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
2024-01-30
`v0.5.2310-2`

Abstract

This thesis explores Evolutionary Computation (EC) within Artificial Intelligence, particularly focusing on its application in software engineering and scientific computing. It introduces a Kotlin-based EC framework, highlighting its modularity and ability to integrate various EC algorithms. The framework's efficacy is demonstrated through case studies on Genetic Algorithms and Linear Genetic Programming, emphasizing its ability to solve complex computational problems.

The development is grounded in a detailed review of existing EC frameworks, with an emphasis on their functionalities and architectural details. Additionally, the thesis explores the theoretical dimensions of EC, providing a comprehensive analysis of its fundamental concepts and methodologies. This study enriches the understanding of EC and contributes to academic discussions, establishing a firm foundation for future research and applications.

Central to the thesis is the hypothesis that Kotlin, with its expressive syntax and cross-platform capabilities, can enhance evolutionary algorithms. This hypothesis is examined through various research questions, focusing on the essentials for an efficient genetic algorithm framework, leveraging Kotlin's unique features, and applying the framework to real-world problems.

The structure of the thesis is methodically laid out as follows:

1. **Background**: Establishes the theoretical foundation for the central concepts.
2. **State of the Art**: Reviews current advancements and frameworks in EC.
3. **Framework Design and Implementation**: Details the architecture and implementation of the Kotlin-based framework.
4. **Function Optimization Case Study**: Tests the framework with real-world function optimization problems.
5. **Crash Reproduction Problem Case Study**: Demonstrates the framework's application in another domain.
6. **Conclusion and Future Work**: Summarizes the work and proposes future research directions.

This structure ensures a comprehensive understanding of the research's objectives, methodology, and outcomes, providing a coherent narrative throughout the thesis.

Contents

Chapter 1

Introduction

1.1 Motivation

The inception of artificial intelligence (AI) as we perceive it today is deeply rooted in the mid-20th century when Alan Turing proposed the intriguing question, "Can machines think?" [1]. This seminal query catalyzed a remarkable evolution in the field, leading to the emergence of a diverse spectrum of AI applications that persistently extend and advance.

Among the early endeavors to emulate evolution artificially, Nills Aall Barricelli's initiative in 1954 stands out. Barricelli construed evolution as a purely statistical process, setting a novel precedent for the field of evolutionary computation (EC) [3]. This groundbreaking exploration established the parameters for an evolutionary system, which necessitated the system's components to exhibit capabilities of reproduction and mutation, thereby facilitating evolution through natural selection or the survival of the fittest.

Expanding upon these foundational principles, John Holland's influential book, ***Adaptation in Natural and Artificial Systems*** [6], unveiled genetic algorithms (GA). This innovative methodology for addressing optimization problems drew inspiration from natural selection and genetic mechanisms. Holland's work solidified the field of evolutionary algorithms (EA), igniting the development of a plethora of techniques and algorithms, including but not limited to, genetic algorithms, genetic programming, and evolutionary strategies.

Despite the proven efficacy of evolutionary algorithms in resolving intricate optimization problems ranging from scheduling and data mining to machine learning and beyond, their execution can be challenging, frequently demanding extensive, repetitive coding. As the uptake of AI continues to soar, the requirement for streamlined, intuitive libraries and frameworks that empower efficient experimentation and application of these potent algorithms grows concomitantly.

The *Kotlin* programming language emerges as a promising platform for developing such a framework [67]. *Kotlin* distinguishes itself with its lucid syntax, static typing, and flawless interoperability with Java, making it an attractive choice for AI and EC enthusiasts [55, 56, 68]. The significance of *Kotlin* has been formally acknowledged in *Android* app development [53], and it continues to gain popularity among developers, with over 60% of *Android* developers now using it [51]. Furthermore, according to a *GitHub* study, *Kotlin* emerged as the ninth fastest-growing language worldwide from 2021 to 2023, overtaking *Python* by 0.4% [75].

