Keen: Kotlin Genetic Algorithms Framework

Thesis for the degrees of

Civil Engineer in Computer Science

and

Master of Science in Computing

Ignacio Slater Muñoz
Departamento de Ciencias de la Computación
Facultad de Ciencias Físicas y Matemáticas
Universidad de Chile.
Thesis Author

Nancy Hitschfeld. PhD.
Departamento de Ciencias de la Computación
Facultad de Ciencias Físicas y Matemáticas
Universidad de Chile
Thesis Supervisor

Alexandre Bergel. PhD.
Relational AI
Switzerland
Second Supervisor

Santiago, Chile
2023-08-02
`v0.4.2307.4`

Abstract

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

This is the second paragraph. Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

And after the second paragraph follows the third paragraph. Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

After this fourth paragraph, we start a new paragraph sequence. Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Contents

Chapter 1

Theoretical Framework

The objective of this chapter is to provide the reader with the theoretical background necessary to understand the rest of the document.

1.1 Evolutionary Algorithms

In the field of computational intelligence, evolutionary algorithms (EA) [**yuIntroductionEvolutionaryAlgorithms2010**] are a family of algorithms inspired by the process of natural selection. They are part of the larger field of evolutionary computation,[1] which is a subfield of metaheuristics.[2]

EAs are algorithms that perform optimization or learning tasks by evolving solutions to a given problem. These tasks may range from function optimization to machine learning or game AI development. EAs have three main characteristics:

- **Population-based:** These algorithms work with a population of solutions, allowing them to explore the search space in parallel.[3]

- **Fitness-oriented:** The solutions in the population are evaluated using a fitness function, which is a problem-dependent function that assigns a value to each solution based on its quality. The goal of the algorithm is to find the solution with the highest[4] fitness.

- **Variation-driven:** The candidate solutions are modified using genetic operators, such as mutation, crossover, and selection, to create new solutions. These operators are usually based on the biological processes of mutation and recombination.

While these principles serve as the foundation for most EAs, it's important to note that some variants may prioritize some principles over others, or introduce new principles. This diversity allows EAs to be adapted to a wide range of problems and scenarios.

[1]See **??**

[2]See **??**

[3]Not to be confused with parallelization, which is a technique used to speed up the execution of an algorithm by running it in parallel on multiple processors. "Parallel" in this context means that the algorithm is exploring multiple points in the search space (**??**) simultaneously.

[4]In some cases, the goal is to minimize the fitness function, in which case the algorithm will aim to find the solution with the lowest fitness.

1.2 Genetic Algorithms

Genetic Algorithms (GA)[5] [**hollandAdaptationNaturalArtificial1992a**, **kozaGeneticProgrammingProgramming1992a** **yuIntroductionEvolutionaryAlgorithms2010**, **shiffmanNatureCode2012**] are a type of EA where a ***population*** of ***individuals***[6] representing candidate solutions to an optimization problem evolves towards better solutions. Each individual is defined by its location in the search space, known as its ***genotype***[7], and its fitness value, computed by a ***fitness function***. At a high level, GA is an automatic method for problem-solving, starting from a *high-level statement* of the desired outcome, without needing the user to predefine the solution's form or structure.

The classical GA operates as follows:

Algorithm 1 Genetic Algorithm

```
1: population ← initializePopulation()            ▷ Creates a random population of individuals
2: evaluate(population)                            ▷ Calculates the fitness of each individual
3: repeat
4:     parents ← selectParents(population)         ▷ Selects a subset of individuals as parents
5:     alter(offspring)                  ▷ Applies genetic operators to offspring, creating variations
6:     population ← selectSurvivors(population, offspring)  ▷ Selects individuals for the next generation
7: until termination condition is met ▷ Could be a pre-defined number of generations, a desired fitness level, etc.
8: return fittest(population)                      ▷ Returns the most fit individual
```

Here, initializePopulation() generates a random population of individuals, while evaluate(*population*) assesses the fitness of each individual in the population.

The algorithm then continually performs the following steps until a termination condition is met:

1. selectParents(*population*) chooses a subset of individuals from the population to parent the next generation.
2. alter(*offspring*) modifies the offspring to introduce variability ("new genetic material") into the population.
3. evaluate(*offspring*) computes the fitness of each new individual.
4. selectSurvivors(*population*, *offspring*) selects the individuals that will survive to the next generation.

Finally, the algorithm returns the most fit individual in the population.

The exact implementation of each of these steps depends on the specific problem at hand. Factors such as the problem's complexity, the representation of individuals, or even the computational resources available, can greatly influence the choice of methods used for initialization, selection, alteration, and survivor selection.

1.2.1 Representation and Evaluation

One of the most important aspects of a GA is the representation of the individuals. The representation is the encoding of potential solutions to the problem into a form that can be manipulated by the algorithm. This defines the search space of the algorithm, and it is one of the main factors that determines the performance of the algorithm.

The most general representation of an individual is a matrix of genes[8] called the ***genotype***[9] of the individual, where each column of the matrix is called a ***chromosome***[10].

Definition 1.1 (Cardinality of the search space)**.** *The **cardinality of the search space** is the number of different individuals that can be represented by the encoding.*

[5]Also known as Simple Genetic Algorithms (SGA) [**yuIntroductionEvolutionaryAlgorithms2010**], or Traditional Genetic Algorithms (TGA) [**shiffmanNatureCode2012**].

[6]See **??**.

[7]See **??**

[8]See **??**.

[9]See [**wilhelmstotterJeneticsJavaGenetica**].

[10]See **??**.

Formally, given a vector of alphabets $(\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n)$, and a representation **G** *with n chromosomes of length $(m_1, m_2, \ldots, m_n)$ where each chromosome is encoded using the alphabet $\mathcal{A}_i$, the cardinality of the search space S is defined as:*

$$|S| = \prod_{i=1}^{n} |\mathcal{A}_i|^{m_i} \tag{1.1}$$

Note that this definition assumes that the chromosomes are independent, which may not be the same for all evolutionary algorithms.

Remark. *In the original publication of the GA [***hollandAdaptationNaturalArtificial1992a***], the genotype was known as the* ***environment*** *(E) and the search space was defined as a class $\mathcal{E}$ of all possible environments.*

To illustrate this concept, consider the following problem: given a binary string of length n, find the string that has the most ones.[11] In this case, we can use a single column matrix **G** to represent the individual, where each gene $g_i \in \mathcal{A}$ represents the i-th bit of the string, where $\mathcal{A} = \{0, 1\}$ is the alphabet containing the two possible values of a bit.

Then,

$$|S| = \prod_{i=1}^{1} |\mathcal{A}|^n = 2^n$$

Knowing this, we can conclude that an exhaustive search of the search space would require evaluating 2^n individuals, and thus the algorithm would have a time complexity of $\mathcal{O}(2^n)$.

This is a very simple example, but we can see how a naive search algorithm would have a very high time complexity. This would be of the utmost importance in a real world problem, where the search space would be much larger.

With a representation defined, we can now define an evaluation method for the individuals, which is done using a ***fitness function***.

Definition 1.2 (Fitness function)**.** *A* ***fitness function*** *is a function $\phi : S \to \mathbb{R}^n$, where S is the search space and n is the number of objectives of the optimization problem, that takes a genotype as input and returns a vector of real numbers representing how close the individual is to the global optimum of each objective.*

The fitness function is usually defined by the user of the algorithm, and it is problem dependent.

Definition 1.3 (Batch fitness function)**.** *A* ***batch fitness function*** *$\Phi : \mathbb{P} \to \mathbb{R}^{m \times n}$ is a function that maps a population to a matrix of real numbers, where m is the number of individuals in the population and n is the number of objectives of the optimization problem.*

The one max problem is a maximization problem with a single objective, so the fitness function would be defined as follows:

$$\phi_{\mathbf{G}} = \sum_{i=1}^{n} g_i \tag{1.2}$$

In the ***Representation and Evaluation*** section, we examined the key aspect of a genetic algorithm (GA) — the representation of individuals, their encoding and search space. The performance of a GA significantly depends on how solutions are encoded to form individuals. We use the concepts of genotype, chromosome, and gene to describe the individual's representation. The cardinality of the search space, defined as the total number of different individuals that can be represented, is crucial as it impacts the algorithm's complexity. We introduced the "One Max" problem as an example, using a binary string representation. The fitness function, which evaluates individuals' fitness, plays a critical role in navigating the search for optimal solutions. In the "One Max" problem, the fitness function sums the binary string elements, representing the number of ones in the string.

[11]This is known as the ***One Max*** problem [**OneMaxProblema**] or ***Ones Counting*** problem [**wilhelmstotterJeneticsJavaGenetica**].

1.2.2 Initialization

GA operates on a group of individuals called a ***population***. The algorithm designer must define the size of the population, and how to initialize it. The initialization process is usually random, but it can also be guided by some prior knowledge about the problem being solved. For example, if the problem is to find a solution to a maze, the population could be initialized with individuals that represent paths from the start to the end of the maze. This would speed up the search process, since the algorithm would not have to start from scratch.

Once the population is initialized, the algorithm performs an evaluation of each individual in the population, and assigns a ***fitness value*** to each individual. This is done in an effort to learn something about the problem, and to guide the search process towards better solutions.

