**Definitions**

Genetic algorithms are a family of search algorithms inspired by the principles of evolution in nature. By imitating the process of natural selection and reproduction, genetic algorithms can produce high-quality solutions for various problems involving search, optimization, and learning. At the same time, their analogy to natural evolution allows genetic algorithms to overcome some of the hurdles that are encountered by traditional search and optimization algorithms, especially for problems with a large number of parameters and complex mathematical representations.

Genetic algorithms implement a simplified version of the Darwinian evolution that takes place in nature. The principles of the Darwinian evolution theory can be summarized using the following principles:

**The principle of variation.** The traits (attributes) of individual specimens belonging to a population may vary. As a result, the specimens differ from each other to some degree; for example, in their behavior or appearance.

**The principle of inheritance.** Some traits are consistently passed on from specimens to their offspring. As a result, offspring resemble their parents more than they resemble unrelated specimens.

**The principle of selection.** Populations typically struggle for resources within their given environment. The specimens possessing traits that are better adapted to the environment will be more successful at surviving, and will also contribute more offspring to the next generation.

An important enabler of evolution is crossover or recombination – where offspring are created with a mix of their parents' traits. Crossover helps in maintaining the diversity of the population and in bringing together the better traits over time. In addition, mutations – random variations in traits – can play a role in evolution by introducing changes that can result in a leap forward every once in a while.

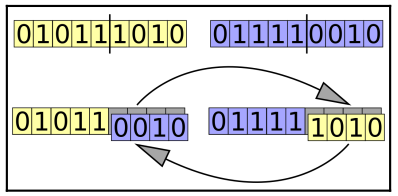
Genetic algorithms seek to find the optimal solution for a given problem. Whereas Darwinian evolution maintains a population of individual specimens, genetic algorithms maintain a population of candidate solutions, called individuals, for that given problem. These candidate solutions are iteratively evaluated and used to create a new generation of solutions. Those who are better at solving this problem have a greater chance of being selected and passing their qualities to the next generation of candidate solutions. This way, as generations go by, candidate solutions get better at solving the problem at hand.

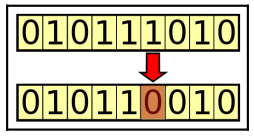
In nature, breeding, reproduction, and mutation are facilitated via the genotype – a collection of genes that are grouped into chromosomes. If two specimens breed to create offspring, each chromosome of the offspring will carry a mix of genes from both parents. Mimicking this concept, in the case of genetic algorithms, each individual is represented by a chromosome representing a collection of genes. For example, a chromosome can be expressed as a binary string, where each bit represents a single gene: 010111010. It’s an example of one such binary-coded chromosome, representing one particular individual.

At any point in time, genetic algorithms maintain a population of individuals – a collection of candidate solutions for the problem at hand. Since each individual is represented by some chromosome, this population of individuals can be seen as a collection of such chromosomes. The population continually represents the current generation and evolves over time when the current generation is replaced by a new one.

At each iteration of the algorithm, the individuals are evaluated using a fitness function (also called the target function). This is the function we seek to optimize or the problem we attempt to solve. Individuals who achieve a better fitness score represent better solutions and are more likely to be chosen to reproduce and be represented in the next generation. Over time, the quality of the solutions improves, the fitness values increase, and the process can stop once a solution is found with a satisfactory fitness value.

After calculating the fitness of every individual in the population, a selection process is used to determine which of the individuals in the population will get to reproduce and create the offspring that will form the next generation. This selection process is based on the fitness score of the individuals. Those with higher score values are more likely to be chosen and pass their genetic material to the next generation. Individuals with low fitness values can still be chosen, but with lower probability. This way, their genetic material is not completely excluded.

To create a pair of new individuals, two parents are usually chosen from the current generation, and parts of their chromosomes are interchanged (crossed over) to create two new chromosomes representing the offspring. This operation is called crossover, or recombination.

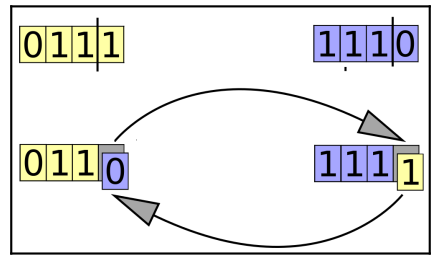
The purpose of the mutation operator is to periodically and randomly refresh the population, introduce new patterns into the chromosomes, and encourage search in uncharted areas of the solution space. A mutation may manifest itself as a random change in a gene. Mutations are implemented as random changes to one or more of the chromosome values; for example, flipping a bit in a binary string.

**The theory behind genetic algorithms**

The building-block hypothesis underlying genetic algorithms is that the optimal solution to the problem at hand is assembled of small building blocks, and as we bring more of these building blocks together, we get closer to this optimal solution.

Individuals in the population who contain some of the desired building blocks are identified by their superior scores. The repeated operations of selection and crossover result in the better individuals conveying these building blocks to the next generations, while possibly combining them with other successful building blocks. This creates genetic pressure, thus guiding the population toward having more and more individuals with the building blocks that form the optimal solution.

As a result, each generation is better than the previous one and contains more individuals that are closer to the optimal solution.

For example, if we have a population of four-digit binary strings and we want to find the string that has the largest possible sum of digits, the digit 1 appearing at any of the four string positions will be a good building block. As the algorithm progresses, it will identify solutions that have these building blocks and bring them together. Each generation will have more individuals with 1 values in various positions, ultimately resulting in the string 1111, which combines all the desired building blocks. 

The preceding image demonstrates how two individuals that are good solutions for this problem (each has three 1 values) create an offspring that is the best possible solution (four 1 bits, that is, the offspring on the right-hand side) when the crossover operation brings the desired building blocks of both parents together.

**The schema theorem**

A more formal expression of the building-block hypothesis is Holland's schema theorem, also called the fundamental theorem of genetic algorithms.

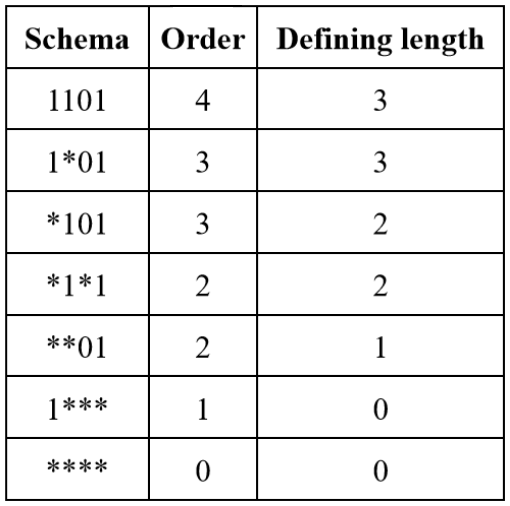
This theorem refers to schemata (plural of schema), which are patterns (or templates) that can be found within the chromosomes. Each schema represents a subset of chromosomes that have a certain similarity among them.