The allure of *Kotlin* is further amplified by its multiplatform feature, which permits code sharing across a wide range of platforms, including JVM, *JavaScript*, *Android*, *iOS*, and native desktop applications. This cross-platform compatibility promotes enhanced productivity and consistency in software development.

Kotlin offers support for functional programming (FP) and coroutines, significantly easing the development of asynchronous and concurrent applications. This feature is particularly advantageous in the context of evolutionary algorithms, which frequently require the parallel execution of multiple processes. While Kotlin currently lacks certain FP features like pattern matching, plans are in place to introduce these enhancements in future versions, further bolstering its capabilities for evolutionary computation. Moreover, *Kotlin*'s capabilities align well with the design of *Domain Specific Languages* (DSLs) [76], which can be used to concisely and intuitively define the problem domain.

Given these compelling features, this thesis aims to exploit the prowess of *Kotlin* to architect and implement a pioneering genetic algorithms framework, thereby augmenting the ever-growing application of *Kotlin* within the AI community. We expect this framework to stimulate research and applications in the field, hastening the creation of novel algorithms and techniques.

1.2 Hypothesis and Research Questions

1.2.1 Hypothesis

We propose that developing an evolutionary algorithms framework focused on extensibility and modularity, could contribute positively to the field of evolutionary computation. With a focus on modularity and extensibility, it is expected to offer a user-friendly and adaptable environment. It may aid in simplifying evolutionary algorithm implementations, potentially leading to more efficient research and innovation in the field.

Additionally, we hypothesize that creating a generalized crossover algorithm adaptable to multiple input and output sizes will further enrich the framework's capabilities. Moreover, we believe that integrating advanced and extensible monitoring tools and robust, expressive data validation mechanisms will accelerate evolutionary algorithm development by effectively separating and managing the algorithm's core functions from its monitoring and validation aspects.

1.2.2 Research Questions

To assess the validity of our hypothesis and further investigate the potential benefits of the proposed *Kotlin*-based framework, we outline the following pivotal research questions:

1. What constitute the *fundamental requirements* for a versatile, efficient, and user-friendly evolutionary algorithms framework?
2. How can the features of a language with a *rich type system* –such as static typing, type inference, generics, and null safety– be leveraged to fulfill these requirements and facilitate the development and use of evolutionary algorithms?
3. How does the *ease of use* of the framework compare to that of existing solutions, particularly in terms of syntactical complexity?
4. In what ways can the proposed framework be applied to *address complex real-world problems*, thereby demonstrating its practical viability and potential to contribute to advancements in AI and evolutionary computation?
5. What potential opportunities exist for the *enhancement and expansion* of the proposed evolutionary algorithm framework in the future?

These research questions serve as guideposts for our exploration into the feasibility and potential impact of focusing on extensibility and modularity to develop a novel evolutionary algorithms framework. Each question targets a specific facet of the project, offering a holistic understanding of the technical intricacies, practical applications, and future prospects of our proposed framework.

1.3 Objectives

The primary objectives of this thesis are set forth to establish a solid foundation for the design, implementation, and evaluation of a novel, efficient, and user-friendly genetic algorithms framework using the *Kotlin* programming language.

1.3.1 Main Objective

The main objective of this study is to develop a *Kotlin*-based genetic algorithms framework that is versatile, efficient, and user-friendly. The framework should be able to support classical GAs, as seen in section 2.2 on page 6, and be easily extensible to support future enhancements and expansions such as new algorithms, like GP, and genetic operators.

