1
An Introduction to Genetic Algorithms

Drawing its inspiration from Charles Darwin's theory of natural evolution, one of the most fascinating techniques for problem-solving is the algorithm family suitably named **evolutionary computation**. Within this family, the most prominent and widely used branch is known as **genetic algorithms**. This chapter is the beginning of your journey to mastering this extremely powerful, yet extremely simple, technique.

In this chapter, we will introduce **genetic algorithms** and their analogy to Darwinian evolution, and dive into their basic principles of operation as well as their underlying theory. We will then go over the differences between genetic algorithms and traditional ones and cover the advantages and limitations of genetic algorithms and their uses. We will conclude by reviewing the cases where the use of a genetic algorithm may prove beneficial.

In this introductory chapter, we will cover the following topics:

- What are genetic algorithms?
- The theory behind genetic algorithms
- Differences between genetic algorithms and traditional algorithms
- Advantages and limitations of genetic algorithms
- When to use genetic algorithms

What are genetic algorithms?

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.

In the rest of this section, we will review the basic ideas of genetic algorithms, as well as their analogy to the evolutionary processes transpiring in nature.

Darwinian evolution

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.

In other words, evolution maintains a population of individual specimens that vary from each other. Those who are better adapted to their environment have a greater chance of surviving, breeding, and passing their traits to the next generation. This way, as generations go by, species become more adapted to their environment and to the challenges presented to them.

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.

The genetic algorithms analogy

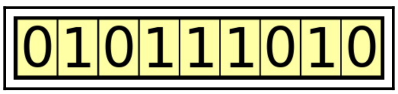
Genetic algorithms seek to find the optimal solution for a given problem. Whereas Darwinian evolution maintains a population of individual specimens, genetic algorithms maintain a population of candidate solutions, called **individuals**, for that given problem. These candidate solutions are iteratively 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 the following sections, we will describe the various components of genetic algorithms that enable this analogy for Darwinian evolution.

Genotype

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**:

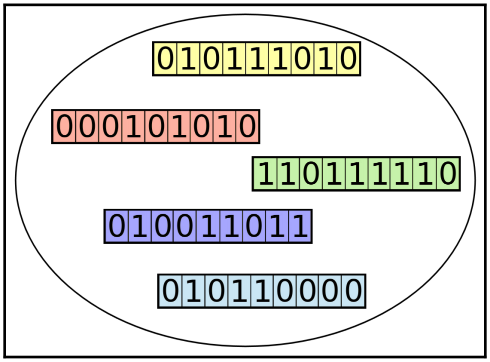


Simple binary-coded chromosome

The preceding image shows an example of one such binary-coded chromosome, representing one particular individual.

Population

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:



A population of individuals represented by binary-coded chromosomes

The population continually represents the current generation and evolves over time when the current generation is replaced by a new one.

Fitness function

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.

Selection

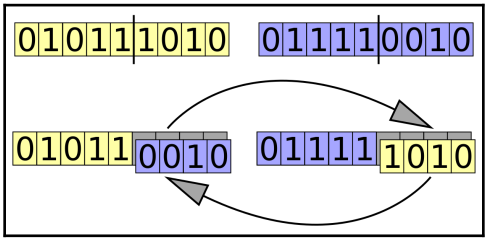
After calculating the fitness of every individual in the population, a selection process is used to determine which of the individuals in the population will get to reproduce and create the offspring that will form the next generation.

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.

Crossover

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:



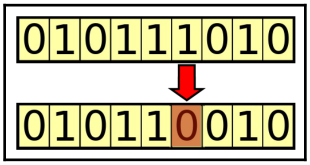
Crossover operation between two binary-coded chromosomes
Source: https://commons.wikimedia.org/wiki/File:Computational.science.Genetic.algorithm.Crossover.One.Point.svg.Image by Yearofthedragon. Licensed under Creative Commons CC BY-SA 3.0: https://creativecommons.org/licenses/by-sa/3.0/deed.en

The preceding image illustrates a simple crossover operation of creating two offspring from two parents.

Mutation

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:



Mutation operator applied to a binary-coded chromosome

The preceding image shows an example of the mutation operation.

Now, let's look at the theory behind genetic algorithms.

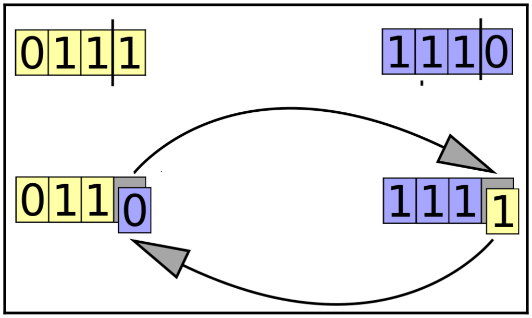
The theory behind genetic algorithms

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. This is illustrated in the following image:



Demonstration of a crossover operation bringing the building blocks of the optimal solution together

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

The following table provides several examples of four-digit binary schemata and their measurements:

Schema	Order	Defining Length
1101	4	3
1*01	3	3
*101	3	2
*1*1	2	2
**01	2	1
1***	1	0
****	0	0

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.

Consequentially, 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. We will look at the difference between genetic and traditional algorithms in the next section.

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

In the upcoming sections, we will describe these factors in greater detail.

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. Now, let's look at the advantages of genetic algorithms.

Advantages of genetic algorithms

The unique characteristics of genetic algorithms that we discussed in the previous sections provide several advantages over traditional search algorithms.

The main advantages of genetic algorithms are as follows:

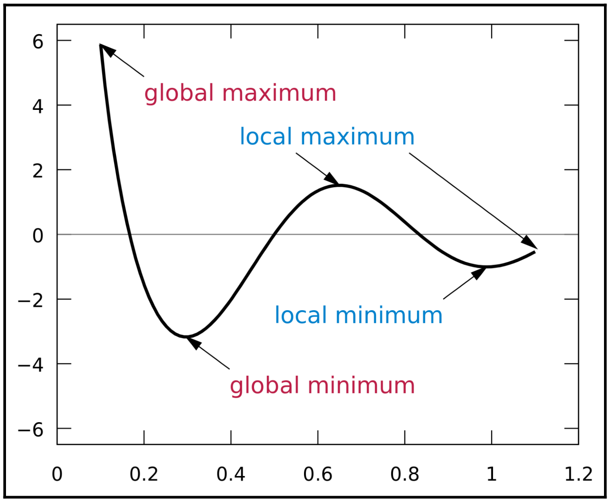
- Global optimization capability
- Handling problems with a complex mathematical representation
- Handling problems that lack mathematical representation
- Resilience to noise
- Support for parallelism and distributed processing
- Suitability for continuous learning

We will cover each of these in the upcoming sections.