For example, if the set of chromosomes is represented by binary strings of length four, the schema 1\*01 represents all those chromosomes that have a 1 in the leftmost position, 01 in the rightmost two positions, and either a 1 or a 0 in the second from left position, since the \* represents a wildcard value.

For each schema, we can assign two measurements:

**Order**: The number of digits that are fixed (not wildcards)

**Defining length**: The distance between the two furthermost fixed digits

Each chromosome in the population corresponds to multiple schemata in the same way that a given string matches regular expressions. The chromosome 1101, for example, corresponds to each and every one of the schemata that appear in this table since it matches each of the patterns they represent. If this chromosome has a higher score, it is more likely to survive the selection operation, along with all the schemata it represents. As this chromosome gets crossed over with another, or as it gets mutated, some of the schemata will survive and others will disappear. The schemata of low order and short defining length are the ones more likely to survive.

Consequently, the schema theorem states that the frequency of schemata of low order, short defining length, and above-average fitness increases exponentially in successive generations. In other words, the smaller, simpler building blocks that represent the attributes that make a solution better will become increasingly present in the population as the genetic algorithm progresses.

**Differences from traditional algorithms**

There are several important differences between genetic algorithms and traditional search and optimization algorithms, such as gradient-based algorithms.

The key characteristics of genetic algorithms distinguishing them from traditional algorithms are:

* Maintaining a population of solutions
* Using a genetic representation of the solutions
* Utilizing the outcome of a fitness function
* Exhibiting a probabilistic behavior

**Population-based**

The genetic search is conducted over a population of candidate solutions (individuals) rather than a single candidate. At any point in the search, the algorithm retains a set of individuals that form the current generation. Each iteration of the genetic algorithm creates the next generation of individuals.

In contrast, most other search algorithms maintain a single solution and iteratively modify it in search of the best solution. The gradient descent algorithm, for example, iteratively moves the current solution in the direction of steepest descent, which is defined by the negative of the given function's gradient.

**Genetic representation**

Instead of operating directly on candidate solutions, genetic algorithms operate on their representations (or coding), often referred to as chromosomes. An example of a simple chromosome is a fixed-length binary string.

The chromosomes allow us to facilitate the genetic operations of crossover and mutation. Crossover is implemented by interchanging chromosome parts between two parents, while mutation is implemented by modifying parts of the chromosome.

A side effect of the use of genetic representation is decoupling the search from the original problem domain. Genetic algorithms are not aware of what the chromosomes represent and do not attempt to interpret them.

**Fitness function**

The fitness function represents the problem we would like to solve. The objective of genetic algorithms is to find the individuals that yield the highest score when this function is calculated for them.

Unlike many of the traditional search algorithms, genetic algorithms only consider the value that's obtained by the fitness function and do not rely on derivatives or any other information. This makes them suitable to handle functions that are hard or impossible to mathematically differentiate.

**Probabilistic behavior**

While many of the traditional algorithms are deterministic in nature, the rules that are used by genetic algorithms to advance from one generation to the next are probabilistic.

For example, when selecting the individuals that will be used to create the next generation, the probability of selecting a given individual increases with the individual's fitness, but there is still a random element in making that choice. Individuals with low score values can still be chosen as well, although with a lower probability.

Mutation is probability-driven as well, usually occurs with low likelihood, and makes changes at random location(s) in the chromosome.

The crossover operator can have a probabilistic element as well. In some variations of genetic algorithms, the crossover will only occur at a certain probability. If no crossover takes place, both parents are duplicated into the next generation without change.

Despite the probabilistic nature of this process, the genetic algorithm-based search is not random; instead, it uses the random aspect to direct the search toward areas in the search space where there is a better chance to improve the results.

**Use cases of genetic algorithms**

Genetic algorithms are best suited for the following types of problems:

* **Problems with complex mathematical representation:** Since genetic algorithms only require the outcome of the fitness function, they can be used for problems with target functions that are hard or impossible to differentiate, problems with a large number of parameters, and problems with a mix of parameter types.
* **Problems with no mathematical representation:** Genetic algorithms don't require a mathematical representation of the problem as long as a score value can be obtained or a method is available to compare two solutions.
* **Problems involving a noisy environment:** Genetic algorithms are resilient to problems where data may not be consistent, such as data originating from sensor output or from human-based scoring.
* **Problems involving an environment that changes over time:** Genetic algorithms can respond to slow changes in the environment by continuously creating new generations that will adapt to the changes that occur.

On the other hand, when a problem has a known and specialized way of being solved, using an existing traditional or analytic method is likely to be a more efficient choice.

**Basic flow of a genetic algorithm**

The main stages of the basic genetic algorithm flow are shown in the flowchart on the next page.

***Creating the initial population***

The initial population is a set of valid candidate solutions (individuals) chosen randomly. Since genetic algorithms use a chromosome to represent each individual, the initial population is actually a set of chromosomes. These chromosomes should conform to the chromosome format that we chose for the problem at hand, for example, binary strings of a certain length.

***Calculating the fitness***

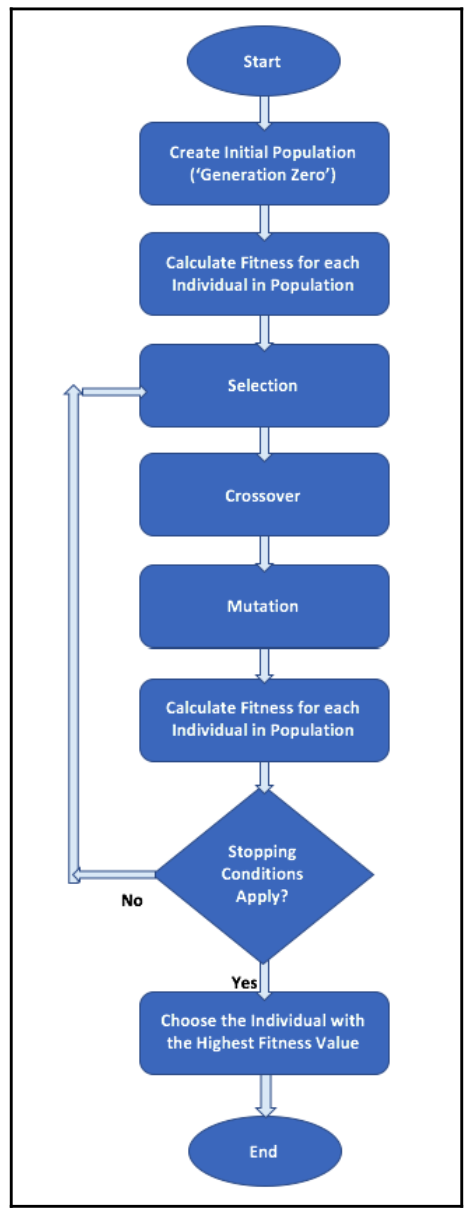
The value of the fitness function is calculated for each individual. This is done once for the initial population, and then for every new generation after applying the genetic operators of selection, crossover, and mutation. As the fitness of each individual is independent of the others, this calculation can be done concurrently.