1.3.2 Specific Objectives

Building upon the primary objectives, the following specific objectives have been established to guide this research:

1. **Language Feature Utilization:** Examine and utilize *Kotlin*'s unique language features, such as its expressive syntax, static typing, and its ease of developing DSLs, to design and implement a genetic algorithms framework. This exploration should enhance the design and implementation of the genetic algorithms framework, ensuring it is efficient, robust, and user-friendly.
2. **Framework Efficiency:** Design the genetic algorithms framework to optimize computational efficiency. This should involve algorithmic improvements, effective use of *Kotlin*'s language features, and careful resource management.
3. **Comparative Study:** Conduct comparative studies between the *Kotlin*-based genetic algorithms framework and existing solutions, particularly in terms of syntactical complexity, and ease of use. This should provide a benchmark to evaluate the advantages and potential areas of improvement for our framework.
4. **Real-world Applications:** Apply the proposed *Kotlin*-based framework to address complex real-world problems. This will not only demonstrate the practical viability of the framework but also its potential to contribute to advancements in AI and EC.
5. **Future Enhancements and Expansion:** Anticipate and propose potential enhancements and extensions for the *Kotlin*-based genetic algorithms framework. Discuss its adaptability to incorporate future advancements in AI and EC, ensuring its sustainability and continued relevance.

These specific objectives aim to add more granularity to our research focus, ensuring a comprehensive and detailed exploration of the possibilities offered by *Kotlin* for the development of a novel genetic algorithms framework.

1.4 Methodology

Our research methodology employs a multi-faceted approach comprised of distinct phases. This approach maintains a balance between theoretical exploration and practical application, with a consistent emphasis on rigorous analysis and evaluation.

1. **Literature Review and Language Exploration:** We initiate our research with an exhaustive study and analysis of the current literature pertaining to genetic algorithm frameworks, thereby discerning their respective merits and shortcomings. This analysis will serve as a cornerstone for defining the quintessential attributes for a versatile, efficient, and user-friendly genetic algorithms framework. Concurrently, we will conduct an in-depth investigation of the language features offered by *Kotlin*, discerning their potential contributions to the design and development of the proposed framework.
2. **Framework Design and Development:** Informed by the comprehensive literature review and language exploration, we will transition to the design and implementation phase of the *Kotlin*-based genetic algorithms framework. The design strategy will involve the thoughtful employment of *Kotlin*'s unique features such as its expressive syntax, static typing, and ease of crafting DSLs, thereby ensuring that the resulting framework is efficient, robust, and user-friendly.
3. **Application to Real-world Challenges:** Subsequently, we will utilize the developed *Kotlin*-based framework to address complex real-world problems. This stage will function as a testing ground to demonstrate the practical applicability, potential contributions, and viability of the framework in fostering advancements in the realms of AI and evolutionary computation.
4. **Future Adaptability and Expansion:** In the final phase of our research, we will deliberate upon prospective enhancements and extensions for the *Kotlin*-based genetic algorithms framework. Moreover, this phase will entail an assessment of the framework's adaptability and scalability to accommodate future advancements in AI and evolutionary computation, thereby assuring its long-term sustainability and relevance.

Throughout the implementation of our research methodology, we will uphold an analytical and critical perspective to discern the significance of our findings and their implications. We acknowledge and uphold the ethical principles of transparency and honesty in reporting our results and documenting the progress of our research.

1.5 Thesis Structure

1. **Background (chapter 2 on page 5):** This chapter offers a necessary foundation for understanding the central concepts utilized in this thesis. It encompasses a theoretical analysis of the relevant algorithms, setting the stage for the discussions and investigations that follow.

2. **State of the Art (chapter 3 on page 31):** Here, we present a comprehensive review of the current advancements in the domain of evolutionary computation. We particularly emphasize and examine the prevailing frameworks in this field, facilitating a comparative context for our research.
3. **Framework Design and Implementation (chapter 4 on page 45):** This chapter elucidates the architecture of our framework, detailing the design considerations and the subsequent implementation.
4. **Function Optimization Case Study (chapter 5 on page 71):** In this section, we put our framework to the test by applying it to a real-world function optimization problem. We utilize 20 classical benchmark functions to assess the framework's performance, with the functions elaborately displayed in appendix B on page 101.
5. **Crash Reproduction Problem Case Study (chapter 6 on page 81):** In our third case study, we explore the application of the framework to a crash reproduction problem, showcasing its utility in a distinctly different domain.
6. **Conclusion and Future Work (chapter 7 on page 91):** In the concluding chapter, we summarize the breadth of work accomplished in this thesis and propose potential avenues for future investigations and enhancements based on our findings.

