crossover function consists in first choosing a pair of parents and a random point and then switching all their elements after such random point among the two parents. The mutation, instead, uses the fact that our individuals are composed by 0 and 1. Hence, a mutation

can be thought as a changing either from 0 to 1 or from 1 to 0. Thus, we can simply update the element as "1 - its old value". In this way, if the old value is 1, then it becomes $1 - 1 = 0$ and, respectively, if the old value is 0, we obtain $1 - 0 = 1$.
Of course, both mutation and crossover are not sure events: we have to consider some parameters to regulate the evolution so that our transformations happen with certain probabilities. In real applications, these two values, say *mutation rate* and *crossover rate*, are to be established properly to describe the real-world problem. Here, we also choose different parameters to spot different evolutions of the population.

As output of our GA, we return the best fitness values at the end of a fixed number of generations. In particular, every time a new best individual is found, we print it and we store its fitness. In this way, we can draw the evolution of the fitness across time measured in generations.

Let us now make a few comments.
Our selection algorithms works pretty well in this simple example. The first to converge is the fitness proportional one, but sometimes it converges to a local minimum. This is not too surprising, since it is totally in line with what we have said before, in section 2.1.
The Linear Rank selection has also a good performance: it finds better solutions than Fitness Proportionate. However, it is much slower: this could be due to the sorting part. Here, Linear Rank selection is an adequate choice because fitness values are not so different from each other. A possible improvement could be about the sorting operation. After few generations, our individuals tends to keep the same order. So, we could think a way to sort just a relevant part of the population. We still have to sort them, thus a $O(n \log n)$ time complexity, but now with n smaller than the population size. In this way, the whole process could be sped up.
The most intriguing selection method is the Tournament. With a low mutation rate, this method selects the best individual in the first generation and then it can not find an element with good enough fitness. As a result, differently from the other algorithms, the GA can not evolve any more. If we use a bigger mutation rate, for instance ~ 0.2, the selection improves and, if we double it again, this improvement is even more evident. The increasing in mutation rate is, on the other hand, a decreasing in stability and, in fact, the other two methods get worse. Look at figures in the next page for a visual example of this issue.
Therefore, Tournament selection is not very fitted in this type of setting.

To conclude: in this paper we have only focused on the selection step of a GA. We have studied some known methods and then we have implemented them. We have also tried to improve the execution time with the dictionary data structure and the space usage with the NumPy array data structure. Moreover, with this simple example, we clarify how even the simplest methods perform differently on the same problem.
In general, as we said at the beginning, the behaviour of GAs is based on the balance between exploration and exploitation of the search space. In a more complex situation, we do not use only one selection method, but a combination of them. Some procedures work better in the first steps of the GA, some others in the final ones and so, in order to take all the advantages, a mixture of these method is needed.

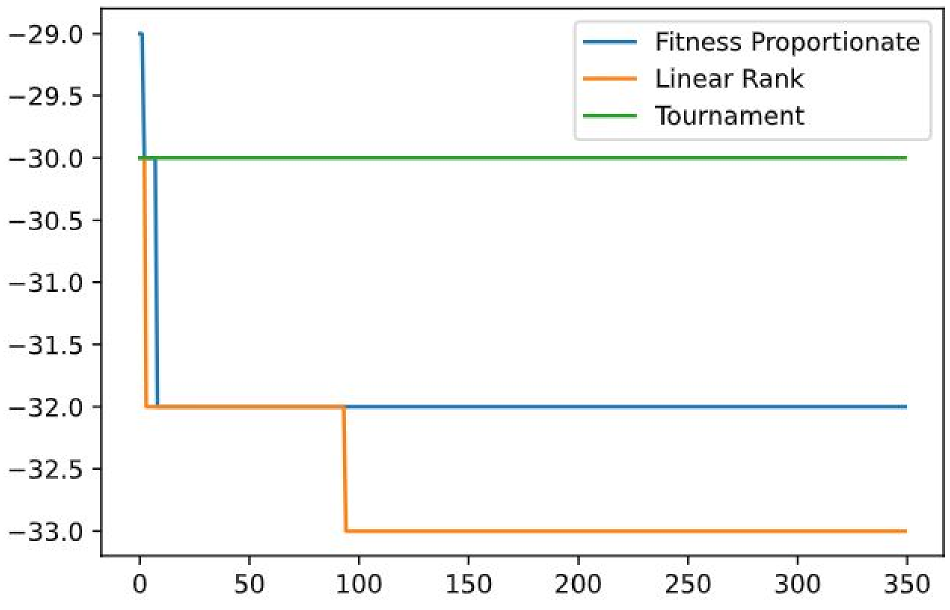

Figure 2: 350 iterations, population of 500 individuals with 40 bits, mutation rate = 0.025

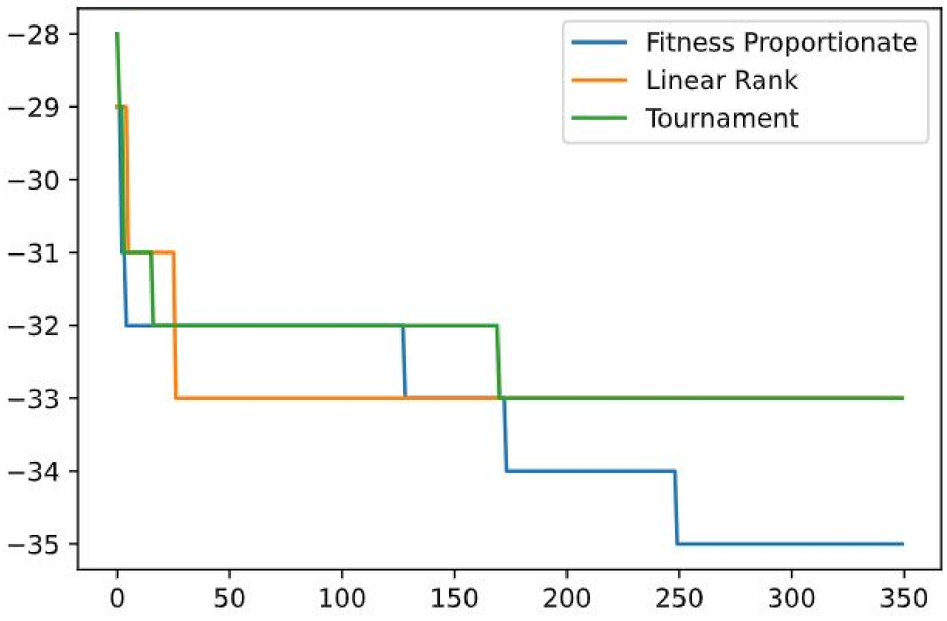

Figure 3: 350 iterations, population of 500 individuals with 40 bits, mutation rate = 0.5