Global optimization

In many cases, optimization problems have local maxima and minima points; these represent solutions that are better than those around them, but not the best overall.

The following image illustrates the differences between global and local maxima and minima points:



The global and local maxima and minima of a function
Source: https://commons.wikimedia.org/wiki/File:Computational.science.Genetic.algorithm.Crossover.One.Point.svg.

Most traditional search and optimization algorithms, and particularly those that are gradient-based, are prone to getting stuck in a local maximum rather than finding the global one. This is because, in the vicinity of a local maximum, any small change will degrade the score.

Genetic algorithms, on the other hand, are less sensitive to this phenomenon and are more likely to find the global maximum. This is due to the use of a population of candidate solutions rather than a single one, and the crossover and mutation operations that will, in many cases, result in candidate solutions that are distant from the previous ones. This holds true as long as we manage to maintain the diversity of the population and avoid **premature convergence**, as we will mention in the next section.

Handling complex problems

Since genetic algorithms only require the outcome of the fitness function for each individual and are not concerned with other aspects of the fitness function such as derivatives, they can be used for problems with complex mathematical representations or functions that are hard or impossible to differentiate.

Other complex cases where genetic algorithms excel include problems with a large number of parameters and problems with a mix of parameter types; for example, a combination of continuous and discrete parameters.

Handling a lack of mathematical representation

Genetic algorithms can be used for problems that lack mathematical representation altogether. One such case of particular interest is when the fitness score is based on human opinion. Imagine, for example, that we want to find the most attractive color palette to be used on a website. We can try different color combinations and ask users to rate the attractiveness of the site. We can apply genetic algorithms to search for the best scoring combination while using this opinion-based score as the fitness function outcome. The genetic algorithm will operate as usual, even though the fitness function lacks any mathematical representation and there is no way to calculate the score directly from a given color combination.

As we will see in the next chapter, genetic algorithms can even deal with cases where the score of each individual cannot be obtained, as long as we have a way to compare two individuals and determine which of them is better. An example of this is a machine learning algorithm that drives a car in a simulated race. A genetic algorithm-based search can optimize and tune the machine learning algorithm by having different versions of it compete against each other to determine which version is better.

Resilience to noise

Some problems present **noisy** behavior. This means that, even for similar input parameter values, the output value may be somewhat different every time it's measured. This can happen, for example, when the data that's being used is being read from sensor outputs, or in cases where the score is based on human opinion, as was discussed in the previous section.

While this kind of behavior can throw off many traditional search algorithms, genetic algorithms are generally resilient to it, thanks to the repetitive operation of reassembling and reevaluating the individuals.

Parallelism

Genetic algorithms lend themselves well to parallelization and distributed processing. Fitness is independently calculated for each individual, which means all the individuals in the population can be evaluated concurrently.

In addition, the operations of selection, crossover, and mutation can each be performed concurrently on individuals and pairs of individuals in the population.

This makes the approach of genetic algorithms a natural candidate for distributed as well as cloud-based implementation.

Continuous learning

In nature, evolution never stops. As the environmental conditions change, the population will adapt to them. Similarly, genetic algorithms can operate continuously in an ever-changing environment, and at any point in time, the best current solution can be fetched and used.

For this to be effective, the changes in the environment need to be slow in relation to the generation turnaround rate of the genetic algorithm-based search. Now that we have covered the advantages, let's look at the limitations of genetic algorithms.

Limitations of genetic algorithms

To get the most out of genetic algorithms, we need to be aware of their limitations and potential pitfalls.

The limitations of genetic algorithms are as follows:

- The need for special definitions
- The need for hyperparameter tuning
- Computationally-intensive operations
- The risk of premature convergence
- No guaranteed solution

We will cover each of these in the upcoming sections.

Special definitions

When applying genetic algorithms to a given problem, we need to create a suitable representation for them – define the fitness function and the chromosome structure, as well as the selection, crossover, and mutation operators that will work for this problem. This can often prove to be challenging and time-consuming.

Luckily, genetic algorithms have already been applied to countless different types of problems, and many of these definitions have been standardized. This book covers numerous types of real-life problems and the way they can be solved using genetic algorithms. Use this as guidance whenever you are challenged by a new problem.

Hyperparameter tuning

The behavior of genetic algorithms is controlled by a set of hyperparameters, such as the population size and mutation rate. When applying genetic algorithms to the problem at hand, there are no exact rules for making these choices.

However, this is the case for virtually all search and optimization algorithms. After going over the examples in this book and doing some experimentation of your own, you will be able to make sensible choices for these values.

Computationally-intensive

Operating on (potentially large) populations and the repetitive nature of genetic algorithms can be computationally intensive, as well as time consuming before a good result is reached.

These can be alleviated with a good choice of hyperparameters, implementing parallel processing, and in some cases, caching the intermediate results.

Premature convergence

If the fitness of one individual is much higher than the rest of the population, it may be duplicated enough that it takes over the entire population. This can lead to the genetic algorithm getting prematurely stuck in a local maximum, instead of finding the global one.

To prevent this from occurring, it is important to maintain the diversity of the population. Various ways to maintain diversity will be discussed in the next chapter.

No guaranteed solution

The use of genetic algorithms does not guarantee that the global maximum for the problem at hand will be found.

However, this is almost the case for any search and optimization algorithm, unless it is an analytical solution for a particular type of problem.

Generally, genetic algorithms, when used appropriately, are known to provide good solutions within a reasonable amount of time. Now, let's look at a few use cases of genetic algorithms.

Use cases of genetic algorithms

Based on the material we covered in the previous sections, 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.

Summary

In this chapter, we started by introducing genetic algorithms, their analogy to Darwinian evolution, and their basic principles of operation, including the use of population, genotype, the fitness function, and the genetic operators of selection, crossover, and mutation.

Then, we covered the theory underlying genetic algorithms by going over the building-block hypothesis and the schema theorem and illustrating how genetic algorithms work by bringing together superior, small building blocks to create the best solutions.