In the case of the ***One Max*** problem, there is no prior knowledge about the problem, so the population via a blind search of the search space (in other words, the initialization is random). This is done by generating a random binary string of length n for each individual in the population.

Let's assume that we have a population of size 4, and that the length of the binary strings is $n = 4$.

The initialization process could generate the following individuals:[12]

Generation 0

Individual	**Binary string**	**Fitness**
I_1	1100	2
I_2	0001	1
I_3	0000	0
I_4	0100	1

Table 1.1: Population of individuals in generation 0

	Fitness	**Individual**
Best	2	I_1
Worst	0	I_3
Average	1	
Standard deviation	0.817	

Table 1.2: Fitness of the individuals in generation 0

In the initialization phase of a genetic algorithm, we define and setup the population of individuals to be used in the search process. This population can be randomly generated or informed by some prior knowledge about the problem at hand. Each individual is evaluated to determine its fitness, guiding the algorithm's search for optimal solutions. In our "One Max" problem example, we initialized a population of four individuals with binary strings of length $n = 4$ and evaluated their fitness. This setup marks the beginning of the evolutionary process, setting the stage for the subsequent stages of selection (**??**), crossover (**??**), and mutation (**??**).

1.2.3 Selection

Once initialization is complete, the Genetic Algorithm (GA) enters its main loop, where the core evolutionary processes take place. In the GA, a mechanism that simulates natural selection operates, providing fitter individuals with higher chances of survival and breeding opportunities.

Suppose that we have a population P of N individuals, each with a fitness value f_i, where $i \in \{1, \ldots, N\}$. Let σ be the survival rate, a parameter controlling the degree of elitism. This is the proportion of individuals that

[12]Since the nature of genetic algorithms is stochastic, the initialization process could generate different individuals each time the algorithm is run. For this example, we selected a specific set of individuals in a way that makes it easier to get a grasp of the algorithm.

will survive (unmodified) to the next generation. The GA will then select $\lfloor \sigma N \rfloor$ individuals to survive to the next generation, and $\lceil (1-\sigma)N \rceil$ individuals to be replaced by the offspring.[13]

Definition 1.4 (Selection operator)**.** *An operator used to select individuals from a population.*

Formally, a selection operator is a function

$$\Sigma : \mathbb{P} \times \mathbb{N} \times \cdots \to \mathbb{P}; (P, n, \dots) \mapsto \Sigma(P, n, \dots)$$

where:

- $\mathbb{P}$ *is the set of populations;*
- $\mathbb{N}$ *is the set of positive natural numbers;*
- P *is a population;*
- n *is the number of individuals to select from* P*;*
- $\Sigma(P,\, n)$ *is the population of* n *individuals selected from* P*.*

The selection operator is typically implemented as a ***stochastic*** operator, introducing some randomness into the selection process.

As an illustration, consider a *roulette wheel* selection operator[14] applied to a population of four individuals with a survival rate of 0.25. In this selection scheme, each individual is assigned a selection probability proportional to its fitness value (assuming higher fitness is better). The selection probability of an individual i is calculated as follows:

$$p_i = \frac{f_i}{\sum_{j=1}^{N} f_j} \tag{1.3}$$

In our example, the selection probabilities are detailed in **??**.

Individual	Fitness	Selection Probability
I_1	2	50%
I_2	1	25%
I_3	0	0%
I_4	1	25%

Table 1.3: Selection probabilities for the individuals in the example population.

The selection operator then selects individuals at random, each with a probability equal to their selection probability. Suppose I_2 is selected to survive to the next generation, then I_1, I_3, and I_4 will be replaced by the offspring.

This section has introduced the concept of selection in GAs, which will be explored further in **??**. Next, we will delve into variation operators responsible for generating the offspring that will replace the individuals not selected to survive to the next generation.

1.2.4 Variation

Variation is the process of creating new individuals from existing ones in the pursuit of exploring the solution space. This is crucial in a Genetic Algorithm (GA) to avoid premature convergence to sub-optimal solutions. In a GA, variation is achieved by applying ***variation operators*** to the individuals in the population. The most common variation operators are ***crossover*** and ***mutation***, which will be explored in this section.

[13] The sole purpose of employing both *floor* and *ceiling* functions is to guarantee that the total number of individuals chosen for survival and replacement equals N, which makes these functions interchangeable in this context.

[14] See **??** for a detailed description of the roulette wheel selection operator.

Definition 1.5 (Variation operator)**.** *A variation operator is used to create new individuals from existing ones. Formally, it is a variadic function represented as*

$$\varphi : \mathbb{P} \times \mathbb{R} \times \cdots \to \mathbb{P};\ (P, \rho, \dots) \mapsto \varphi(P, \rho, \dots)$$

where:

- $\mathbb{P}$ *is the set of all possible populations,*
- $\mathbb{R}$ *is the set of real numbers,*
- P *is the population to be varied,*
- ρ *is the probability of applying the operator to an individual in the population.*

The additional arguments depend on the specific implementation of the variation operator. The role of these arguments will be clarified in **??***.*

Crossover

The variation operator in genetic algorithms often involves a procedure known as ***crossover***, which emulates the process of genetic recombination observed in nature.[15] This process involves the exchange of genetic material between two individuals to create a new generation.

Definition 1.6 (Crossover operator)**.** *A crossover operator is a variation operator that is used to create new individuals from existing ones by performing a recombination of their genetic material.*

Formally, it is a variadic function represented as

$$X : \mathbb{P} \times \mathbb{R} \times \cdots \to \mathbb{P}; (P, \rho, \dots) \mapsto X(P, \rho, \dots)$$

where:

- $\mathbb{P}$ *is the set of all possible populations,*
- $\mathbb{R}$ *is the set of real numbers,*
- P *is the population to be varied,*
- $\rho \in [0, 1]$ *is the probability of applying the operator to an individual in the population.*

For the problem under consideration, we utilize a simplified form of the ***single-point crossover***[16] operator. This operator selects the first half of the genes from two parent individuals and generates two new offspring by interchanging these selected genes.

For instance, consider two parent individuals selected via the ***roulette wheel*** selector described earlier: $I_1 = 1100$ and $I_2 = 0001$. The single-point crossover operator selects the first half of the genes from each parent, i.e., 11 from I_1 and 00 from I_2, and produces a pair of new chromosomes with the first half, and produces two new offspring by exchanging these selected parts: $O_1 = 1101$ and $O_2 = 0000$ (as illustrated in **??**).

Following another iteration of the single-point crossover operator, we can generate a result as shown in **??**, leading to a new population $\mathbf{O} = \{(0000,\ 0),\ (1101,\ 3),\ (0101,\ 2)\}$.

If we now use these offspring as-is to create the next generation, we would obtain the population shown in **??**:

[15] This is referred to as ***crossing-over*** in [**hollandAdaptationNaturalArtificial1992a**].
[16] See **??**.

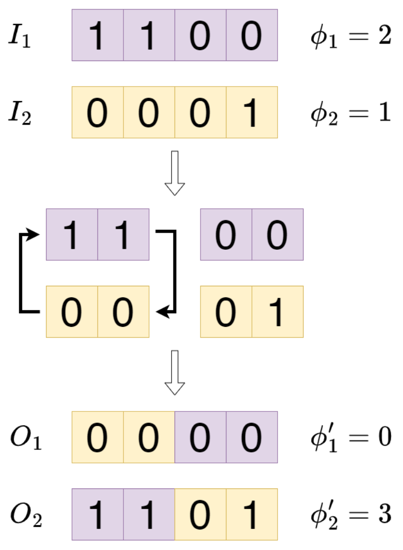


Figure 1.1: Single-point crossover

Generation 0 → Generation 1

$\mathbf{I}$	$\Phi_\mathbf{I}$	$\mathbf{O}$	$\Phi_\mathbf{O}$
$\begin{bmatrix}1100\\0001\end{bmatrix}$	$\begin{bmatrix}2\\1\end{bmatrix}$	$\begin{bmatrix}0000\\1101\end{bmatrix}$	$\begin{bmatrix}0\\3\end{bmatrix}$
$\begin{bmatrix}0001\\0100\end{bmatrix}$	$\begin{bmatrix}1\\1\end{bmatrix}$	$\begin{bmatrix}0101\\\cdot\end{bmatrix}$	$\begin{bmatrix}2\\\cdot\end{bmatrix}$

Table 1.4: Illustration of the single-point crossover operation. In this procedure, two parent individuals are selected and a cut point is chosen. Each offspring is then formed by combining the genes from the parents: one gets the genes from the first part of the first parent and the second part of the second parent, while the other gets the genes from the first part of the second parent and the second part of the first parent. Here, · represents a "discarded" value (since according to the survival rate, only three offspring need to be produced).

Generation 1

Individual	**Binary String**	**Fitness**
I_2	0001	1
O_1	0000	0
O_2	1101	3
O_3	0101	2

Table 1.5: Population after applying the single-point crossover operator. Note that I_2 is the survivor of the previous generation picked in **??**.

	Fitness	**Individual**
Best	3	O_2
Worst	0	O_1
Average	1.25	
Standard deviation	1.291	

Table 1.6: Fitness of the population after applying the single-point crossover operator. "Best" refers to the individual with the highest fitness, and "Worst" refers to the individual with the lowest fitness

As observed from **??**, the average fitness of the population has increased from 1 to 1.25, and the fitness of the best individual has improved from 2 to 3. This improvement showcases how the crossover operator helps guide the search towards superior solutions.