Since the selection stage that follows the fitness calculation usually considers individuals with higher fitness scores to be better solutions, genetic algorithms are naturally geared toward finding the maximum value(s) of the fitness function. If we have a problem where the minimum value is desired, the fitness calculation should inverse the original value, for example, through multiplying it by a value of (-1).

***Applying selection, crossover, and mutation***

Applying the genetic operators of selection, crossover, and mutation to the population results in the creation of a new generation that is based on better individuals than the current ones.

The **selection** operator is responsible for selecting individuals from the current population in a way that gives an advantage to better individuals.

The **crossover** (or **recombination**) operator creates offspring from the selected individuals. This is usually done by taking two selected individuals at a time and interchanging parts of their chromosomes to create two new chromosomes representing the offspring. 

The **mutation** operator can randomly introduce a change to one or more of the chromosome values (genes) of each newly created individual. The mutation usually occurs with a very low probability.

***Checking the stopping conditions***

There can be multiple conditions to check against when determining whether the process can stop. The two most commonly used stopping conditions are:

* A maximum number of generations has been reached. This also serves to limit the runtime and computing resources consumed by the algorithm.
* There was no noticeable improvement over the last few generations. This can be implemented by storing the best fitness value achieved at every generation, and comparing the current best value to the one achieved a predefined number of generations ago. If the difference is smaller than a certain threshold, the algorithm can stop.

Other stopping conditions can be:

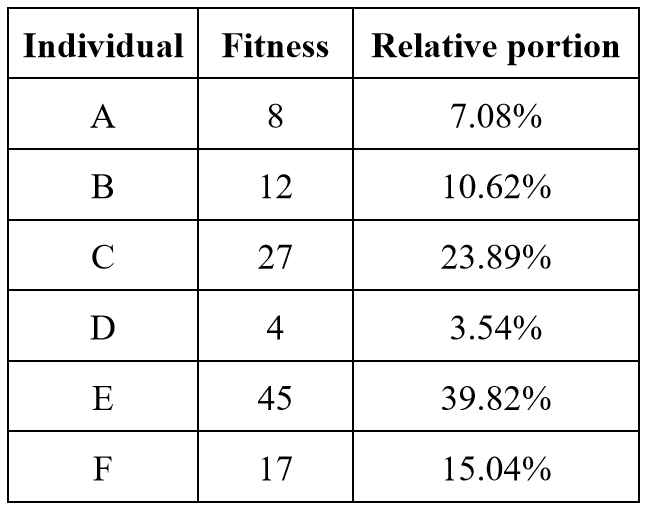
* A predetermined amount of time has elapsed since the process began.
* A certain cost or budget has been consumed, such as CPU time and/or memory.
* The best solution has taken over a portion of the population that is larger than a

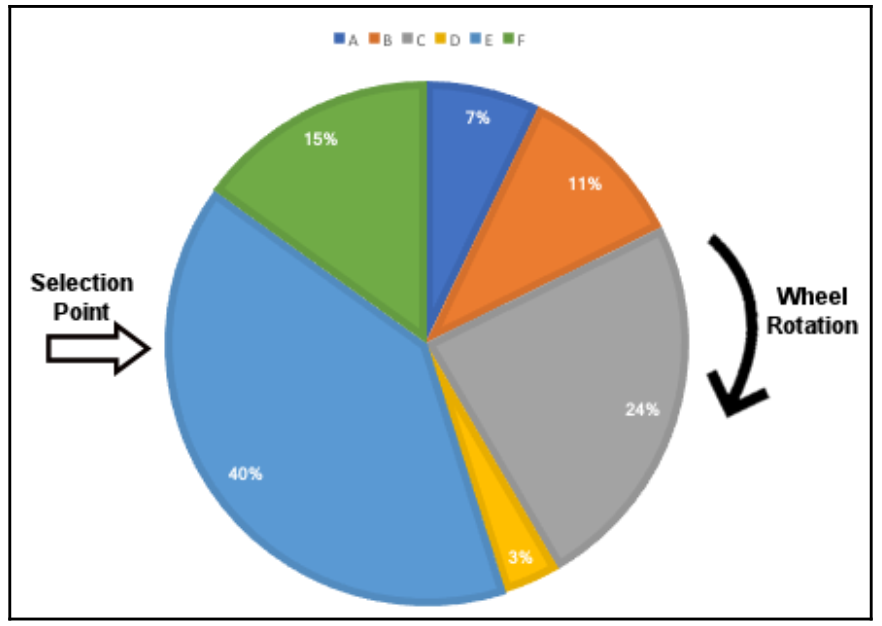
preset threshold.

To summarize, the genetic algorithm flow starts with a population of randomly generated candidate solutions (individuals), which are evaluated against the fitness function. The heart of the flow is a loop where the genetic operators of selection, crossover, and mutation are successively applied, followed by re-evaluation of the individuals. The loop continues until a stopping condition is encountered, upon which the best individual of the existing population is selected as our solution.

**Selection methods**

Selection is used at the beginning of each cycle of the genetic algorithm flow, to pick individuals from the current population that will be used as parents for the individuals of the next generation. The selection is probability-based, and the probability of an individual being picked is tied to its fitness value, in a way that gives an advantage to individuals with higher fitness values.