Next, we went over the differences between genetic algorithms and traditional ones, such as maintaining a population of solutions and using a genetic representation of the solutions.

We continued by covering the strengths of genetic algorithms, including their capacity for global optimization, handling problems with complex or non-existent mathematical representations and resilience to noise, followed by their weaknesses, including the need for special definitions and hyperparameter tuning, as well as the risk of premature convergence.

We concluded by going over the cases where the use of a genetic algorithm may prove beneficial, such as mathematically complex problems and optimization tasks in a noisy or ever-changing environment.

In the next chapter, we will delve deeper into the key components and the implementation details of genetic algorithms in preparation for the following chapters, where we will use them to code solutions for various types of problems.

Further reading

For more information on what we covered in this chapter, please refer to *Introduction to genetic algorithms*, from the book *Hands-On Artificial Intelligence for IoT* by *Amita Kapoor*, January 2019, available at `https://subscription.packtpub.com/book/big_data_and_business_intelligence/9781788836067`.

2
Understanding the Key Components of Genetic Algorithms

In this chapter, we will dive deeper into the key components and the implementation details of genetic algorithms, in preparation for the following chapters, where we will use genetic algorithms to create solutions for various types of problems.

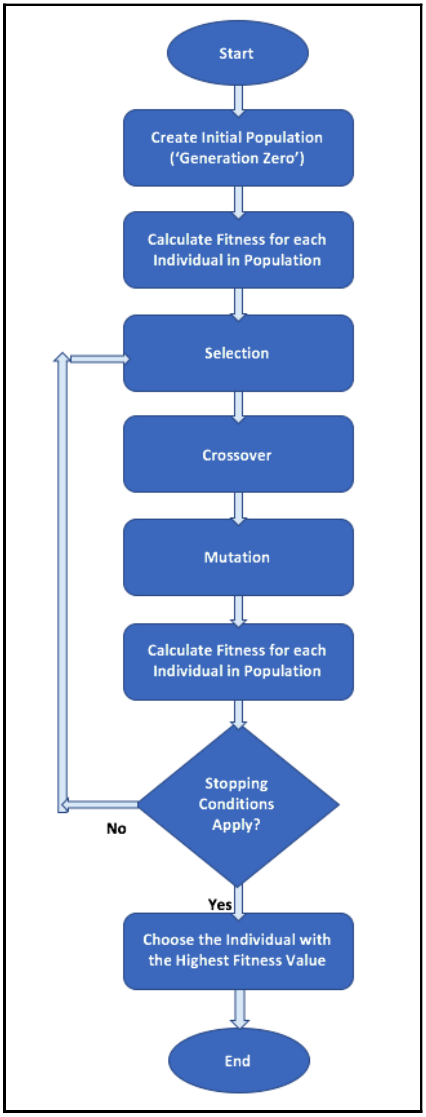
First, we will outline the basic flow of a genetic algorithm, then break it down into its different components while demonstrating various implementations of selection methods, crossover methods, and mutation methods. Next, we will look into real-coded genetic algorithms, which facilitate search in a continuous parameter space. This will be followed by an overview of the intriguing topics of elitism, niching, and sharing in genetic algorithms. Finally, we will study the art of solving problems using genetic algorithms.

At the end of this chapter, you will have achieved the following:

- Be familiar with the key components of genetic algorithms
- Understand the stages of the genetic algorithm flow
- Understand the genetic operators and become familiar with several of their variants
- Know the various options for stopping conditions
- Understand what modifications are needed to the basic genetic algorithm when applied to real numbers
- Understand the mechanism of elitism
- Understand the concepts and implementation of niching and sharing
- Know the choices you have to make when starting to work on a new problem

Basic flow of a genetic algorithm

The main stages of the basic genetic algorithm flow are shown in the following flowchart:



Basic flow of a genetic algorithm

These stages are described in detail in the following sections.

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. Examples of selection operators are given in the *Selection methods* section.

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. Examples of selection operators are given in the *Crossover methods* section.

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. Examples of mutation operators are given in the *Mutation methods* section.

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. Let's now look at the selection methods.

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.

The following sections describe some of the commonly used selection methods and their characteristics.

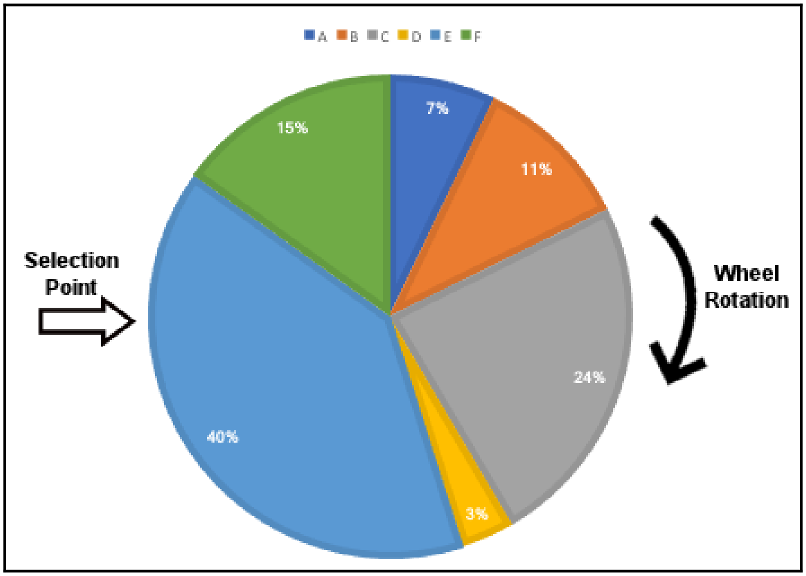
Roulette wheel selection

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 following table. The relative portion of the roulette wheel dedicated to each individual is calculated based on these fitness values:

Individual	Fitness	Relative portion
A	8	7%
B	12	11%
C	27	24%
D	4	3%
E	45	40%
F	17	15%

The matching roulette wheel is depicted in the following diagram:

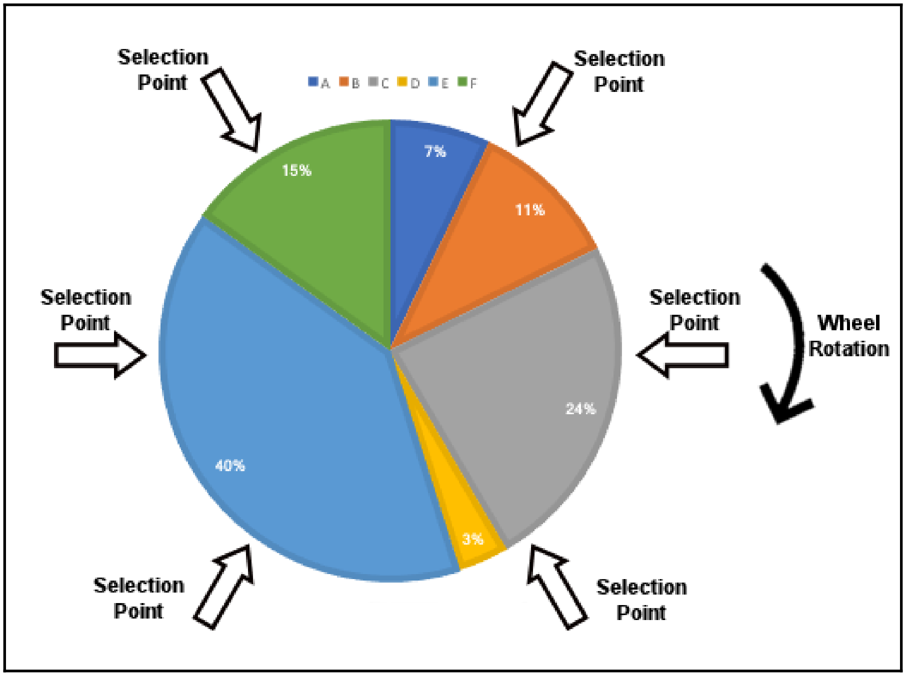


Roulette wheel selection example

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 following diagram:



Stochastic universal sampling example

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.

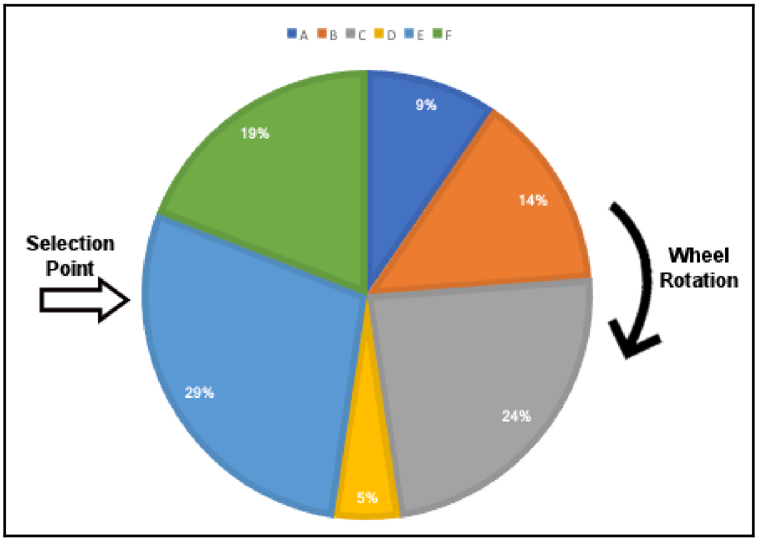
Rank-based selection

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:

Individual	Fitness	Rank	Relative portion
A	8	2	9%
B	12	3	14%
C	27	5	24%
D	4	1	5%
E	45	6	29%
F	17	4	19%

The matching roulette wheel is depicted in the following diagram:



Rank-based selection example

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:

$$\textit{scaled fitness} = \boldsymbol{a} \times (\textit{raw fitness}) + \boldsymbol{b}$$

Here, $\boldsymbol{a}$ and $\boldsymbol{b}$ 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 $\boldsymbol{a}$ and $\boldsymbol{b}$ constants using the following equations, representing these two individuals:

- $50 = \boldsymbol{a} \times 4 + \boldsymbol{b}$ (lowest fitness value)
- $100 = \boldsymbol{a} \times 45 + \boldsymbol{b}$ (highest fitness value)

Solving this simple system of linear equations will yield the following scaling parameter values:

$$\boldsymbol{a} = 1.22,\ \boldsymbol{b} = 45.12$$

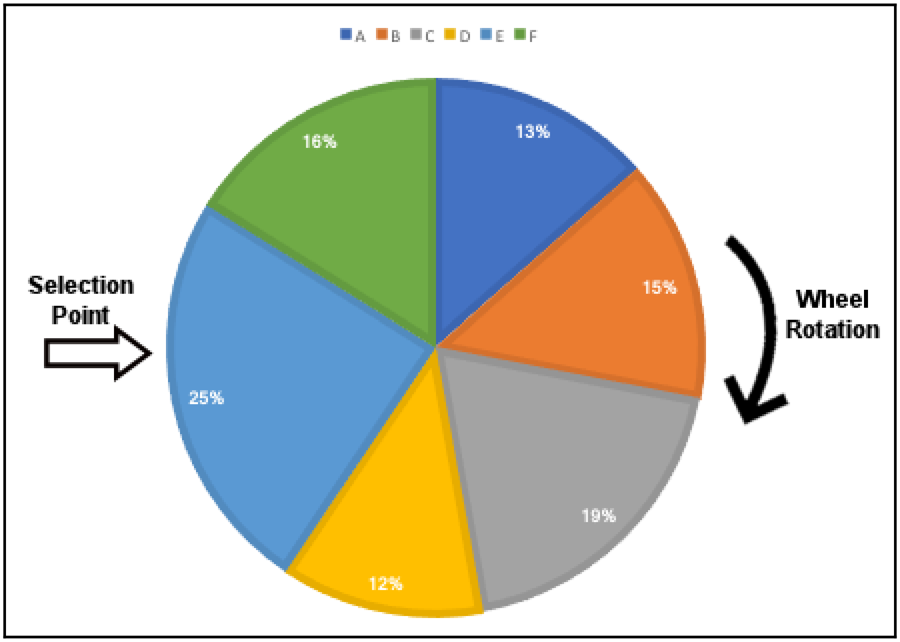
This means that the scaled fitness values can be calculated as follows:

$$\textit{scaled fitness} = \mathbf{1.22} \times (\textit{raw fitness}) + \mathbf{45.12}$$

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:

Individual	Fitness	Scaled fitness	Relative portion
A	8	55	13%
B	12	60	15%
C	27	78	19%
D	4	50	12%
E	45	100	25%
F	17	66	16%

The matching roulette wheel is depicted in the following diagram:



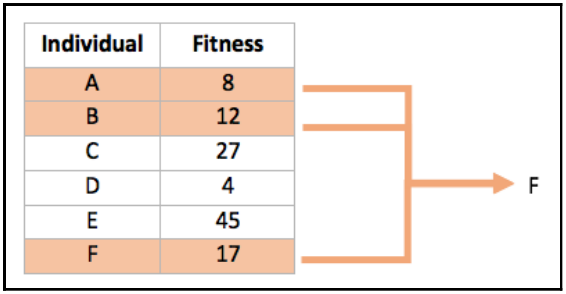
Roulette wheel selection example after fitness scaling

As the diagram illustrates, 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 following 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:



Tournament selection example with a tournament size of three

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. We will now look at the crossover methods.