While the crossover operation has indeed enhanced the average fitness of the population, to further augment genetic diversity within the population and prevent premature convergence to suboptimal solutions (local optima), the introduction of a ***mutation*** operator is often beneficial. This operation will be discussed in the next section.

Mutation

One limitation of the crossover operator is its reliance on existing genetic material in the population.

This constraint can lead to premature convergence, particularly for problems with numerous local optima such as the ***Rastrigin function*** optimization.[17]

To counteract this and introduce *diversity* in the population, we use the *mutation* operator.

This operator alters the genetic material of an individual within the population according to a specific probability.

Definition 1.7 (Mutation operator)**.** *A mutation operator is a function that alters the genetic material of individuals within a population based on a certain probability, thereby producing a new population.*

Formally, a mutation operator is a variadic function

$$M : \mathbb{P} \times \mathbb{R} \times \cdots \to \mathbb{P};\ (P,\ \mu,\ \dots) \mapsto M(P,\ \mu,\ \dots)$$

where:

- $\mathbb{P}$ *represents the set of all possible populations;*
- $\mathbb{R}$ *represents the set of real numbers;*
- P *is the current population;*
- μ *represents the mutation rate—the probability that an individual in the population will undergo mutation.*

The other arguments are specific to the mutation operator being used.

For instance, in the “One Max” problem, we can use a ***bit-flip*** mutation.[18] This operator scans each gene in an individual and substitutes it with its complement according to a predetermined probability.

Suppose we set the ***mutation rate*** $\mu = 1$, and apply the mutation operator to the population resulting from the crossover operation described in **??**. As shown in **??**, the resulting population would be $\mathbf{O} = \{(1111,\ 4), (0010,\ 1), (1010,\ 3)\}$.

Generation 0 → Generation 1

$\mathbf{I}$	$\Phi_{\mathbf{I}}$	$\mathbf{O}$	$\Phi_{\mathbf{O}}$
$\begin{bmatrix}0000\\1101\\0101\end{bmatrix}$	$\begin{bmatrix}0\\3\\2\end{bmatrix}$	$\begin{bmatrix}1111\\0010\\1010\end{bmatrix}$	$\begin{bmatrix}4\\1\\2\end{bmatrix}$

Table 1.7: Illustration of the ***bit-flip*** mutation operator applied to the population resulting from the crossover operation in **??**.

If the mutated offspring were used to generate the next population, as shown in **??**, you can observe the increased diversity.

[17]See **??**.
[18]See **??**.

Generation 1

Individual	Binary String	Fitness
I_2	0001	1
O'_1	1111	4
O'_2	0010	1
O'_3	1010	2

Table 1.8: Population after applying the ***bit-flip*** mutation operator to the population resulting from the crossover operation in **??**.

	Fitness	Individual
Best	4	O'_1
Worst	0	(I_2, O'_2)
Average	2	
Standard deviation	1.414	

Table 1.9: Fitness of the population after applying the ***bit-flip*** mutation operator to the population resulting from the crossover operation in **??**.

Clearly, the mutation operator has introduced diversity into the population. In the original population, no individual had a 1[19] in the third position. Therefore, the crossover operator could never produce an individual with a 1 in that position. But the mutation operator has introduced three individuals with a 1 in the third position.[20]

In conclusion, the mutation operator plays a crucial role in genetic algorithms by introducing diversity into the population and preventing premature convergence to local optima. It facilitates a more thorough exploration of the search space, allowing new and potentially beneficial traits to emerge. However, it's essential to note that a high mutation rate might disrupt advantageous traits, while a low rate might not sufficiently prevent premature convergence. The specific mutation operator and the mutation rate used are crucial factors in shaping the genetic algorithm's search process.

With this, the variation process is complete, and we can proceed to the next step of the genetic algorithm.

1.2.5 Termination

After each generation – when a new population is fully created – the genetic algorithm verifies if the termination criteria have been met. If so, the algorithm terminates and returns the best individual identified. Otherwise, the process continues to the next generation.

Consider a scenario where the termination criterion is defined as the discovery of an individual possessing the maximum number of ones, represented as 1111. This would correspond to a condition where $\phi_{\mathbf{G}} = 4$.

Recall that we found the individual 1111 after applying the variation operators to the population. As a result, the termination criterion is met and the genetic algorithm concludes its process.

It's worth noting that not all search space has been explored, as demonstrated in **??**. The algorithm's fitness-oriented search strategy means it performs a guided, rather than exhaustive, search. However, the increasing fitness of the population's individuals across generations indicates convergence towards an optimal solution.

For small search spaces like in our example, the distinction between this algorithm and a purely random search may seem minimal. But for larger search spaces, as explored later in this thesis, the difference becomes highly significant.

It's important to underline that genetic algorithms, being stochastic in nature, do not guarantee discovery of the optimal solution. Their effectiveness depends on various factors such as the fitness function, the representation

[19] $P = \{1100, 0001, 0000, 0100\}$
[20] $P' = \{0001, 1111, 0010, 1010\}$

	00	01	10	11
00	■	■	■	
01	■	■		
10			■	
11	■	■		■

Table 1.10: Candidates from the search space that were explored by the genetic algorithm. Cells that are coloured in dark gray represent candidates that were explored by the genetic algorithm. Each individual is defined by the row and column that it occupies in the search space, where the row represents the first 2 bits of the individual and the column represents the last 2 bits of the individual; e.g. the individual 0001 is located in the first row and second column of the table.

scheme, the variation operators, and the selection strategy. These components and their impact on performance across different problems will be thoroughly examined in this thesis.

In summary, the termination phase of the genetic algorithm represents a crucial step in determining the overall process outcome. By utilizing a targeted termination criterion – such as the discovery of an individual with the highest possible fitness score – the algorithm effectively navigates the search space. While not exhaustive in its exploration, the algorithm uses a fitness-oriented strategy to guide its trajectory towards an optimal solution. It's essential to recognize the inherent limitations of genetic algorithms due to their stochastic nature. Despite these, their potential to outperform random searches, especially in large search spaces, is considerable. However, success relies heavily on choosing appropriate parameters and procedures, a topic to be explored in-depth in subsequent sections of this thesis.

1.3 Genetic Programming

Genetic Programming (GP) [**kozaGeneticProgrammingProgramming1992a**, **kozaGeneticProgrammingII1994**, **poliFieldGuideGenetic2008a**, **yuIntroductionEvolutionaryAlgorithms2010**] is a specialized branch of Evolutionary Algorithms (EA) which focuses on evolving a population of computer programs to solve a given problem. One can perceive GP as an extension of Genetic Algorithms (GA), the key distinction being the problem each approach solves: GA optimizes parameters to enhance a given function, whilst GP induces programs.[21]

Despite these differences, GP and GA share various characteristics such as the utilization of a population of individuals, the employment of a fitness function to evaluate the individuals, and the application of genetic operators to generate new individuals. Notwithstanding, GP adopts a unique representation for the individuals and unique genetic operators.

Remark. *Although GP operates a fitness-guided search in the space of computer programs, it can be deemed as an optimization problem, akin to GA.*

Each individual in a GP population embodies a computer program composed of a set of primitives, referred to as ***functions*** and ***terminals***. An intuitive way to comprehend primitives is by visualizing a composite pattern where the functions equate to composite objects and the terminals to leaf objects (refer to **??**). An ***abstract syntax tree*** (AST) is an example of a program representation where the functions correspond to the internal nodes, and the terminals to the leaf nodes.

The general GP algorithm is represented in **??**, which closely resembles the GA algorithm with the main disparity being the nature of the *genetic operators* in GP.

In the ensuing sections, we will delve into the fundamental components of a GP algorithm and elucidate them through an example.

[21]For a formal definition of program induction, refer to **??**.

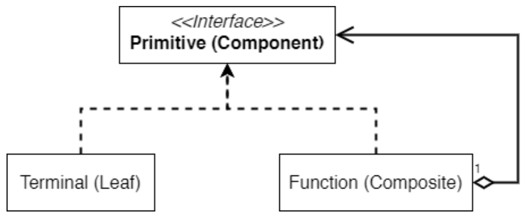


Figure 1.2: The composite structure of a GP individual, illustrating the relationship between functions and terminals

Algorithm 2 Outline of the Genetic Programming algorithm, showcasing its structural similarities with the Genetic Algorithm

```
1: Generate an initial population by recursively building random programs.
2: Execute each program and assign a fitness value to it.
3: repeat
4:     $parents \leftarrow$ selectParents($population$)    ▷ Parent selection for reproduction
5:     alter($offspring$)    ▷ Apply genetic operators to offspring
6:     $population \leftarrow$ selectSurvivors($population$, $offspring$)    ▷ Survivor selection to form next generation
7: until termination condition is met    ▷ Termination can be a fixed number of generations, or a satisfactory
   fitness level, etc.
8: return fittest($population$)    ▷ Return the best solution found
```

1.3.1 Representation and Evaluation

Representation

As with GAs, the representation of the individuals is one of the most important aspects of a GP. The representation is the encoding of potential solutions to the problem into a form that can be manipulated,[22] executed and evaluated by the algorithm.