This structure facilitates a progressive narrative of our research, allowing for a coherent understanding of our objectives, methodology, and outcomes.

Chapter 2

Theoretical Framework

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

2.1 Evolutionary Algorithms

In the field of computational intelligence, ***evolutionary algorithms*** (EA) [27] 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 via emergent intelligence [10]. 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*.
- **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[3] 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*.

Remark. *"Parallel" in this context means that the algorithm is exploring multiple points in the search space in the same generation (iteration). It should not be confused with parallel computing, which is a technique used to speed up the execution of the algorithm by running it on multiple processors.*
Even though EAs are parallel in nature, they can be run on a single processor. Nevertheless, this parallel nature makes them a good candidate for parallel computing; in this thesis we will not explore the use of parallel computing to speed up the execution of certain stages of the algorithm.

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] Which comprises both the practice and the study of evolutionary algorithms. See definition A.6 on page 97.
[2] See definition A.11 on page 98
[3] 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.

2.2 Genetic Algorithms

Genetic Algorithms (GA)[4] [6, 7, 27, 31] are a type of EA where a ***population*** of ***individuals***[5] 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***[6], 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:

Listing 2.1– Genetic algorithm

```
var population = initialize population  // Creates a random population of individuals
population.forEach { it -> evaluate it }  // Calculates the fitness of each individual
while (termination condition is not met) {
↪  // Could be a pre-defined number of generations, a desired fitness level, etc.
    val survivors = select survivors from population
    ↪  // Selects individuals for the next generation
    val parents = select parents from population
    ↪  // Selects a subset of individuals as parents
    val offspring = alter parents
    ↪  // Applies genetic operators to parents to create new individuals
    offspring.forEach { it -> evaluate it }
    ↪  // Calculates the fitness of each new individual
    population = survivors + offspring // Creates the next generation
}
return fittest from population  // Returns the most fit individual
```

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.

2.2.1 Representation and Evaluation

A pivotal component of a Genetic Algorithm (GA) is the representation of individuals, which encodes potential solutions into a form manipulable by the GA. This representation delineates the algorithm's search space and is a prime determinant of its performance.

Definition 2.1 (Gene)**.** *Representation of a single component of a candidate solution to a given optimization problem. Formally, for a multi-dimensional function f, a gene is an element g in the domain of f.*

The predominant representation is a matrix of genes termed the ***genotype***[7], wherein each column is denoted as a ***chromosome***[8].

[4] Also known as Simple Genetic Algorithms (SGA) [27], or Traditional Genetic Algorithms (TGA) [31].

[5] ***Individual:*** A candidate solution to a given optimization problem. Formally, an individual is a pair $(\mathbf{G}, \mathbf{f})$, where $\mathbf{G}$ is the genotype and $\mathbf{f}$ is the fitness value of the individual.

[6] ***Genotype:*** Representation of the full genetic information of a candidate solution to a given optimization problem. Formally, a genotype is a matrix $\mathbf{G} = (\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_n)$, where $\mathbf{c}_i$ is a chromosome.

[7] See [77]

[8] ***Chromosome:*** Representation of a single column of genetic information of a candidate solution to a given optimization problem. Formally, a chromosome is a vector $\mathbf{c} = (g_1, g_2, \ldots, g_n)$, where g_i is a gene.