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.

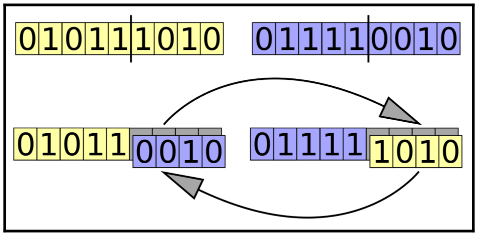
The crossover operator is typically applied with some (high) probability value. Whenever crossover is *not* applied, both parents are directly cloned into the next generation.

The following sections describe some of the commonly used crossover methods and their typical use cases. However, in certain situations, you may opt to use a problem-specific crossover method that will be more suitable for a particular case.

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 following 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:



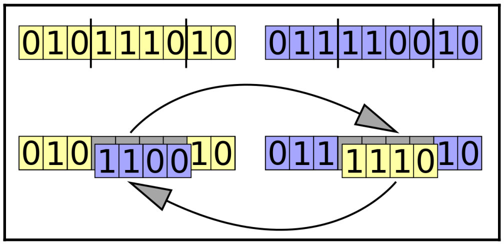
Single-point crossover example
Source: https://commons.wikimedia.org/wiki/File:Computational.science.Genetic.algorithm.Crossover.One.Point.svg. Image by Yearofthedragon. Licensed under Creative Commons CC BY-SA 3.0: https://creativecommons.org/licenses/by-sa/3.0/deed.en.

In the next section, we will cover extensions of this method, namely two-point and k-point crossover.

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 following 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:

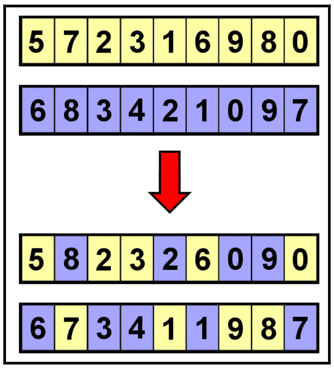


Two-point crossover example
Source: https://commons.wikimedia.org/wiki/File:Computational.science.Genetic.algorithm.Crossover.Two.Point.svg. Image by Yearofthedragon. Licensed under Creative Commons CC BY-SA 3.0: https://creativecommons.org/licenses/by-sa/3.0/deed.en.

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, as illustrated in the following diagram:



Uniform crossover example

Note that, 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.

> In this example, we used integer-based chromosomes, 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 and will be covered in detail in one of the following chapters.

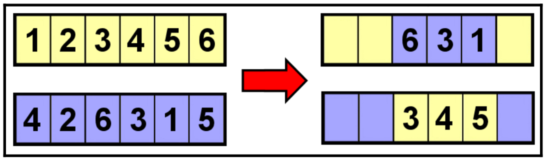
If, for instance, we have four cities, a convenient way to represent a possible solution for this problem would be a four-integer chromosome showing the order of visiting the cities, for example, (1,2,3,4) or (3,4,2,1). A chromosome having two of the same values, or missing one of the values such as (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. One of these methods, *Ordered crossover*, is covered in the following section.

Ordered crossover

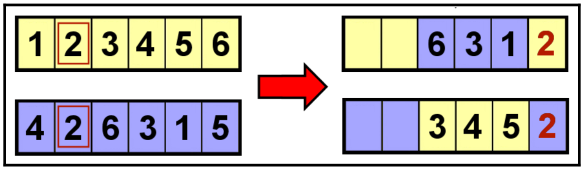
The **ordered crossover** (**OX1**) method strives to preserve the relative ordering of the parent's genes as much as possible. We will demonstrate it using chromosomes with a length of six.

The first step is a two-point crossover with random cut points, as shown in the following diagram (with the parents depicted on the left side):



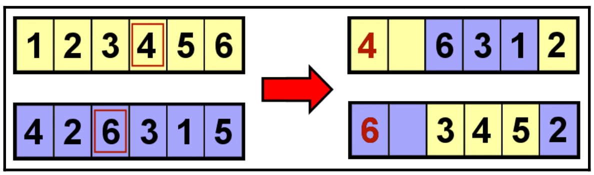
Ordered crossover example—step 1

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. This is shown in the diagram as well:



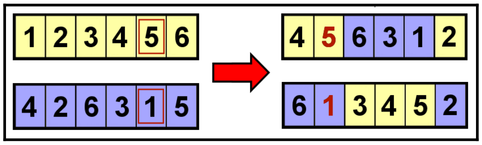
Ordered crossover example—step 2

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. The results are illustrated in the following diagram:



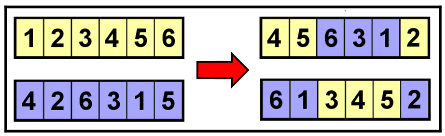
Ordered crossover example—step 3

We continue in a similar fashion with the next genes not yet present in the offspring, and fill in the last available spots, as depicted in the following diagram:



Ordered crossover example—step 4

This completes the process of producing two valid offspring chromosomes, as the following diagram demonstrates:



Ordered crossover example—step 5

There are numerous other methods to implement crossover, some of which we will encounter later in this book. However, thanks to the versatility of genetic algorithms, you can always come up with your own methods. We will now look at the mutation methods.

Mutation methods

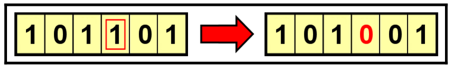
The mutation is the last genetic operator to be applied in the process of creating a new generation. The mutation operator is applied to the offspring that were created as a result of the selection and crossover operations.