Various methods exist for program representation, such as utilizing an abstract syntax tree, a linear sequence of instructions, a stack of instructions, or a combination of these approaches. However, the most common representation is the ***tree representation***, where the program is represented as a composite data structure like the one shown in the introduction to this section.

Let's illustrate this with an example problem: given a set of n points in the plane, find the curve that best fits the points. This is a very common problem in statistics, and it is known as the ***symbolic regression*** problem [**kozaGeneticProgrammingProgramming1992a**]. In this example, our goal is to use symbolic regression to approximate the function

$$f(x) = 5x^3 - 2x^2 + \sin(x) - 7;\ x \in [-1, 1] \tag{1.4}$$

using this function, we can generate a set of points that lies on the curve as shown in **??** and **??**.

The next step in preparing our GP setup is to define the primitive set, which includes the functions and terminals that the algorithm can use to construct candidate solutions. In this case, we will use the following set of functions and terminals:

- Functions: 1. $+$ (Addition) 2. $-$ (Subtraction) 3. $\times$ (Multiplication) 4. / (Division) 5. sin (Sine) 6. cos (Cosine) 7. pow (Power)
- Terminals: 1. x (The variable x) 2. $\{c \mid c \in [1,\, 7] \land c \in \mathbb{Z}\}$ (An ephemeral constant)[23]

Using this set of functions and terminals, we can represent the program as a tree, as shown in **??**.

[22]For example, by applying genetic operators.
[23]See **??**.

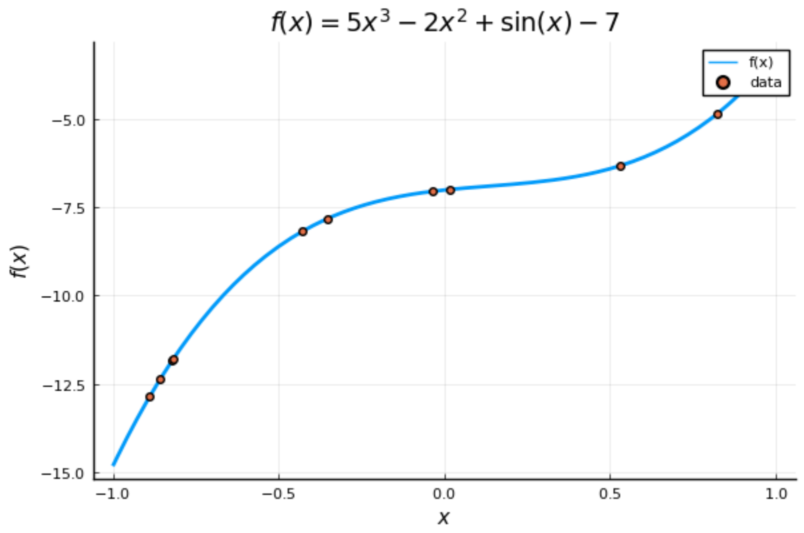


Figure 1.3: A set of points generated from the function $5x^3 - 2x^2 + \sin(x) - 7$

x	y
0.889 160	-12.872 629
0.856 103	-12.358 361
0.821 295	-11.851 004
0.818 193	-11.807 452
0.429 859	-8.183 442
0.352 328	-7.812 033
0.035 776	-7.038 557
0.017 450	-6.983 134
0.529 010	-6.314 804
0.821 101	-4.848 557

Table 1.11: A set of points generated from the function $5x^3 - 2x^2 + \sin(x) - 7$

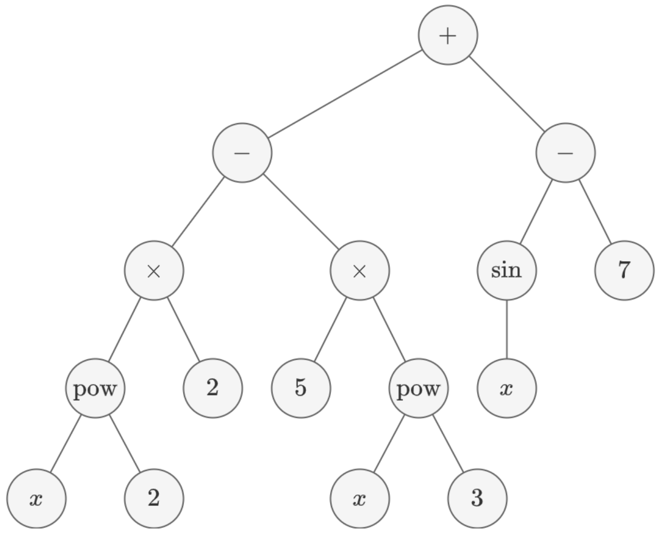


Figure 1.4: A possible tree representation of the program $5x^3 - 2x^2 + \sin(x) - 7$

Note that this definition arises the possibility of having a program that has an infinite number of nodes, as the tree can grow indefinitely. This leads the search to be unsuccessful since the probability of finding a solution is close to zero, for example, the probability of finding a solution on the initial population would be: $\lim_{x\to\infty}\frac{1}{x}=0$. To avoid this issue of potentially infinite trees, we typically impose certain ***constraints*** on the generation of the trees. The most common constraints are the ***maximum height*** of the tree and the ***maximum number of nodes*** in the tree.

Search space

Using this representation, we can define the ***genotype*** of the individuals to contain only one ***chromosome*** which is composed of a single ***gene***[24] that is the tree representation of the program. Recalling the definition of cardinality presented in **??**, we can see that the cardinality of the search space will be the number of possible trees that can be generated using the primitive set and the maximum height of the tree.

Lemma 1.1. *Let $\mathbb{T}_H$ be the set of all possible* ***labeled trees*** *of height H, with $H \in \mathbb{N}$. Given the sets $\mathcal{T}$ and $\mathcal{F}$ corresponding to the possible labels of terminal nodes (nodes that do not have children) and the possible labels of internal nodes (nodes that have children) respectively, the number of trees in $\mathbb{T}_H$ is given by the following recurrence relation:*

$$|\mathbb{T}_H(\mathcal{T},\mathcal{F})| = \begin{cases} |\mathcal{T}| & \text{if } H = 0 \\ \displaystyle\sum_{f\in\mathcal{F}} |\mathbb{T}_{H-1}(\mathcal{T},\mathcal{F})|^{A(f)} & \text{if } H > 0 \end{cases} \tag{1.5}$$

where $A(f)$ is the arity of the node f.

Proof. For the proof, we will use induction on the height of the tree. For the sake of brevity, we will use the notation $|\mathbb{T}_H(\mathcal{T},\mathcal{F})| = |\mathbb{T}_H|$.

Base case: $H = 0$ If the height of the tree is 0, then the tree is composed of a single node, which is a terminal node. Thus, the number of possible trees is equal to the number of possible terminal nodes, which arises:

$$|\mathbb{T}_0| = |\mathcal{T}|$$

Base case: $H = 1$ If the height of the tree is 1, then the tree is composed of a root node, which is an internal node, and a set of children, which are terminal nodes. Suppose that the root node has the label $f \in \mathcal{F}$ and arity $A(f)$.

Since the children are terminal nodes, each children can have any of the labels in $\mathcal{T}$. Thus, the number of trees rooted at f is equal to the number of possible combinations of $A(f)$ elements (with repetition and order) from the set $\mathcal{T}$, this is:

$$\prod_{i=1}^{A(f)} |\mathcal{T}| = |\mathcal{T}|^{A(f)}$$

Since the root node can have any of the labels in $\mathcal{F}$, the number of possible trees of height 1 is equal to:

$$|\mathbb{T}_1| = \sum_{f\in\mathcal{F}} |\mathcal{T}|^{A(f)}$$

[24]Although the most common representation is to have a single gene referencing the root of the tree, several variations that use multi-gene chromosomes have been proposed, such as Koza's *Automatically Defined Functions* [**kozaGeneticProgrammingII1994**], Angeline and Pollack's *Genetic Library Builder* [**peterjangelineEvolutionaryInductionSubroutines1992**, **peterj.angelineGeneticProgrammingEmergent1994**], or Rosca and Ballard's *Adaptive Representation* [**roscaLearningAdaptingRepresentations1994**].

Inductive step: $H > 1$ Suppose the statement holds true for $H = h$. We aim to prove that the statement also holds true for $H = h + 1$.

Since a terminal node cannot have children,[25] each tree of height $h + 1$ has a root with one of the labels from the set $\mathcal{F}$, and the remaining h layers are fully formed subtrees of height h.

For a given node label $f \in \mathcal{F}$ with arity $A(f)$, each child is the root of a subtree of height h. Given our inductive assumption, there are $|\mathbb{T}_h|$ possible such subtrees.

Since all subtrees are independent, the number of possible trees with the root f is $|\mathbb{T}_h|^{A(f)}$, which is the product of $|\mathbb{T}_h|$ over the arity of f.