Definition 2.2 (Cardinality of the search space)**.** *The **cardinality of the search space** is the number of different individuals that can be represented by the encoding.*
Formally, given a vector of alphabets $(\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n)$*, and a representation* $\mathbf{G}$ *with* n *chromosomes of lengths* $(m_1, m_2, \ldots, m_n)$ *where each chromosome is encoded using the alphabet* $\mathcal{A}_i$*, the cardinality of the search space* S *is defined as:*

$$|S| = \prod_{i=1}^{n} |\mathcal{A}_i|^{m_i} \tag{2.1}$$

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 [6], 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, this is known as the ***One Max problem*** (OMP) [14][9] (refer to section 3.1 on page 31 for a more detailed description of the problem). In this case, we can use a single column matrix $\mathbf{G}$ 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,

$$|S_{\text{OMP}}| = \prod_{i=1}^{1} |\mathcal{A}|^n = 2^n$$

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 2.3 (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 –this may consist of maximizing or minimizing the objective.*
The fitness function is usually defined by the user of the algorithm, and it is problem dependent.

Definition 2.4 (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,[10] so the fitness function would be defined as follows:

$$\phi_{\mathbf{G}} = \sum_{i=1}^{n} g_i \tag{2.2}$$

Where g_i is the i-th gene of the genotype $\mathbf{G}$.

[9] Also ***Ones Counting*** problem [77], or ***Max Ones*** problem [60].
[10] A single objective, or unimodal, problem is a problem with a single objective function. In contrast, a multi-objective, or multimodal, problem is a problem with multiple objective functions.

Having laid the groundwork by illustrating the pivotal role of representation in GAs, we've delineated how the representation forms the backbone of the algorithm's search space and in turn determines its performance. Key terminologies, including the genotype, chromosomes, cardinality of the search space, and the crucial role of the fitness function, have been elucidated with references and examples. With this foundational knowledge, in the forthcoming section, we will delve deeper into the initialization phase of the algorithm, setting the stage for how the GA seeds its initial population and embarks on the quest for optimal solutions.

2.2.2 Initialization

In the initialization phase of a genetic algorithm, the foundational stage is set for the algorithm's evolutionary journey. Here, we define and setup the population of individuals to be employed in the search process. Whether informed by prior knowledge or randomly generated, each individual's evaluation is pivotal to steer the algorithm's quest for optimal solutions.

As we delve further, stages such as selection (section 2.2.3 on the next page), crossover (section 2.2.4.1 on page 10), and mutation (section 2.2.4.2 on page 11) build upon this foundational phase.

A GA operates on a group of individuals termed a ***population***. Determining the population's size and its initialization is crucial. Typically, this initialization process leans on randomness, but it can also be informed by insights about the problem at hand [21].

Upon initializing the population, each individual undergoes evaluation to receive a ***fitness value***. This step is imperative to glean insights about the problem, thereby directing the search towards enhanced solutions.

Using the ***One Max*** problem as a backdrop, where there's an absence of prior problem knowledge, the initialization encompasses a blind search of the search space, rendering it random. For each population member, a random binary string of length n is generated.

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:[11]

Generation 0

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

Table 2.1: Population of individuals in generation 0

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

Table 2.2: Fitness of the individuals in generation 0

Emphasis should be placed on the aggregation functions outlined in table 2.2. The ***average fitness*** is straightforward, offering an overall view of the population's performance; a higher[12] average suggests better overall performance. On the other hand, the ***standard deviation*** gauges the dispersion of fitness values around this average. A low standard deviation indicates a homogeneous population, which might hint at **premature convergence** due to limited search space exploration. Conversely, a high

[11] 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.

[12] Remember, a "higher fitness" refers to proximity to the optimal solution, not necessarily the greatest numerical value.

standard deviation signifies a diverse population, potentially suggesting **excessive exploration** without honing in on promising areas.

The initialization phase of a genetic algorithm sets the foundation by establishing the population of individuals for the search. This population might be randomized or informed by prior domain knowledge. Each member of this population undergoes a fitness evaluation, directing the search towards optimal outcomes. Using our OMP example, we commenced with a four-individual population with binary strings of length $n = 4$, assessing their fitness. This foundational step paves the way for the ensuing phases of selection (section 2.2.3), crossover (section 2.2.4.1 on the following page), and mutation (section 2.2.4.2 on page 11).

2.2.3 Selection

Following the initialization phase, a genetic algorithm (GA) progresses through its main evolutionary cycle. Key to this cycle is the process of selection, which mirrors natural selection by favoring individuals with higher fitness for reproduction.