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.

The following sections describe some of the commonly used mutation methods and their typical use cases. However, remember that you can always choose to use your own problem-specific mutation method that you deem more suitable for your particular use case.

Flip bit mutation

When applying the flip bit mutation to a binary chromosome, one gene is randomly selected and its value is flipped (complemented), as shown in the following diagram:

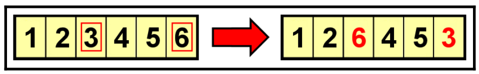


Flip bit mutation example

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, as shown in the following diagram:

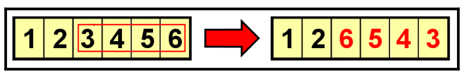


Swap mutation example

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, as depicted in the following diagram:

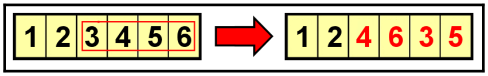


Inversion mutation example

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), illustrated as follows:



Scramble mutation example

In the next section, we will cover some other types of specialized operators created for real-coded genetic algorithms.

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.

Historically, genetic algorithms used binary strings to represent integers as well as real numbers, however, this was not ideal. The precision of a real number represented using a binary string is limited by the length of the string (number of bits). Since we need to determine this length in advance, we may end up with binary strings that are too short, resulting in insufficient precision, or are overly long.

Moreover, when a binary string is used to represent a number, the significance of each bit varies by its location—the most significant bit being on the left. This can cause imbalance related to schemas—the patterns occurring in the chromosomes. For example, the schema 1**** (representing all five-digit binary strings starting with 1) and the schema ****1 (representing all five-digit binary strings ending with 1) both have an order of 1 and a defining length of 0, however, the first one carries much more significance than the other.

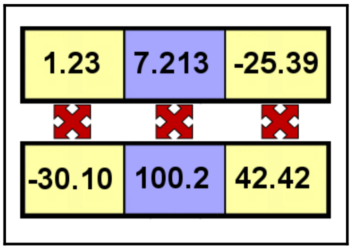
Instead of using binary strings, arrays of real-valued numbers were found to be a simpler and better approach. For example, if we have a problem involving three real-valued parameters, the chromosome will look like [x_1, x_2, x_3], where x_1, x_2, x_3 represent real numbers, such as [1.23, 7.2134, -25.309] or [-30.10, 100.2, 42.424].

The various selection methods mentioned earlier in this chapter 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 covered so far 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 following pairs:

- 1.23 and -30.10 (first dimension)
- 7.213 and 100.2 (second dimension)
- -25.39 and 42.42 (third dimension)

This is illustrated in the following diagram:



Real-coded chromosomes crossover example

Similarly, the mutation operator, when applied to a real-coded chromosome, will apply separately to each dimension.

Several of these real-coded crossover and mutation methods are described in the following sections. Later, in `Chapter 6`, *Optimizing Continuous Functions*, we will get to see them in action.

Blend crossover

In the **blend crossover** (also known as **BLX**), each offspring is randomly selected from the following interval created by its parents:

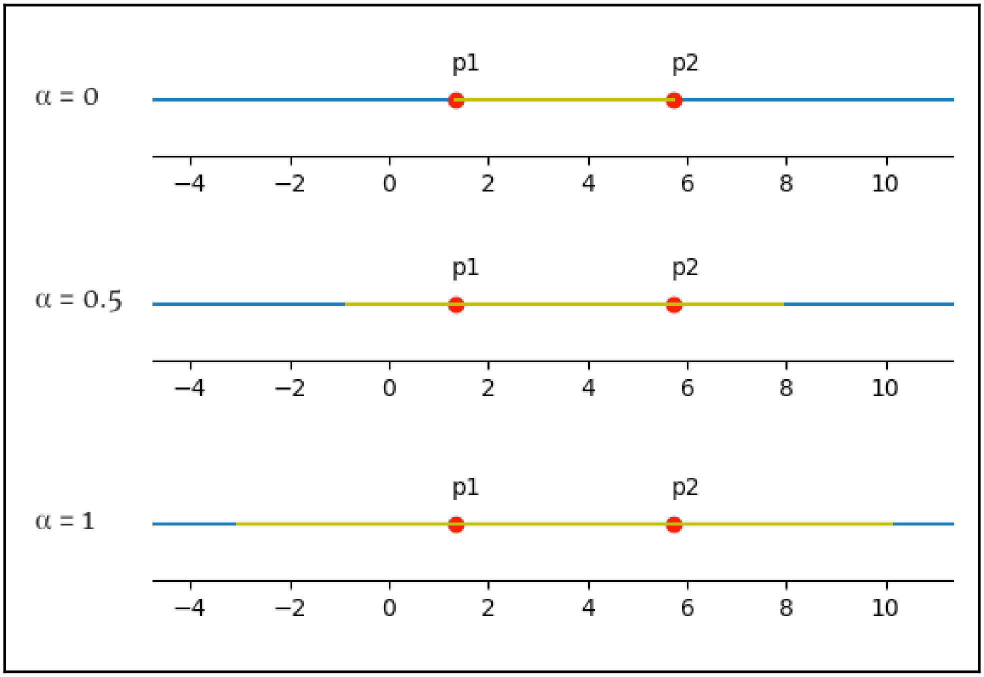
$$[parent_1 - \alpha(parent_2 - parent_1),\;\; parent_2 + \alpha(parent_2 - parent_1)]$$

The parameter α is a constant, whose value lies between 0 and 1. With larger values of α, the interval gets wider.

For example, if the parents' values are 1.33 and 5.72, the following will be the case:

- An α value of 0 will yield the interval [1.33, 5.72] (similar to the interval between the parents)
- An α value of 0.5 will yield the interval [-0.865, 7.915] (twice as wide as the interval between the parents)
- An α value of 1.0 will yield the interval [-3.06, 10.11] (three times wider than the interval between the parents)

These examples are illustrated in the following diagram where the parents are labeled by **p1** and **p2**, and the crossover interval is colored yellow:



Blend crossover example

When using this crossover method, the α value is commonly set to 0.5.

Simulated binary crossover

The idea behind the **simulated binary crossover** (**SBX**) is to imitate the properties of the single-point crossover that is commonly used with binary-coded chromosomes. One of these properties is that the average of the parents' values is equal to that of the offsprings' values.