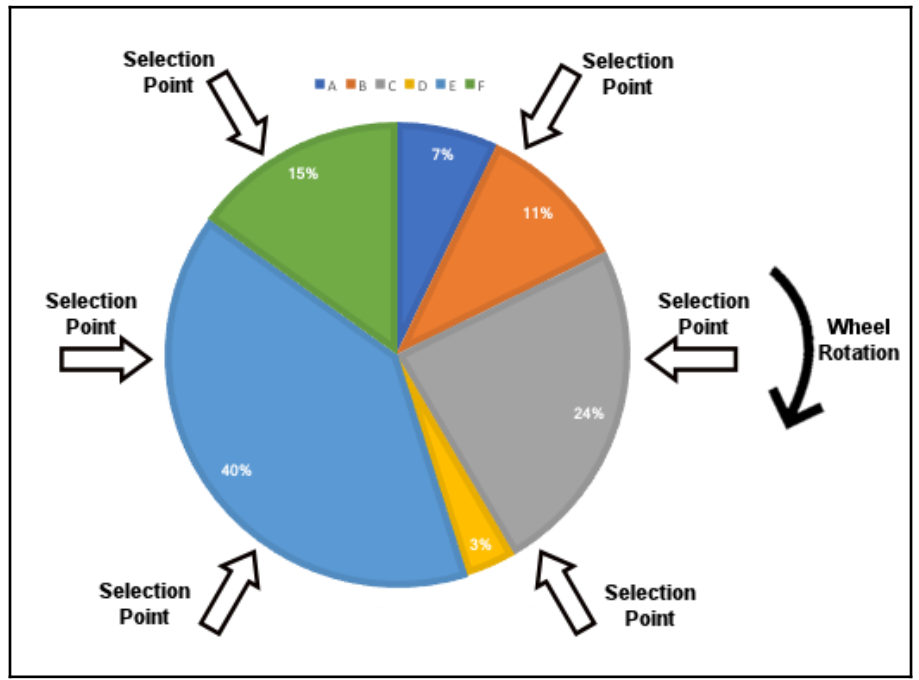
**Roulette wheel selection**

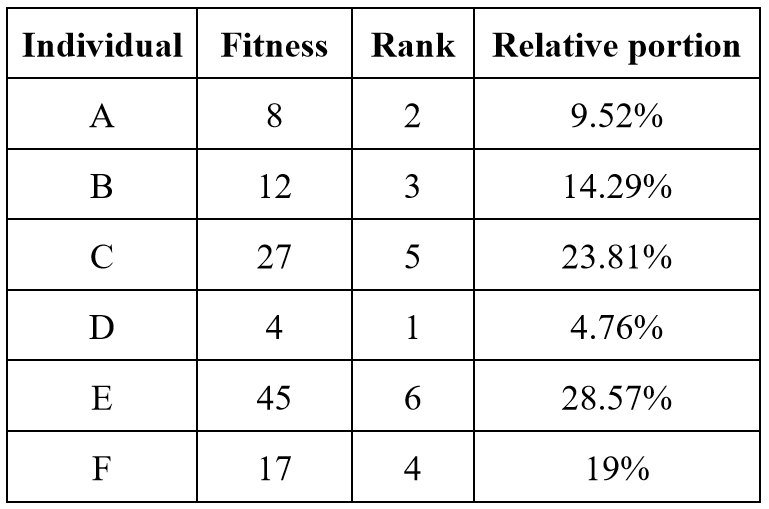
In the roulette wheel selection method, also known as fitness proportionate selection (FPS), the probability for selecting an individual is directly proportionate to its fitness value. This is comparable to using a roulette wheel in a casino and assigning each individual a portion of the wheel proportional to its fitness value. When the wheel is turned, the odds of each individual being selected are proportional to the size of the portion of the wheel that it occupies.

For example, suppose we have a population of six individuals with fitness values as shown in the table. The relative portion of the roulette wheel dedicated to each individual is calculated based on these fitness values.

Each time the wheel is turned, the selection point is used to choose a single individual from the entire population. The wheel is then turned again to select the next individual until we have enough individuals selected to fill the next generation. As a result, the same individual can be picked several times.

**Stochastic universal sampling**

Stochastic universal sampling (SUS) is a slightly modified version of the roulette wheel selection described previously. The same roulette wheel is used, with the same proportions, but instead of using a single selection point and turning the roulette wheel again and again until all needed individuals have been selected, we turn the wheel only once and use multiple selection points that are equally spaced around the wheel. This way, all the individuals are chosen at the same time, as depicted in the diagram. 

This selection method prevents individuals with particularly high fitness values from saturating the next generation by overly getting chosen over and over again. It thereby provides weaker individuals with a chance to be chosen, reducing the somewhat unfair nature of the original roulette wheel selection method.

**Rank-based selection**

The rank-based selection method is similar to the roulette wheel selection, but instead of directly using the fitness values to calculate the probabilities for selecting each individual, the fitness is used just to sort the individuals. Once sorted, each individual is given a rank representing its position, and the roulette probabilities are calculated based on these ranks.

For example, let's take the same population of six individuals we previously used with the same fitness values. To that, we add the rank of each individual. As the population size in our example is six, the highest-ranking individual gets the rank value of 6, the next one gets the rank value of 5, and so on. The relative portion of the roulette wheel dedicated to each individual is now calculated based on these rank values instead of using the fitness values.

Rank-based selection can be useful when a few individuals have much larger fitness values than all the rest. Using rank instead of raw fitness prevents these few individuals from taking over the entire population of the next generation, as ranking eliminates the large differences.

Another useful case is when all individuals have similar fitness values, where rank-based selection will spread them apart, giving a clearer advantage to the better ones even if the fitness differences are small.

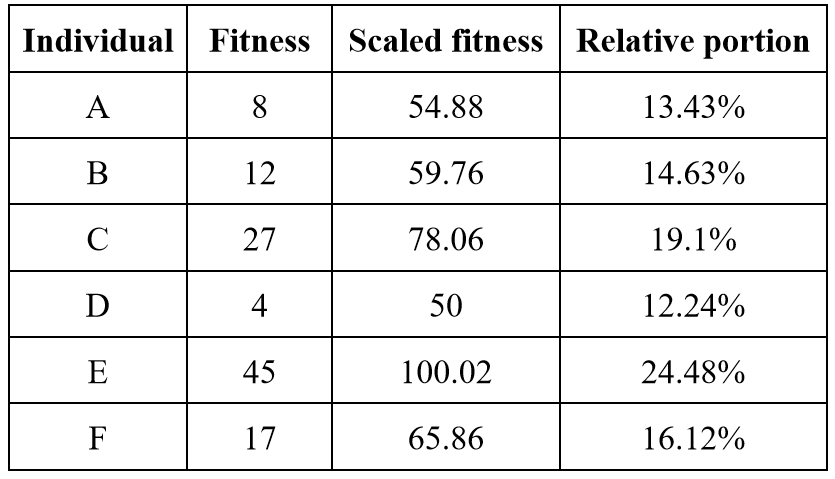
**Fitness scaling**

While rank-based selection replaces each fitness value with the individual's rank, fitness scaling applies a scaling transformation to the raw fitness values and replaces them with the transformation's result. The transformation maps the raw fitness values into a desired range, as follows: *scaled fitness* = ***a*** × (*raw fitness*) + ***b***

Here, ***a*** and ***b*** are constants that we can select to achieve the desired range of the scaled fitness. For example, if we use the same values from the previous examples, the range of the raw fitness values is between 4 (lowest fitness value, individual D) and 45 (highest fitness value, individual E). Suppose we want to map the values into a new range, between 50 and 100. We can calculate the values of the a and b constants using the following equations, representing these two individuals:

50 = ***a*** × 4 + ***b*** (lowest fitness value)

100 = ***a*** × 45 + ***b*** (highest fitness value)

Solving this simple system of linear equations will yield the following scaling parameter values: ***a*** = 1.22, ***b*** = 45.12.