We can sum this quantity over all $f \in \mathcal{F}$ to get the total number of possible trees of height $h + 1$:

$$|\mathbb{T}_{h+1}| = \sum_{f \in \mathcal{F}} |\mathbb{T}_h|^{A(f)}$$

□

Lemma 1.2. *Let $\mathbb{T}_{\leq H}$ be the set of all possible* ***labeled trees*** *of height $h \leq H$, with $H \in \mathbb{N}$ and $h \in \mathbb{N}$. Given the sets $\mathcal{T}$ and $\mathcal{F}$ corresponding to the possible labels of terminal nodes and the possible labels of internal nodes respectively, the number of trees in $\mathbb{T}_{\leq H}$ is given by the following recurrence relation:*

$$|\mathbb{T}_{\leq H}(\mathcal{T}, \mathcal{F})| = \begin{cases} |\mathcal{T}| & \text{if } H = 0 \\ |\mathbb{T}_H(\mathcal{T}, \mathcal{F})| + |\mathbb{T}_{\leq H-1}(\mathcal{T}, \mathcal{F})| & \text{if } H > 0 \end{cases} \tag{1.6}$$

Where $\mathbb{T}_H$ is the set of all possible trees of height H.

Proof. For the sake of simplicity, we will use the notation $|\mathbb{T}_{\leq H}| = |\mathbb{T}_{\leq H}(\mathcal{T}, \mathcal{F})|$ and $|\mathbb{T}_H| = |\mathbb{T}_H(\mathcal{T}, \mathcal{F})|$.

The set $\mathbb{T}_{\leq H}$ can be partitioned into two disjoint sets: the set of all possible trees of height H and the set of all possible trees of height $h < H$. Thus we have:

$$|\mathbb{T}_{\leq H}| = |\mathbb{T}_H| + |\mathbb{T}_{\leq H-1}|$$

□

Theorem 1.1. *Let $\mathbb{T}_{\leq H}$ be the set of all possible* ***labeled trees*** *of height $h \leq H$, with $H \in \mathbb{N}$ and $h \in \mathbb{N}$. Given the sets $\mathcal{T}$ and $\mathcal{F}$ corresponding to the possible labels of terminal nodes and the possible labels of internal nodes respectively, the number of trees in $\mathbb{T}_{\leq H}$ is given by the following recurrence relation:*

$$|\mathbb{T}_{\leq H}(\mathcal{T}, \mathcal{F})| = \begin{cases} |\mathcal{T}| & \text{if } H = 0 \\ \left(\sum_{h=0}^{H-1} \sum_{f \in \mathcal{F}} |\mathbb{T}_h(\mathcal{T}, \mathcal{F})|^{A(f)} \right) + |\mathcal{T}| & \text{if } H > 0 \end{cases} \tag{1.7}$$

Where $\mathbb{T}_H$ is the set of all possible trees of height H and $A(f)$ is the arity of the node f.

Proof. From **??** we know the number of trees of height H or less is given by:

$$|\mathbb{T}_{\leq H}| = |\mathbb{T}_H| + |\mathbb{T}_{\leq H-1}|$$

Then, by applying **??**, we get:

[25] This could also be interpreted as a terminal node having an arity of 0, or that all terminal nodes are leaves.

$$|\mathbb{T}_{\leq H}| = \left(\sum_{f \in \mathcal{F}} |\mathbb{T}_{H-1}|^{A(f)} \right) + |\mathbb{T}_{\leq H-1}|$$

By unrolling the recurrence relation, we get:

$$\begin{aligned}
|\mathbb{T}_{\leq H}| &= \left(\sum_{f \in \mathcal{F}} |\mathbb{T}_{H-1}|^{A(f)} \right) + |\mathbb{T}_{\leq H-1}| \\
&= \left(\sum_{f \in \mathcal{F}} |\mathbb{T}_{H-1}|^{A(f)} \right) + \left(\sum_{f \in \mathcal{F}} |\mathbb{T}_{H-2}|^{A(f)} \right) + \cdots + \left(\sum_{f \in \mathcal{F}} |\mathbb{T}_{1}|^{A(f)} \right) + \left(\sum_{f \in \mathcal{F}} |\mathbb{T}_{0}|^{A(f)} \right) + |\mathbb{T}_0| \\
&= \left(\sum_{h=0}^{H-1} \sum_{f \in \mathcal{F}} \mathbb{T}_h(\mathcal{T}, \mathcal{F})|^{A(f)} \right) + |\mathcal{T}|
\end{aligned}$$

□

With this result, we can now calculate the cardinality of the search space of the genetic programming algorithm for a given maximum height H, since a program can be seen as a ***labeled tree***. Then, given the set $\mathcal{T} = \{x, c\}$ and the set $\mathcal{F} = \{+, -, \times, /, \sin, \cos, \exp, \log\}$, where

$$A(f) = \begin{cases} 2 & \text{if } f \in \{+, -, \times, /, \text{pow}\} \\ 1 & \text{if } f \in \{\sin, \cos\} \end{cases}$$

Given that the height of the AST of the target program is 4, we can use 5 as the maximum height of the programs in the search space (to allow a little of "breathing room" to the generated programs). Noting that, since c can be one of 7 possible values, $|\mathcal{T}| = 8$, we can calculate the cardinality of the search space as follows:

$$\begin{aligned}
|\mathcal{S}| = |\mathbb{T}_{\leq 5}(\mathcal{T}, \mathcal{F})| &= \left(\sum_{h=0}^{4} \sum_{f \in \mathcal{F}} |\mathbb{T}_h(\mathcal{T}, \mathcal{F})|^{A(f)} \right) + 8 \\
&\approx 9.531\,142 \times 10^{37} \\
&\approx 10^{38}
\end{aligned}$$

Thus, the search space of the genetic programming algorithm is of the order of 10^{38} programs.[26] It should be easy to see that the size of the search space make it unfeasible to perform an exhaustive search. This is why we need to use a heuristic search algorithm, such as genetic programming. In **??** we can see how the number of trees of height less or equal to h rapidly increases as h increases.

Evaluation

Now, we need to define how to evaluate the fitness of a program. Again, there are many ways to do this, but a common way is to use the ***mean squared error*** (MSE) between the points and the program.

Definition 1.8 (Mean Squared Error)**.** *If a vector of n predictions is generated from a sample of n data points on all variables, and $\mathbf{y}_i$ is the i-th observed value and $\hat{\mathbf{y}}_i$ is the i-th prediction, then the MSE of the predictor is a function* MSE : $\mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ *defined as:*

[26] This value was computed using the script shown in **??**.

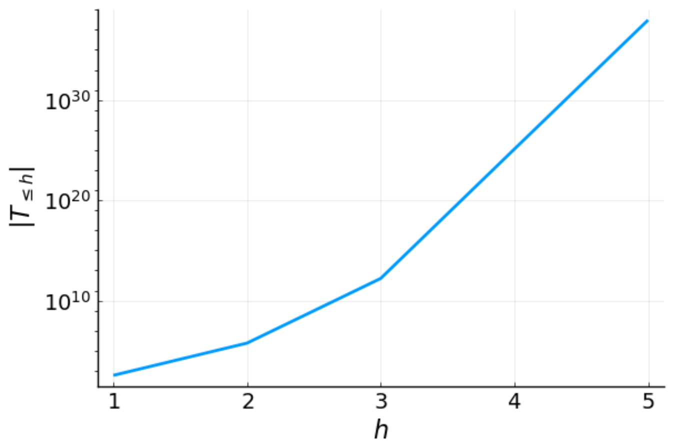


Figure 1.5: Total number of trees of height less or equal to h for $h \in \{0, \dots, 5\}$ and $\mathcal{T} = \{x, c\}$ and $\mathcal{F} = \{+, -, \times, /, \sin, \cos, \mathrm{pow}\}$. Note that the Y axis is in logarithmic scale.

$$\mathrm{MSE}(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{n}\sum_{i=1}^{n}(\mathbf{y}_i - \hat{\mathbf{y}}_i)^2 \tag{1.8}$$

The MSE is a common measure of the quality of an estimator, used in many *machine learning* problems.

In our case, we will use the MSE to evaluate the fitness of a program. Consider a program P and two sets of points, $\mathbf{x}$ and $\mathbf{y}$, as outlined in **??**. Suppose also that $\mathsf{P}[\mathbf{x}]$ is the set of points generated by evaluating P on the points of $\mathbf{x}$, and that $\mathsf{P}(x)$ is the result of evaluating P on the point x. Then, we can define the fitness of P as:

$$\phi_{\mathsf{P}} = \mathrm{MSE}(\mathbf{y}, \mathsf{P}[\mathbf{x}]) = \frac{1}{n}\sum_{i=1}^{n}(\mathbf{y}_i - \mathsf{P}(\mathbf{x}_i))^2 \tag{1.9}$$

This section elucidated the pivotal aspects of Genetic Programming (GP), focusing on representation of individuals and their evaluation. Individuals, potential solutions to a problem, can be encoded in several ways including tree representation. A common statistical problem, symbolic regression, was illustrated using a set of points on a curve from a function involving both polynomial and trigonometric elements. By defining a set of functions and terminals, these points were represented as a tree. The fitness of an individual program was assessed using the Mean Squared Error (MSE) between the points on the curve generated by the function and the points produced by the program. Thus, in the context of this problem, the fittest individual is the one that minimizes this error, i.e., best fits the curve. The methods described here underscore the versatility and applicability of GP to various types of problems. This method of representation and evaluation provides a robust basis for generating evolving populations of programs. In the next sections we will discuss the mechanisms for generating these populations and the evolutionary operators that act on them.

1.3.2 Initialization

As with other evolutionary algorithms, the algorithm starts by generating a population of random individuals.