When applying SBX, the two offspring are created from the two parents using the following formula:

$$offspring_1 = \frac{1}{2}[(1+\beta)parent_1 + (1-\beta)parent_2]$$
$$offspring_2 = \frac{1}{2}[(1-\beta)parent_1 + (1+\beta)parent_2]$$

Here, β is a random number referred to as the *spread factor*.

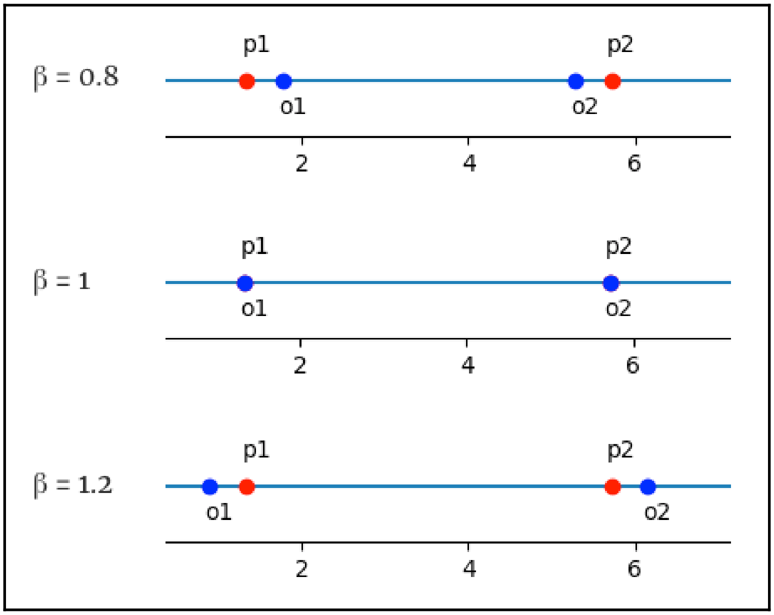
This formula has the following notable properties:

- The average of the two offspring is equal to that of the parents, regardless of the value of β.
- When the β value is 1, the offspring are duplicates of the parents.
- When the β value is smaller than 1, the offspring are closer to each other than the parents were.
- When the β value is larger than 1, the offspring are farther apart from each other than the parents were.

For example, if the parents' values are 1.33 and 5.72, the following will be the case:

- A β value of 0.8 will yield the offspring 1.769 and 5.281
- A β value of 1.0 will yield the offspring 1.33 and 5.72
- A β value of 1.2 will yield the offspring 0.891 and 6.159

These cases are illustrated in the following diagram where the parents are labeled by **p1** and **p2**, and the offspring by **o1** and **o2**:



Simulated binary crossover example

In each of the preceding cases, the average value of the two offspring is 3.525, which is equal to the average value of the two parents.

Another property of the binary single-point crossover that we would like to preserve is the similarity between offspring and parents. This translates to the random distribution of the β value. The probability of β should be much higher for values around 1, where the offspring are similar to the parents. To achieve that, the β value is calculated using another random value, denoted by *u*, that is uniformly distributed over the interval [0, 1]. Once the value of *u* is picked, β is calculated as follows:

If u ≤ 0.5: $$\beta = (2u)^{\frac{1}{\eta+1}}$$

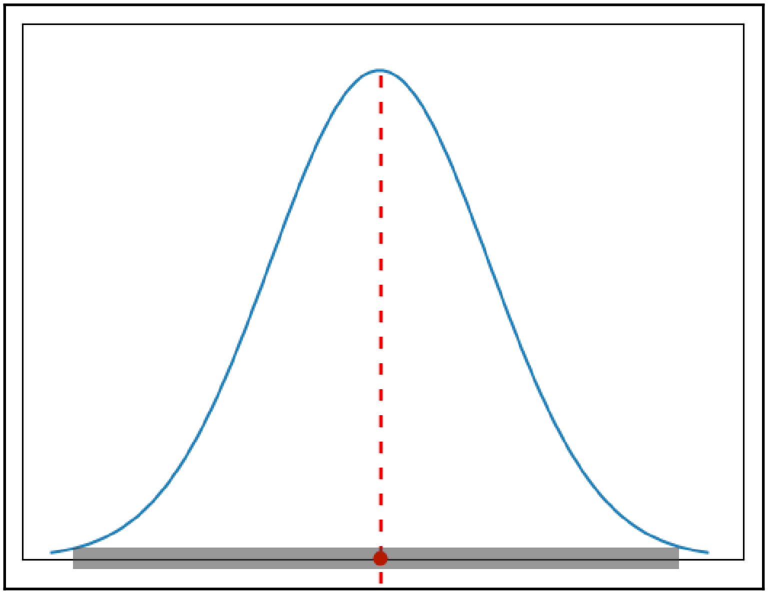
Otherwise: $$\beta = \left[\frac{1}{2(1-u)}\right]^{\frac{1}{\eta+1}}$$

The parameter η (eta) used in these formulas is a constant representing the **distribution index**. With larger values of η, offspring will tend to be more similar to their parents. A common value of η is 10 or 20.

Real mutation

One option for applying mutation in real-coded genetic algorithms is to replace any real value with a brand new one, generated randomly. However, this can result in a mutated individual that has no relationship to the original individual.

Another approach is to generate a random real number that resides in the vicinity of the original individual. An example of such a method is the **normally distributed (or Gaussian) mutation**: a random number is generated using a normal distribution with a mean value of zero and some predetermined standard deviation as shown in the following plot:



Gaussian mutation distribution example

In the next two sections, we will go over a couple of advanced topics, namely elitism and niching.

Understanding 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.

Another interesting way to enhance the results of genetic algorithms is the use of niching, as described in the next section.