This means that the scaled fitness values can be calculated as follows:

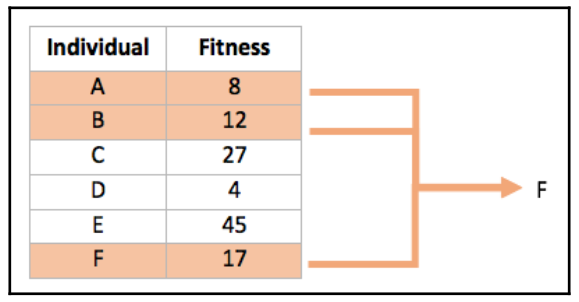
*scaled fitness* = 1.22 ×   
(*raw fitness*) + 45.12

After adding a new column to the table containing the scaled fitness values, we can see that the range is indeed between 0 and 50, as desired.

Scaling the fitness values to the new range provided a much more moderate partition of the roulette wheel compared to the original partition. The best individual (with a scaled fitness value of 100) is now only twice more likely to be selected than the worst one (with a scaled fitness value of 50), instead of being more than 11 times more likely to be chosen when using the raw fitness values.

**Tournament selection**

In each round of the tournament selection method, two or more individuals are randomly picked from the population, and the one with the highest fitness score wins and gets selected.

For example, suppose we have the same six individuals and the same fitness values we used in the previous examples. The diagram illustrates randomly selecting three of them (A, B, and F), then announcing F as the winner since it has the largest fitness value (17) among these three individuals. 

The number of individuals participating at each tournament selection round (three in our example) is suitably called tournament size. The larger the tournament size, the higher the chances that the best individuals will participate in the tournaments, and the lesser the chances of low-scoring individuals winning a tournament and getting selected.

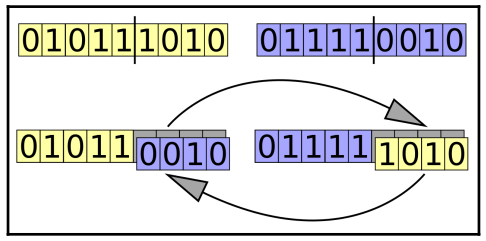
An interesting aspect of this selection method is that, as long as we can compare any two individuals and determine which of them is better, the actual value of the fitness function is not needed.

**Crossover methods**

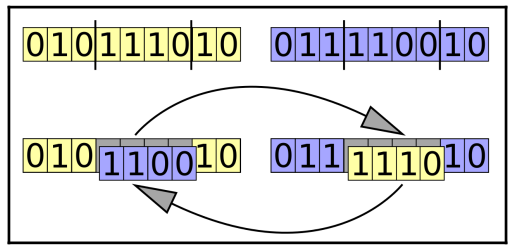
The crossover operator, also referred to as recombination, corresponds to the crossover that takes place during sexual reproduction in biology, and is used to combine the genetic information of two individuals, serving as parents, to produce (usually two) offspring.

The crossover operator is typically applied with some (high) probability value. Whenever crossover is not applied, both parents are directly cloned into the next generation.

**Single-point crossover**

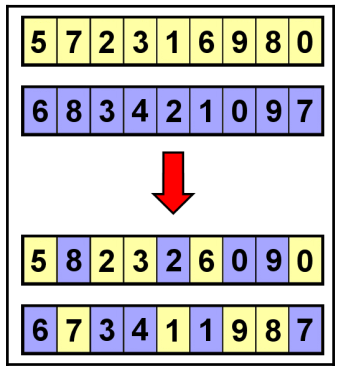
In the single-point crossover method, a location on the chromosomes of both parents is selected randomly. This location is referred to as the crossover point, or cut point. Genes to the right of that point are swapped between the two parent chromosomes. As a result, we get two offspring, where each of them carries some genetic information from both parents. The diagram demonstrates a single-point crossover operation conducted on a pair of binary chromosomes, with the crossover point located between the fifth and the sixth genes. Two-point and k-point crossover are the extensions of this method.

**Two-point and k-point crossover**

In the two-point crossover method, two crossover points on the chromosomes of both parents are selected randomly. The genes residing between these points are swapped between the two parent chromosomes.

The diagram demonstrates a two-point crossover carried out on a pair of binary chromosomes, with the first crossover point located between the third and fourth genes, and the other between the seventh and eighth genes.

The two-point crossover method can be implemented by carrying out two single-point crossovers, each with a different crossover point. A generalization of this method is the k-point crossover, where *k* represents a positive integer, and *k* crossover points are used.

**Uniform crossover**

In the uniform crossover method, each gene is independently determined by randomly choosing one of the parents. When the random distribution is 50%, each parent has the same chance of influencing the offspring.

In this example, the second offspring was created by complementing the choices made for the first offspring, however, both offspring can also be created independently of each other. Also, integer-based chromosomes were used, but it would work similarly with binary ones.

Since this method does not exchange entire segments of the chromosome, it has greater potential for diversity in the resulting offspring.

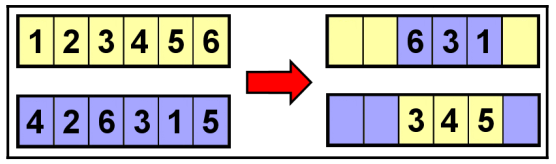
**Crossover for ordered lists**

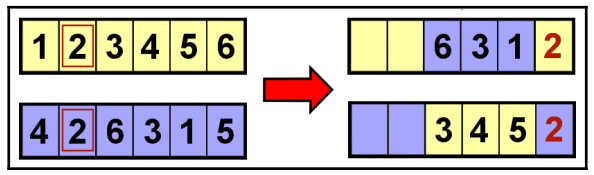
In the previous example, we saw the results of a crossover operation on two integer-based chromosomes. While each of the parents had every value between 0 and 9 appear exactly once, each of the resulting offspring had certain values appearing more than once (for example, 2 in the top offspring and 1 in the other), and other values were missing (such as 4 in the top offspring and 5 in the other).

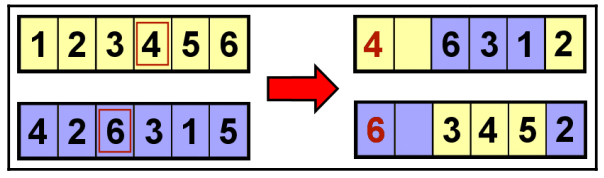
In some tasks, however, integer-based chromosomes may represent indices of an ordered list. For example, suppose we have several cities, we know the distance between each, and we need to find the shortest possible route through all of them. This is known as **the traveling salesman problem**.