There are many ways to generate random individuals, but a very common (and simple) method is the ***grow method*** [**kozaGeneticProgrammingProgramming1992a**]. In this method, a maximum height is defined, and the algorithm then creates a random tree with a given minimum height and a maximum height.

Remark. *A tree with a height of 0 is a tree with only one node, the root.*

The grow method then proceeds to recursively generate the trees with random nodes until a terminal node is selected or the maximum height is reached. This method is shown in **??**.

Algorithm 3 The grow method for generating random trees

```
Require: $l \in \mathbb{N}$, $h \in \mathbb{N}$
Require: $l \leq h$
Require: a random integer $n$ such that $l \leq n \leq h$
Require: $\mathbf{t}$ and $\mathbf{f}$ are the sets of terminal and function nodes respectively, where $\mathbf{t} \neq \emptyset$ and $\mathbf{f} \neq \emptyset$
Ensure: a random tree with a height between $l$ and $h$
 1: function grow($\mathbf{t}$, $\mathbf{f}$, $d$)
 2:     $c \leftarrow \emptyset$
 3:     if $d = n \lor \left(d \geq l \land \text{random}() < \frac{|\text{t}|}{|\text{t}| + |\text{f}|}\right)$ then
 4:         return a random node from $\mathbf{t}$
 5:     else
 6:         $f \leftarrow$ a random node from $\mathbf{f}$
 7:         for $i$ in $1 \ldots \text{arity}(f)$ do
 8:             $c_i \leftarrow \text{grow}(\mathbf{t}, \mathbf{f}, d+1)$
 9:             $c \leftarrow c \cup \{c_i\}$
10:         end for
11:         return a tree with root $f$ and children $c$
12:     end if
13: end function
```

With this method, the algorithm can generate trees where the size of the longest path from the root to a leaf is a number $n \in [l, h]$.

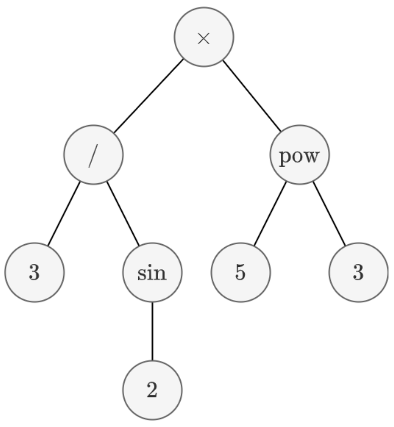
Now that the algorithm can generate random trees, it can generate a random population of trees by generating a random tree for each individual in the population. Assuming a population size of $p = 4$, a maximum height of $h = 3$, a minimum height of $l = 1$, and the primitives set defined in the previous section, the algorithm could generate the population: $\mathbf{P} = \{\mathbf{I}_1, \mathbf{I}_2, \mathbf{I}_3, \mathbf{I}_4\} = \left\{\frac{3}{\sin(2)} \times 5^3,\ 7 - (5 + \sin(x)),\ 7 + 2,\ 5x^2\right\}$, as shown in **??**.

The next step is to calculate the fitness of each individual in the population. If we recall, the fitness function is the MSE between the expected output and the actual output of the individual.

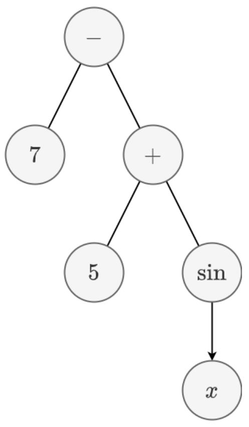
$$
\begin{aligned}
\text{MSE}(\mathbf{y}, \mathbf{I}_1[\mathbf{x}]) &= \frac{1}{10}\sum_{i=1}^{10}(\mathbf{y}_i - \mathbf{I}_1(\mathbf{x}_i))^2 = \frac{1}{10}\sum_{i=1}^{10}\left(\mathbf{y}_i - \frac{3}{\sin(2)} \cdot 5^3\right)^2 \\
&\approx 177\,596.851\,131 \\
\text{MSE}(\mathbf{y}, \mathbf{I}_2[\mathbf{x}]) &= \frac{1}{10}\sum_{i=1}^{10}(\mathbf{y}_i - \mathbf{I}_2(\mathbf{x}_i))^2 = \frac{1}{10}\sum_{i=1}^{10}\left(\mathbf{y}_i - 7 - (5 + \sin(x_i))\right)^2 \\
&\approx 137.398\,836 \\
\text{MSE}(\mathbf{y}, \mathbf{I}_3[\mathbf{x}]) &= \frac{1}{10}\sum_{i=1}^{10}(\mathbf{y}_i - \mathbf{I}_3(\mathbf{x}_i))^2 = \frac{1}{10}\sum_{i=1}^{10}\left(\mathbf{y}_i - (7 + 2)\right)^2 \\
&\approx 331.924\,267 \\
\text{MSE}(\mathbf{y}, \mathbf{I}_4[\mathbf{x}]) &= \frac{1}{10}\sum_{i=1}^{10}(\mathbf{y}_i - \mathbf{I}_4(\mathbf{x}_i))^2 = \frac{1}{10}\sum_{i=1}^{10}\left(\mathbf{y}_i - 5x^2\right)^2 \\
&\approx 138.079\,865
\end{aligned}
$$

With this, we can assign a fitness to each individual in the population as shown in **??**. A summary of the population's fitness is shown in **??**.

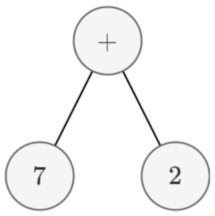
We can observe that the worst individual has an error significantly larger than the best individual. This is to be



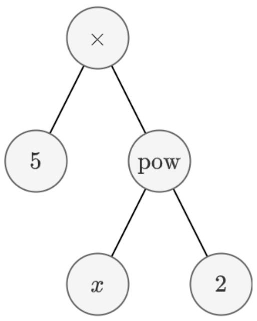
(a) $\mathbf{I}_1 = \frac{3}{\sin(2)} \times 5^3$



(b) $\mathbf{I}_2 = 7 - (5 + \sin(x))$



(c) $\mathbf{I}_3 = 7 + 2$



(d) $\mathbf{I}_4 = 5x^2$

Figure 1.6: A population of random trees.

Generation 0

Individual	Program	Height	Fitness
$\mathbf{I}_1(x)$	$\frac{3}{\sin(2)} \cdot 5^3$	3	177 596.851 131
$\mathbf{I}_2(x)$	$7 - (5 + \sin(x))$	3	137.398 835
$\mathbf{I}_3(x)$	$7 + 2$	1	331.924 267
$\mathbf{I}_4(x)$	$5x^2$	2	138.079 865

Table 1.12: Initial population of the genetic programming algorithm

	Fitness	Individual
Best	137.398 835	$\mathbf{I}_2(x)$
Worst	177 596.851 131	$\mathbf{I}_1(x)$
Average	44 551.063 525	
Standard deviation	76 814.062 197	

Table 1.13: Fitness of the individuals in generation 0

expected, as the MSE is a measure of the error that escalates exponentially with the difference between the expected and actual output.

A graphical representation of the population is shown in **??**. It is clear from the figure that the worst individual is $\mathbf{I}_1$.

In summary, the initialization stage of the genetic programming algorithm, a population of random individuals,

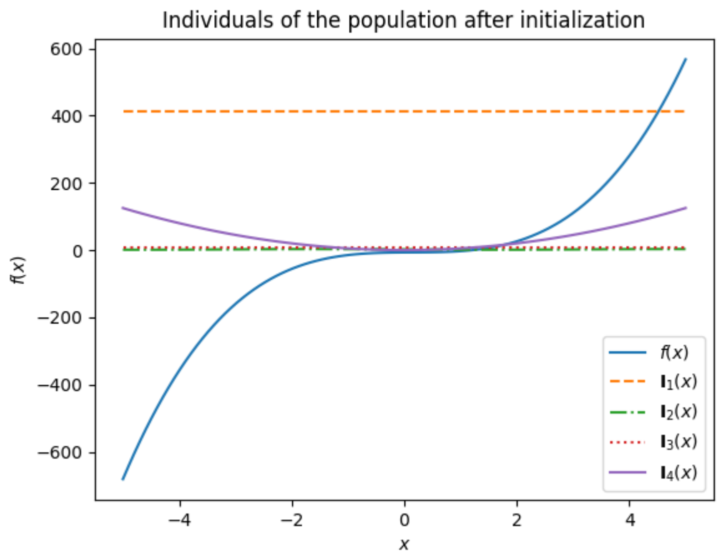


Figure 1.7: Graphical representation of the population in generation 0 compared to the expected output.

represented as trees, was generated using the "grow method". The height of these trees, signifying the length of the longest path from the root to a leaf, is determined randomly within a specified range. Each tree is recursively populated with random nodes until a terminal node is chosen or the maximum height is reached. Once a random population of trees is created, the fitness of each individual, based on the Mean Squared Error (MSE) between the expected and actual output, is calculated. This results in the assignment of fitness scores to all individuals, facilitating the evaluation and selection process in the succeeding stages of the algorithm. The output of this process is a random population of trees with fitness scores and height measurements, ready for the next steps of selection, crossover, and mutation.