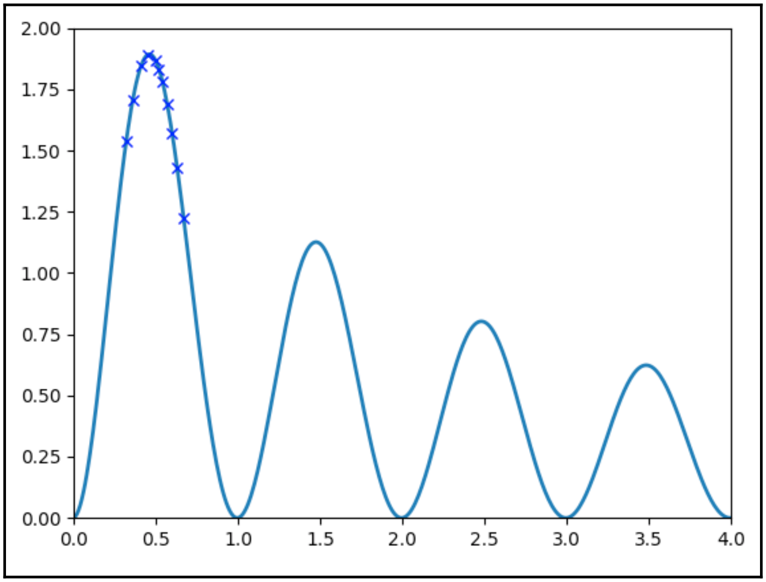
Niching and sharing

In nature, any environment is further divided into multiple sub-environments, or niches, populated by various species taking advantage of the unique resources available in each niche, such as food and shelter. For example, a forest environment is comprised of the treetops, the shrubs, the forest floor, the tree roots, and so on; each of these accommodating different species who are specialized for living in that niche and takes advantage of its resources.

When several different species coexist in the same niche, they all compete over the same resources, and a tendency is created to search for new, unpopulated niches and populate them.

In the realm of genetic algorithms, this niching phenomenon can be used to maintain the diversity of the population as well as for finding several optimal solutions, each considered a niche.

For example, suppose our genetic algorithm seeks to maximize a fitness function having several peaks of varying heights, such as the one in the following plot:

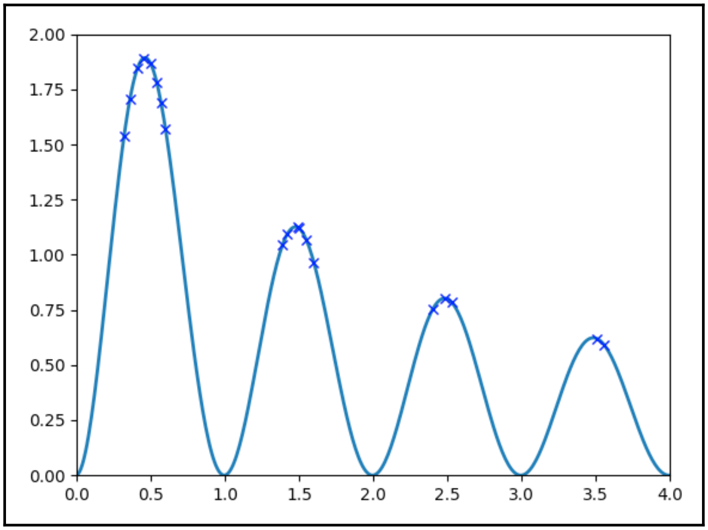


Expected genetic algorithm results without niching

As the tendency of the genetic algorithm is to find the global maximum, we expect, after a while, to see most of the population concentrating around the top peak. This is indicated in the figure by the locations of the × marks on the function graph, representing individuals in the current generation.

However, there are implementations where, in addition to the global maximum, we would like to find some (or all) of the other peaks. To make this happen, we could think of each peak as a niche, offering resources in the amount proportional to its height. We then find a way to share (or divide) these resources among the individuals occupying it. This will ideally drive the population to be distributed accordingly, with the top peak attracting the most individuals as it offers the most reward, and the other peaks populated with decreasing portions of the population as they offer smaller amounts of reward.

This ideal situation is depicted in the following figure:



Ideal genetic algorithm results with niching

The challenge now is to implement this sharing mechanism. One option to accomplish sharing is to divide the raw fitness value of each individual with (some function of) the combined distances from all the other individuals. Another option would be to divide the raw fitness of each individual with the number of other individuals within a certain radius around it.

Serial niching versus parallel niching

Unfortunately, the niching concept as described previously can prove hard to implement as it increases the complexity of the fitness calculation. In practice, it will also require the population size to be the original one multiplied by the number of the expected peaks (which is generally unknown).

One way to overcome these issues is to find the peaks one at a time (serial niching) instead of attempting to find all of them at the same time (parallel niching). To implement serial niching, we use the genetic algorithm as usual and find the best solution. We then update the fitness function so that the area of the maximum point that was found is flattened, and repeat the process of the genetic algorithm.

Ideally, we will now find the next best peak, as the original peak is no longer present. We can repeat this process iteratively and find the next best peak at each iteration.

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 we have seen in the preceding sections, 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
- Other stopping condition(s)
- Elitism (used or not; what size)

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.

If making all these choices appears to be an overwhelming task, don't fret! In the chapters that follow, we will be iterating the process of making these choices time and again for the various types of problems we will tackle. After reading this book, you will be able to look at new problems and make your own wise choices.

Summary

In this chapter, you were first introduced to the basic flow of the genetic algorithm. We then went over the key components of the flow, which included creating the population, calculating the fitness function, applying the genetic operators, and checking for stopping conditions.

Next, we went over various methods of selection, including roulette wheel selection, stochastic universal sampling, rank-based selection, fitness scaling, and tournament selection, and demonstrated the differences between them.

We continued by reviewing several methods of crossover, such as single-point, two-point, and k-point crossover, as well as ordered crossover and partially matched crossover.

You were then introduced to a number of mutation methods, such as flip bit mutation, followed by the swap, inversion, and the scramble mutation.

Real-coded genetic algorithms were presented next, with their specialized chromosome encoding as well as their custom genetic operators of crossover and mutation.

This was followed by an introduction to the concepts of elitism, as well as niching and sharing as used in genetic algorithms.

In the last part of the chapter, you were presented with the various choices that need to be made when approaching a problem to be solved using genetic algorithms; a procedure that will be repeated time and again throughout the book.

In the next chapter, the real fun begins—coding with Python! You will be introduced to DEAP—an evolutionary computation framework that can be used as a powerful tool for applying genetic algorithms to a wide array of tasks. DEAP will be used in the rest of the book as we develop Python programs that tackle numerous different challenges.

Further reading

For more information, please refer to *Chapter 8*, *Genetic Algorithms*, from the book *Artificial Intelligence with Python* by Prateek Joshi, January 2017, available at `https://subscription.packtpub.com/book/big_data_and_business_intelligence/9781786464392/8`.