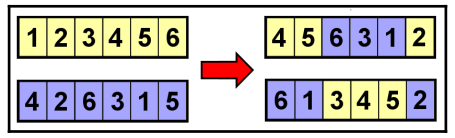
If, for instance, we have four cities, a convenient way to represent a possible solution for this problem would be a four-integer chromosome showing the order of visiting the cities, for example, (1,2,3,4) or (3,4,2,1). A chromosome having two of the same values, or missing one of the values such as (1,2,2,4), will not represent a valid solution.

For such cases, alternative crossover methods were devised to ensure that the offspring created will still be valid. Ordered crossover is one of these methods.

The **ordered crossover (OX1)** method strives to preserve the relative ordering of the parent's genes as much as possible. We will demonstrate it using chromosomes with a length of six.

The first step is a two-point crossover with random cut points, as shown in the diagram (with the parents depicted on the left side). We will now start filling in the rest of the genes in each offspring by going over all the parent's genes in their original order, starting after the second cut point. For the first parent, we find a 6, but this is already present in the offspring, so we continue (with wrapping around) to 1; this is already present too. The next in order is the 2. Since 2 is not yet present in the offspring, we add it there, as shown in the figure below. For the second parent-offspring pair, we start with the parent's 5, which is already present in the offspring, then move on to 4, which is present as well, and end up with the 2, which is not present yet and therefore gets added.

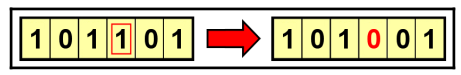
For the top parent, we now continue to 3 (already present in the offspring), and then 4, which gets added to the offspring. For the other parent, the next gene is 6. Since it's not present in the matching offspring, it gets added to it.

We continue in a similar fashion with the next genes not yet present in the offspring, and fill in the last available spots. This completes the process of producing two valid offspring chromosomes.

**Mutation methods**

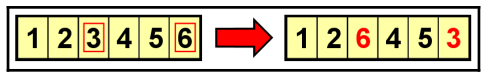
The mutation is the last genetic operator to be applied in the process of creating a new generation. The mutation operator is applied to the offspring that were created as a result of the selection and crossover operations.

The mutation is probability-based and usually occurs at a (very) low probability as it carries the risk of harming the performance of any individual it is applied to. In some versions of genetic algorithms, the mutation probability gradually increases as the generations advance to prevent stagnation and ensure diversity of the population. On the other hand, if the mutation rate is excessively increased, the genetic algorithm will turn into the equivalent of a random search.

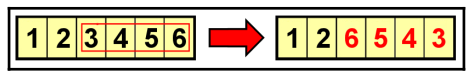
**Flip bit mutation**

When applying the flip bit mutation to a binary chromosome, one gene is randomly selected and its value is flipped (complemented). This can be extended to several random genes being flipped instead of just one.

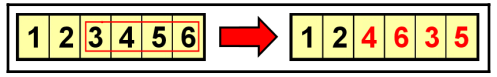
**Swap mutation**

When applying the swap mutation to binary or integer-based chromosomes, two genes are randomly selected and their values are swapped. This mutation operation is suitable for the chromosomes of ordered lists, as the new chromosome still carries the same genes as the original one.

**Inversion mutation**

When applying the inversion mutation to binary or integer-based chromosomes, a random sequence of genes is selected and the order of the genes in that sequence is reversed. Similar to the swap mutation, the inversion mutation operation is suitable for the chromosomes of ordered lists.

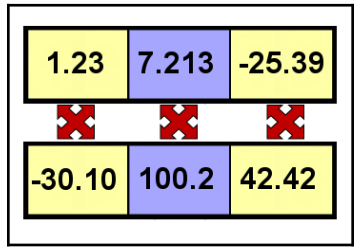
**Scramble mutation**

Another mutation operator suitable for the chromosomes of ordered lists is the scramble mutation. When applied, a random sequence of genes is selected and the order of the genes in that sequence is shuffled (or scrambled).

**Real-coded genetic algorithms**

So far, we have seen chromosomes that represented binary or integer parameters. Consequently, the genetic operators were suitable for working on these types of chromosomes. However, we often encounter problems where the solution space is continuous. In other words, the individuals are made up of real (floating-point) numbers. Instead of using binary strings, arrays of real-valued numbers were found to be a simpler and better approach. For example, if we have a problem involving three real-valued parameters, the chromosome will look like [x1, x2, x3], where x1, x2, x3 represent real numbers, such as [1.23, 7.2134, -25.309] or [-30.10, 100.2, 42.424].

The various selection methods mentioned earlier will work just the same for real-coded chromosomes as they only depend on the fitness of the individuals and not their representation.

However, the crossover and mutation methods mentioned earlier will not be suitable for the real-coded chromosomes and so specialized ones need to be used. One important point to remember is that these crossover and mutation operations are applied separately for each dimension of the array that forms the real-coded chromosome. For example, if [1.23, 7.213, -25.39] and [-30.10, 100.2, 42.42] are parents selected for the crossover operation, the crossover will be separately done for the pairs on the right. Similarly, the mutation operator, when applied to a real-coded chromosome, will apply separately to each dimension.

**Elitism**

While the average fitness of the genetic algorithm population generally increases as generations go by, it is possible at any point that the best individual(s) of the current generation will be lost. This is due to the selection, crossover, and mutation operators altering the individuals in the process of creating the next generation. In many cases, the loss is temporary as these individuals (or better individuals) will be re-introduced into the population in a future generation.

However, if we want to guarantee that the best individual(s) always make it to the next generation, we can apply the optional elitism strategy. This means that the top *n* individuals (*n* being a small, predefined parameter) are duplicated into the next generation before we fill the rest of the available spots with offspring that are created using selection, crossover, and mutation. The elite individuals that were duplicated are still eligible for the selection process so they can still be used as the parents of new individuals.

Elitism can sometimes have a significant positive impact on the algorithm's performance as it avoids the potential time waste needed for re-discovering good solutions that were lost in the genetic flow.

**The art of solving problems using genetic algorithms**

Genetic algorithms provide us with a powerful and versatile tool that can be used to solve a wide array of problems and tasks. When we set to work on a new problem, we need to customize the tool and match it to that problem. This is done by making several choices, as described in the following paragraphs.

First, we need to **determine the fitness function**. This is how each individual will be evaluated, where larger values represent better individuals. The function does not have to be mathematical. It can be represented by an algorithm, or a call to an external service, or even a result of a game played, to list a few options. We just need a way to programmatically retrieve the fitness value for any given proposed solution (individual).