1.3.3 Selection

The selection process in Genetic Programming (GP) is similar to that of Genetic Algorithms (GA). However, unique modifications have been proposed for the standard process in GP, which consider the semantics, or the functional behavior, of the programs being evolved [**liskowskiComparisonSemanticawareSelection2015**]. This work will not delve into this topic, as it is beyond the scope of this document.

For the problem under study, we use a selection process similar to the one used in the *One Max* problem (a simple optimization problem where the objective is to maximize the number of ones in a binary string). The equation for calculating the selection probability will diverge from **??**, as it presumes the fittest individual is the one with the highest fitness value. In contrast, the symbolic regression problem, which is our focus, considers the individual with the lowest fitness value as the fittest.

For this particular case of the symbolic regression problem, we adjust our approach to selection. We introduce a ***corrected fitness function***, ϕ', defined as:

$$\phi'(I) = \left(\sum \Phi_{\mathbf{P}}\right) - \phi_I \tag{1.10}$$

Here, ϕ_I signifies the fitness of individual I, and $\Phi_{\mathbf{P}}$ represents the *batch fitness function* defined in **??** applied to the population $\mathbf{P}$. We then define the selection probability for an individual $\mathbf{P}_i$ as:

$$p_i = \frac{\phi'(\mathbf{P}_i)}{\sum_{j=1}^{N} \phi'(\mathbf{P}_j)} \tag{1.11}$$

In this equation, N stands for the size of the population.

With this methodology, we calculate the selection probabilities for the population as illustrated in **??**. The outcome shows that the individual with the highest error has a considerably low probability of being selected, while the other individuals have roughly equal chances.

Individual	**Fitness**	**Selection probability**
$\mathbf{I}_1(x)$	607.402 968	0.113 615%
$\mathbf{I}_2(x)$	178 066.855 263	33.307 633%
$\mathbf{I}_3(x)$	177 872.329 832	33.271 246%
$\mathbf{I}_4(x)$	178 066.174 234	33.307 505%

Table 1.14: Selection probabilities for the symbolic regression problem.

Assuming a ***survival rate*** of 50%, let's consider that the selection process favors I_2 and I_3 as survivors due to their higher selection probabilities (as shown in the previous table). In this scenario, I_1 and I_4 are identified as the ones to be replaced by the offspring in the next generation. A comparison between the survivors and the target function is shown in **??**.

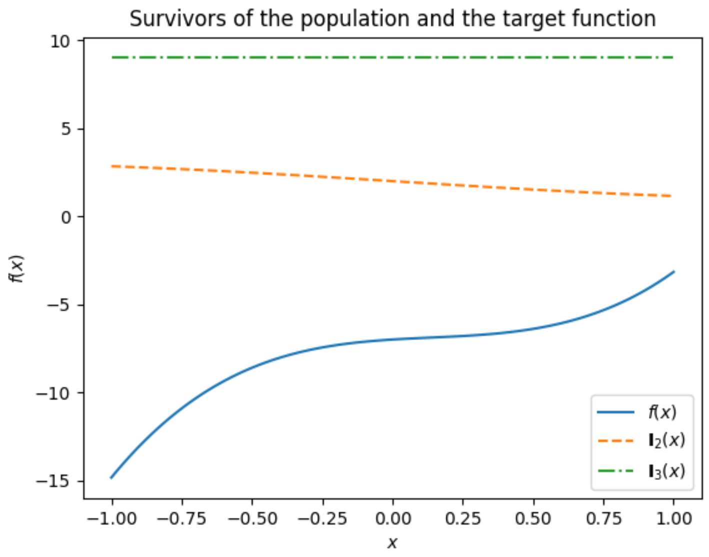


Figure 1.8: Comparison between the survivors and the target function.

To conclude, this section discussed the selection process in Genetic Programming (GP), noting that it largely mirrors that of Genetic Algorithms (GA) with some differences. These differences primarily focus on the semantics of the programs being evolved. The symbolic regression problem, necessitates a corrected fitness function and an adjusted selection probability equation. We outlined how these modifications are implemented and computed selection probabilities for a hypothetical population. A survival rate of 50% resulted in two individuals being selected as survivors and two being marked for replacement in the next generation. This lays the groundwork for the next phase of the process, which is variation.

1.3.4 Variation

Analogous to GA, the individuals in a GP population undergo a variation process. For the symbolic regression problem under consideration, we introduce variation using two specific operators: ***crossover*** and ***mutation***.

Crossover

Variation operators in Genetic Programming (GP) must maintain the syntactic correctness of the programs or individuals. For crossover, this implies that the resultant offspring must be syntactically correct programs.

The crossover operator used in Genetic Algorithms (GAs), described in **??**, is not typically suitable for GP, as it does not guarantee the syntactic correctness of the resulting individuals. Though there could be instances where the crossover operator used in GAs is applicable to GP, as depicted in **??**, these are not common scenarios.

The choice of operator in GP depends on the representation of the individuals.

For tree-based GP, the fundamental crossover operator is the ***subtree crossover***, referenced in **??**.

This operator selects a random node from each parent and exchanges the subtrees rooted at these nodes. Usually, a constraint similar to the one used for generating the initial population is applied to this operator to prevent the creation of overly large trees.

Assuming we select two individuals, $\mathbf{I}_1$ and $\mathbf{I}_2$, from the population, the subtree crossover operator chooses a random node from each individual, say $\clubsuit = 7$ from $\mathbf{I}_1$ and $\diamondsuit = x^2$ from $\mathbf{I}_2$. The subtrees rooted at these nodes are then interchanged, as shown below:

$$\begin{aligned} & \chi(\mathbf{I}_2, \mathbf{I}_4) && = \chi(\clubsuit - (5 + \sin(x)),\, 5\diamondsuit) \\ & && = (\diamondsuit - (5 + \sin(x)),\, 5\clubsuit) \\ \Leftrightarrow \quad & (\mathbf{O}_1, \mathbf{O}_2) && = (x^2 - (5 + \sin(x)),\, 5 \cdot 7) \end{aligned}$$

where χ signifies the subtree crossover operator between two individuals. The crossover of $\mathbf{I}_1$ and $\mathbf{I}_2$ is depicted in **??**.

Following the application of the subtree crossover operator, the fitness of the individuals in the population is evaluated as shown in **??**. A summary of the population's fitness is given in **??**.

Generation 1

Individual	Program	Fitness
$\mathbf{I}_2$	$7 - (5 + \sin(x))$	137.398 836
$\mathbf{I}_3$	$7 + 2$	331.924 267
$\mathbf{O}_1$	$5 \cdot 7$	1944.288 127
$\mathbf{O}_2$	$x^2 - (5 + \sin(x))$	33.740 766

Table 1.15: Population after applying the subtree crossover operator.

	Fitness	Individual
Best	33.740 766	O_2
Worst	1944.288 127	O_1
Average	611.837 999	
Standard deviation	896.858 214	

Table 1.16: Fitness summary of the population after applying the subtree crossover operator.

A notable improvement in the population's fitness is observed after the application of the subtree crossover operator. Comparing the results from **??**, we find that the average fitness (or error) has dropped from 44 551.063 525 to 611.837 999, equating to an improvement of approximately 98.627%.

$$\frac{\bar{\Phi}_i - \bar{\Phi}_X}{\bar{\Phi}_i} = \frac{44\,551.063\,525 - 611.837\,999}{44\,551.063\,525} \approx 98.627\%$$

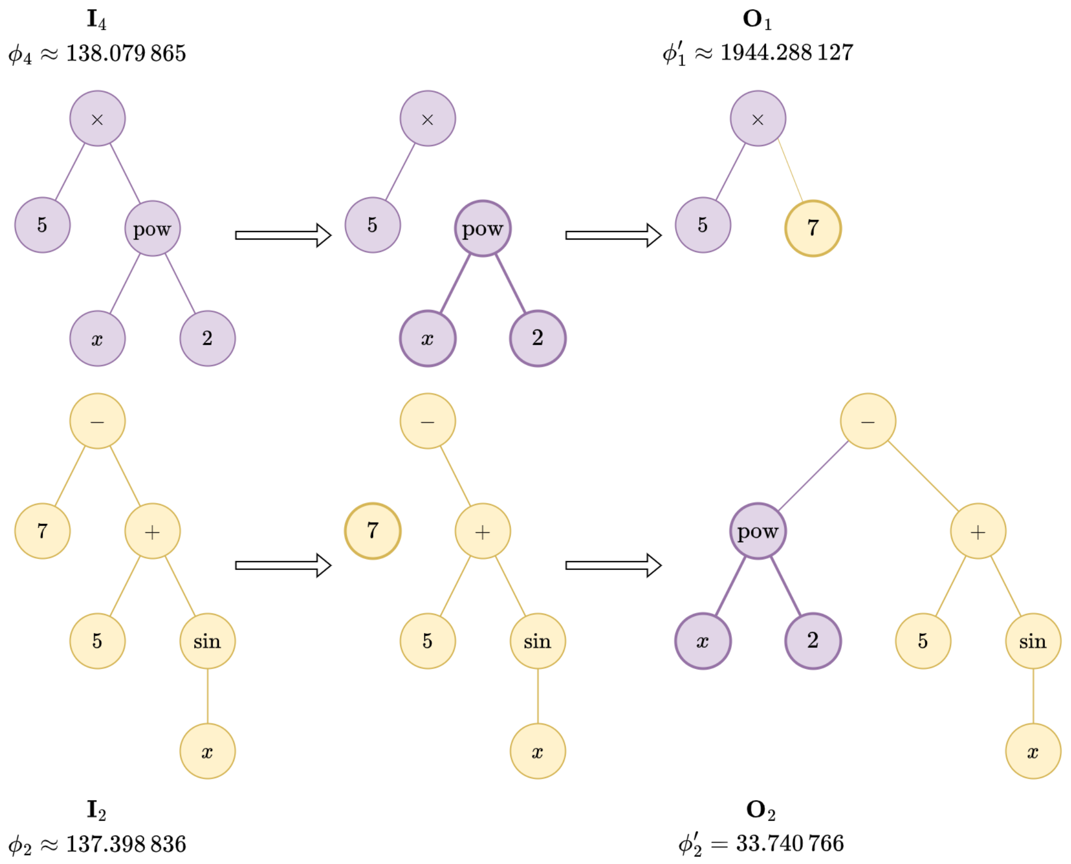


Figure 1.9: Crossing over of $\mathbf{I}_2 = 7 - (5 + \sin(x))$ and $\mathbf{I}_4 = 5x^2$, producing $\mathbf{O}_1 = x^2 - (5 + \sin(x))$ and $\mathbf{O}_2 = 5 \cdot 7$.