Next, we need to **choose an appropriate chromosome encoding**. This is based on the parameters we send to the fitness function. So far, we have seen binary, integer, an ordered list, and real-coded examples. However, for some problems, we may need a mix of parameter types, or may even decide to create our own custom chromosome encoding.

Next, we need to **pick a selection method**. Most selection methods will work for any kind of chromosome type. If the fitness function is not directly accessible, but we still have a way to tell which of several candidate solutions is the best, we can consider utilizing the tournament selection method.

As was stated earlier, the **choice of crossover and mutation operators** will be linked to the chromosome encoding of the individuals. Binary-coded chromosomes will have different crossover and mutation schemes than those that fit real-coded problems. Similar to the choice of chromosome encoding, here, too, you can design your own methods of crossover and mutation to fit your unique use case.

Lastly, there are the hyperparameters of the algorithm. The most common parameter values we need to set are as follows:

* Population size;
* Crossover rate;
* Mutation rate;
* Max number of generations;
* Elitism (used or not; what size);
* Other stopping condition(s).

For these parameters, we can choose what we deem as reasonable values and then tweak them, similar to how hyperparameters are dealt with in almost any other optimization and learning algorithm.

**Using the DEAP Framework**

DEAP (short for Distributed Evolutionary Algorithms in Python) is a Python framework that supports the rapid development of solutions using genetic algorithms, as well as other evolutionary computation techniques. DEAP offers various data structures and tools that prove essential when implementing a wide range of genetic algorithm-based solutions. DEAP was developed at the Canadian Laval University in 2009 and is available under the GNU Lesser General Public License (LGPL). The source code for DEAP is available at https:/​/​github.​com/​DEAP/​deap. We will be using various Python 3 packages including the following: NumPy, Matplotlib, Seaborn (https:/​/​seaborn.​pydata.​org/)

**The OneMax problem**

The OneMax problem is a simple optimization task that is often used as the Hello World of genetic algorithm frameworks. This problem will be used to demonstrate how DEAP can be used to implement a genetic algorithm.

The OneMax task is to find the binary string of a given length that maximizes the sum of its digits. For example, the OneMax problem of length 5 will consider candidates such as the following:

10010 (sum of digits = 2)

01110 (sum of digits = 3)

11111 (sum of digits = 5)

Obviously, the solution to this problem is always the string that comprises all 1s. But the genetic algorithm does not have this knowledge, and needs to blindly look for this solution using its genetic operators. If the algorithm does its job, it will find this solution, or at least one close to it, within a reasonable amount of time. The steps are as follows:

**Choosing the chromosome**

Since the OneMax problem deals with binary strings, the choice of chromosomeis easy—each individual will be represented with a binary string that directly represents a candidate solution. In the actual Python implementation, this will be implemented as a list containing integer values of either 0 or 1. The length of the chromosome matches the size of the OneMax problem. For example, for a OneMax problem of size 5, the 10010 individual will be represented by the list [1, 0, 0, 1, 0].

**Calculating the fitness**

Since we want to find the individual with the largest sum of digits, we are going to use the FitnessMax strategy. And as each individual is represented by a list of integer values of either 0 or 1, the fitness value will be directly calculated as the sum of the elements in the list, for example: sum([1, 0, 0, 1, 0]) = 2.

**Choosing the genetic operators**

We now need to decide on the genetic operators to be used—selection, crossover, and mutation. In the previous chapter, we examined several different types of each of these operators. Choosing the genetic operators is not an exact science, and we can usually experiment with several choices. But while selection operators can typically work with any chromosome type, the crossover and mutation operators we choose need to match the chromosome type we use, otherwise they could produce invalid chromosomes.

For the selection operator, we can start with Tournament selection because it is simple and efficient. We can later experiment with other selection strategies, such as roulette wheel selection and stochastic universal sampling.

For crossover operator, either single-point or two-point crossover operators will be suitable, as the result of crossing over two binary strings using these methods will produce a valid binary string.

The mutation operator can be the simple flip bit mutation that works well for binary strings.

**Setting the stopping condition**

It is always a good idea to put a limit on the number of generations, to guarantee that the algorithm does not run forever. This gives us one stopping condition.

In addition, since we happen to know the best solution for the OneMax problem—a binary string with all 1s, and a fitness value equal to the length of the individual—we can use that as a second stopping condition. For a real-world problem, we typically don't have this kind of knowledge in advance.

If either of these conditions is met, that is, the number of generations reaches the limit or the best solution is found, the genetic algorithm will stop.

The complete program can be found here:

<https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter03/01-OneMax-long.py>

**Particle swarm optimization**

Particle swarm optimization (PSO) draws its inspiration from natural groupings of individual organisms, such as flocks of birds or schools of fish, generally referred to as swarms. The organisms interact within the swarm without central supervision, working together toward a common goal. This observed behavior gave rise to a computational method that can solve or optimize a given problem by using a group of candidate solutions represented by particles analogous to organisms in a swarm. The particles move in the search space, looking for the best solution, and their movement is governed by simple rules that involve their position and velocity (directional speed).

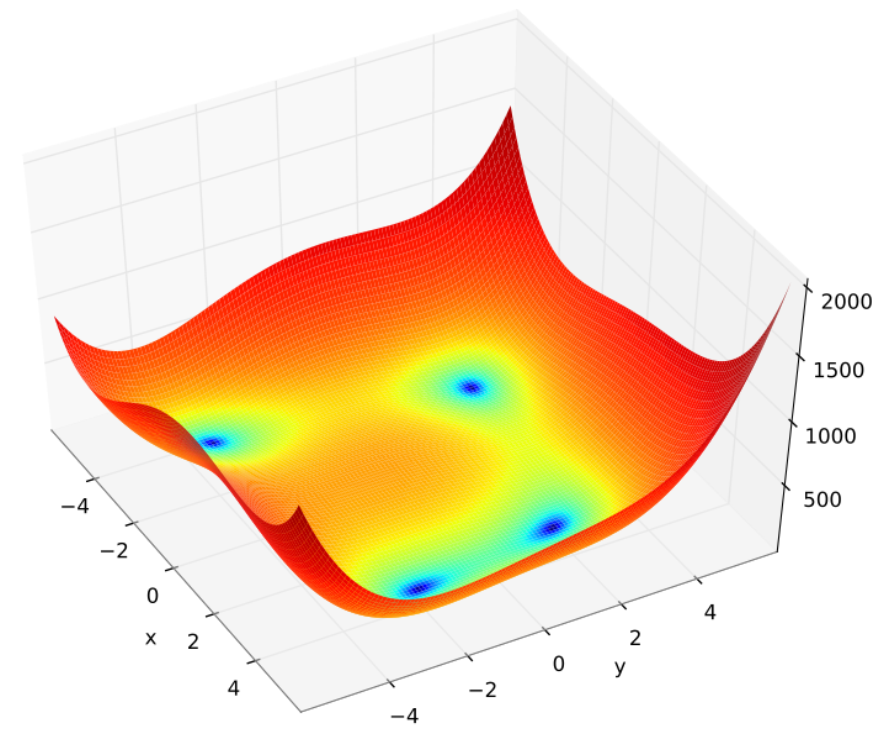
The PSO algorithm is iterative, and in each iteration, every particle's position gets evaluated, and its best location so far, as well as the best location within the entire group of particles, are updated if necessary. Then, each particle's velocity is updated according to the following information:

* The particle's current speed and direction of movement – representing inertia.
* The particle's best position found so far (local best) – representing cognitive force.
* The entire group's best position found so far (global best) – representing social force.

This is followed by an update to the particle's position, based on the newly calculated velocity.

The iterative process continues until some stopping condition, such as the iterations limit, is met. At this point, the group's current best position is taken as the solution by the algorithm.

For demonstration purposes, the PSO algorithm will be used to find the minimum location(s) of Himmelblau's function, a commonly used benchmark. The graphic representation of Himmelblau's function:



This function can be mathematically expressed as follows:

*f*(*x*, *y*) = (*x*2 + *y* - 11)2 + (*x* + *y*2 - 7)2

It has four global minima, evaluating to 0, indicated by the blue areas in the plot. These are located at the following coordinates:

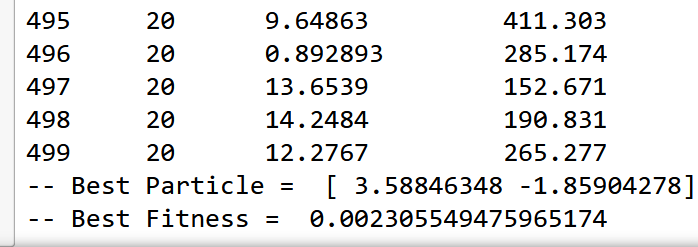
* x = 3.0, y = 2.0
* x = -2.805118, y = 3.131312
* x = -3.779310, y = -3.283186
* x = 3.584458, y = -1.848126

**Particle swarm optimization implementation**

A Python program to locate a minimum of Himmelblau's function using particle swarm optimization, is located at

<https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter12/03-pso-himmelblau.py>

Setting the RANDOM\_SEED=33 we get the following results:



These results indicate that the algorithm was able to locate one of the minima, around x=3.59 and y=-1.86. It is evident that the particles move around quite a bit and, during the run, oscillate closer to the best result and away from it.

To find the other minima locations, the algorithm can be rerun with a different random seed. You can also penalize the solutions in the areas around the previously found minima. Another approach could be using multiple simultaneous swarms to locate several minima in the same run.

**Other related techniques**

**Evolution strategies**

Evolution strategies (ES) are a kind of genetic algorithm that emphasizes mutation rather than crossover as the evolutionary facilitator. The mutation is adaptive, and its strength is learned over the generations. The selection operator in evolution strategy is always rank-based rather than done using actual fitness values. A simple version of this technique is called (1 + 1). It includes only two individuals – a parent and its mutated offspring. The best of them continues to be the parent of the next mutated offspring. In the more general case, called (1 + λ), there is one parent and λ mutated offspring, and the best of the offspring continues to be the parent of the next λ offspring. Some newer variations of the algorithm include more than one parent, as well as a crossover operator.

**Differential evolution**

Differential evolution (DE) is a specialized variant of genetic algorithms that's used for the optimization of real-valued functions. DE differs from genetic algorithms in the following aspects:

* The DE population is always represented as a collection of real-valued vectors.
* Instead of replacing the entire current generation with a new generation, DE keeps iterating over the population, modifying one individual at a time, or keeping the original individual if it's better than its modified version.
* The traditional crossover and mutation operators are replaced by specialized ones, thereby modifying the value of the current individual using the values of three other individuals that are chosen at random.

**Ant colony optimization**

Ant colony optimization (ACO) algorithms are inspired by the way certain species of ants locate food. The ants start by wandering randomly, and when any of them locates food, they go back to their colony while depositing pheromones along the way, marking the path for other ants. Other ants finding food at the same location will reinforce the trail by depositing their own pheromones. The pheromone marks fade away over time, giving the shorter paths and the paths that are traveled more often an advantage.

ACO algorithms use artificial ants that move about in the search space looking for the location of the best solutions. The ants keep track of their locations and the candidate solutions they have found along the way. This information is used by the ants of the subsequent iterations so that they can find better solutions. These algorithms are often combined with the local search method, which is activated after locating an area of interest.

**Artificial immune systems**

Artificial immune systems (AIS) draw their inspiration from the characteristics of adaptive immune systems found in mammals. These systems are capable of identifying and learning new threats, as well as applying the acquired knowledge and responding faster the next time a similar threat is detected.

Recent AIS can be used in various machine learning and optimization tasks, and generally belong to one of the following three subfields:

* **Clonal selection.** Imitating the process by which the immune system selects the best cell to recognize and eliminate an antigen that enters the body. The cell is chosen out of a pool of preexisting cells with varying specificities, and once chosen, it is cloned to create a population of cells that eliminates the invading antigen. This paradigm is typically applied to optimization and pattern recognition tasks.
* **Negative selection.** This follows a process that identifies and deletes cells that may attack self-tissues. These algorithms are typically used in anomaly detection tasks, where normal patterns are used to "negatively" train filters that will then be able to detect anomalous patterns.
* **Immune network algorithms.** This is inspired by the theory that suggests that the immune system is regulated using special types of antibodies that bind to other antibodies. In this type of algorithm, antibodies represent nodes in a network and the learning process involves the creation or removal of edges between the nodes, resulting in an evolving network graph structure. These algorithms are typically used in non-supervised machine learning tasks, as well as in the fields of control and optimization.

**Artificial life**

Rather than being a branch of evolutionary computation, artificial life (ALife) is a broader field that involves systems and processes that imitate natural life in different ways, such as computer simulations and robotic systems.

Evolutionary computation can actually be viewed as an application of ALife, where the population seeking to optimize a certain fitness function is a metaphor for organisms searching for food. The main branches of ALife are as follows:

* **Soft:** represents software-based (digital) simulation.
* **Hard:** represents hardware-based (physical) robotics.
* **Wet:** represents biochemical-based manipulation or synthetic biology.

ALife can also be viewed as the bottom-up counterpart to artificial intelligence since ALife typically builds on the biological environment, mechanisms, and structures rather than high-level cognition.