where $\bar{\Phi}_i$ is the average fitness of the population after initialization and $\bar{\Phi}_X$ is the average fitness of the population after applying the subtree crossover operator.

A reduction in the population's fitness standard deviation, from 88 697.238 974 to 896.858 214, is also seen. This decrease of around 98.989% indicates that the population's diversity has reduced.

$$\frac{\sigma_i - \sigma_X}{\sigma_i} = \frac{88\,697.238\,974 - 896.858\,214}{88\,697.238\,974} \approx 98.989\%$$

where σ_i is the standard deviation of the fitness of the population after initialization and σ_X is the standard deviation of the fitness of the population after applying the subtree crossover operator.

Diversity here is a measure of population's dispersion, and a decline in diversity may signal that the population is not converging towards a solution (too much diversity) or is converging prematurely (too little diversity).

In the following section, we discuss the mutation operator, which can be used to introduce diversity into the population, thereby preventing premature convergence.

Mutation

Mutation, another crucial genetic operator, introduces new genetic material into the population, thus maintaining genetic diversity and preventing premature convergence to suboptimal solutions. In the context of GP, the mutation operator modifies a program in the population, while ensuring the resultant individual's syntactic correctness.

In tree-based GP, a common form of mutation is the ***point mutation*** [**poliFieldGuideGenetic2008a**, **wilhelmstotterJenetic** which selects a random node from an individual and replaces it with a random primitive with the same arity. This operator is similar to the bit-flip mutation operator used on **??**. As with bit-flip mutation, point mutation can also be applied with a certain probability to each node in an individual, meaning that more than one node can be mutated in a single individual.

Suppose we mutate all the individuals in the population resulting from the crossover operation in **??**, and that exactly one node is mutated in each individual. Let the selected nodes be $\clubsuit = \sin$ in $\mathbf{I}_2$, $\spadesuit = 7$ in $\mathbf{I}_3$, $\heartsuit = 5$ in $\mathbf{O}_1$, and $\diamondsuit = 2$ in $\mathbf{O}_2$. Then, a possible result of applying the point mutation operator can be:

$$M\left(\begin{bmatrix} I_2 \\ I_3 \\ O_1 \\ O_2 \end{bmatrix}\right) = M\left(\begin{bmatrix} 7-(5+\clubsuit(x)) \\ \spadesuit + 2 \\ \heartsuit \cdot 7 \\ x^{\diamondsuit} - (5+\sin(x)) \end{bmatrix}\right) = \left(\begin{bmatrix} 7-(5+\clubsuit'(x)) \\ \spadesuit' + 2 \\ \heartsuit' \cdot 7 \\ x^{\diamondsuit'} - (5+\sin(x)) \end{bmatrix}\right) = \left(\begin{bmatrix} 7-(5+\cos(x)) \\ 6 + 2 \\ 6 \cdot 7 \\ x^{3} - (5+\sin(x)) \end{bmatrix}\right)$$

Here, $\clubsuit'$, $\spadesuit'$, $\heartsuit'$, and $\diamondsuit'$ are random primitives with the same arity as $\clubsuit$, $\spadesuit$, $\heartsuit$, and $\diamondsuit$, respectively. That being, $\clubsuit' = \cos$, $\spadesuit' = 6$, $\heartsuit' = 6$, and $\diamondsuit' = 3$.

The fitness of the individuals in the population after applying the subtree mutation operator is then evaluated. The results of this fitness evaluation are shown in **??**. A summary of the fitness of the population is presented in **??**.

Generation 1

Individual	**Program**	**Fitness**
$M(\mathbf{I}_2)$	$7-(5+\sin(x))$	137.398 836
$\mathbf{I}_3$	$7+2$	331.924 267
$\mathbf{O}_1$	$5 \cdot 7$	1944.288 127
$\mathbf{O}_2$	$x^2-(5+\sin(x))$	33.740 766

Table 1.17: Population after applying the subtree crossover operator.

	Fitness	**Individual**
Best	33.740 766	O_2
Worst	1944.288 127	O_1
Average	611.837 999	
Standard deviation	896.858 214	

Table 1.18: Fitness summary of the population after applying the subtree crossover operator.

Just like the crossover operator, mutation can also significantly influence the fitness and diversity of the population. By generating new structures in the population, mutation can help prevent stagnation and maintain diversity, thus avoiding premature convergence to suboptimal solutions.

In the next section, we discuss the combination of crossover and mutation operators and their role in navigating the search space effectively.

Chapter 2

Relevant Work (State of the Art)

In this chapter, we will explore the current state of the art in the field of genetic algorithm frameworks. While the theory of genetic algorithms is not a recent development, the application of these algorithms continues to evolve[1] with the advancement of various programming languages and tools. Therefore, our focus in this chapter will be on current and actively developed frameworks widely used for studying and implementing genetic algorithms.

Specifically, we will look into Agile Artificial Intelligence in Pharo, DEAP, Jenetics, ECJ, and GeneticSharp. Each section will provide a brief overview of the framework along with basic code samples to highlight their syntax and outline their unique features and differences. We believe this approach will provide a comprehensive perspective on the versatility and diversity of tools available in this field.

This state-of-the-art review does not only offer insights into the current trends and tools in genetic algorithms but also sets the stage for our contribution - a genetic algorithm framework in Kotlin.

2.1 The One Max Problem

For the purpose of illustrating the use of the different frameworks, we will use the *One Max problem* introduced on **??**.

The ***One Max problem***, or the Ones Counting problem, is a classic and straightforward optimization problem often used as a benchmark in the study of evolutionary algorithms and other heuristic search methods. It serves as a deceptively simple yet effective test of an algorithm's optimization ability.

Given a binary string of length n, the *One Max problem* is to find a binary string such that the sum of its bits (counting the number of ones) is maximized. In formal terms, if we denote the binary string as $x = (x_1, x_2, ..., x_n)$ where each $x_i \in \{0, 1\}$ for all $i = 1, 2, \ldots, n$, the *fitness function* $\phi(x)$ to be maximized can be expressed as:

$$\phi(x) = \sum_{i=1}^{n} x_i \tag{2.1}$$

The function $f(x)$ counts the number of ones in the string x. The maximum possible value of $f(x)$ is n, which is achieved when all bits in the string are one. The *One Max problem* is an instance of a unimodal problem since there's only one local maximum which is also a global maximum.

It's important to note that despite its simplicity, the *One Max problem* does provide a non-trivial task for many search algorithms. For a binary string of length n, there are 2^n possible solutions. For larger n, an exhaustive search of the solution space is not feasible, hence the need for efficient optimization algorithms.

[1]No pun intended.

Due to its characteristics, the *One Max problem* is often used to evaluate the performance of optimization algorithms especially genetic and evolutionary algorithms. It's particularly well-suited for genetic algorithms as the operations of crossover and mutation can directly change the number of ones in a binary string, thus impacting the fitness of a potential solution.

Despite the straightforward objective function, the *One Max problem* is invaluable in the study of heuristic search methods due to its accessibility, simplicity and the vastness of its search space, which permits the analysis and comparison of the performance of different optimization techniques.

2.2 Agile Artificial Intelligence in Pharo

This is the second paragraph. Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

And after the second paragraph follows the third paragraph. Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

After this fourth paragraph, we start a new paragraph sequence. Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

This is the second paragraph. Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

2.3 Distributed Evolutionary Algorithms in *Python* (DEAP)

Distributed Evolutionary Algorithms in Python (DEAP) [**DEAPDocumentationDEAP**] is a powerful evolutionary computation framework designed for rapid prototyping and validation of concepts. It stands apart from many other evolutionary computation libraries due to its significant modularity and versatility, which enables the construction of a broad range of evolutionary algorithms, genetic algorithms, and even hybrid algorithms. DEAP is open-source and available under the *GNU Lesser General Public License v3.0* (LGPL-3.0) [**GNULesserGeneral**].

DEAP is structured around two primary components: the `Creators` and the `Toolbox`. The `Creators` module facilitates the generation of new classes integral to the genetic algorithm, such as individuals and populations.