Given a population P comprising N individuals, each identified by their fitness value ϕ_i (where $i \in \{1, \ldots, N\}$), the survival rate σ helps control elitism.[13] This rate determines the portion of the population that proceeds to the next generation without change. Specifically, $\lfloor \sigma N \rfloor$ individuals continue, while the remaining $\lceil (1 - \sigma)N \rceil$ are replaced by their descendants.[14]

For the remaining of this document, we will use the following notation to represent functions with named parameters:

$$f(x_1 : \tau_1, \ldots, x_n : \tau_n) \to \tau_r$$

where:

- f: Function name.
- x_i: Parameter name.
- τ_i: Parameter type.
- τ_r: Return type.

Definition 2.5 (Selection Operator)**.** *A tool for choosing specific members from a population, formally expressed as:*

$$\Sigma(P : \mathbb{P}, n : \mathbb{N}, \ldots) \to \mathbb{P}$$

Parameters include:

- $\mathbb{P}$*: Set of possible populations.*
- $\mathbb{N}$*: Set of natural numbers.*
- P*: A given population.*
- n*: Number of selections from* P*.*

This selection operator usually integrates randomness, bringing some unpredictability to the process. As an exemplification, let's consider the *roulette wheel* selection operator.[15] Here, individuals are assigned selection probabilities according to their fitness, as expressed by:

$$\rho_\Sigma(i) = \frac{\phi_i}{\sum_{j=1}^{N} \phi_j} \tag{2.3}$$

Following this, individuals are probabilistically selected for survival. For instance, if I_2 persists, the remaining I_1, I_3, and I_4 yield their places to the offspring.

Concluding this introduction to selection in GAs, the subsequent sections will investigate this process further and introduce the variation operators crucial for offspring generation.

[13] For more on elitism, refer to definition A.4 on page 97.
[14] Using both the *floor* and *ceiling* functions ensures a consistent total population of N.
[15] Refer to **??** on page ?? for details.

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

Table 2.3: Probabilities of selection for our sample population.

2.2.4 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.

Definition 2.6 (Variation operator)**.** *A variation operator is used to create new individuals from existing ones. Formally, it is a variadic function represented as*

$$\varphi(P : \mathbb{P}, \rho_\varphi : \mathbb{R}, \dots) \to \mathbb{P}$$

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_φ *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 section 4.4 on page 51.

2.2.4.1 Crossover

In genetic algorithms, a prominent variation operator is the ***crossover***. This operator mirrors the genetic recombination seen in nature.[16] It facilitates the exchange of genetic material between two individuals, spawning a new generation.

Definition 2.7 (Crossover operator)**.** *The crossover operator recombines genetic material from existing individuals to create new ones. Formally, it is represented as:*

$$X(P : \mathbb{P}, \rho_X : \mathbb{R}, \dots) \to \mathbb{P}$$

with the following parameters:

- $\mathbb{P}$ *– the set of all possible populations,*
- $\mathbb{R}$ *– the set of real numbers,*
- P *– the population under variation,*
- ρ_X *– probability of applying the operator to an individual.*

In our analysis, we employ a condensed version of the ***single-point crossover*** operator.[17] This operator picks the first half of the genes from two parental figures and births two new offspring by swapping these chosen genes.

Take, for instance, two parent individuals, $I_1 = 1100$ and $I_2 = 0001$. Utilizing the *single-point crossover* operator, we select the first half of the genes: 11 from I_1 and 00 from I_2. This exchange may produce the offspring $O_1 = 1101$ and $O_2 = 0000$, demonstrated in fig. 2.1 on the facing page.

Following another iteration of the single-point crossover operator, we can generate a result as shown in table 2.6 on page 12, leading to a new population $\mathbf{O} = \{(0000, 0), (1101, 3), (0101, 2)\}$.

[16] Also known as ***crossing-over*** in [6].
[17] Refer to **??** on page ??.

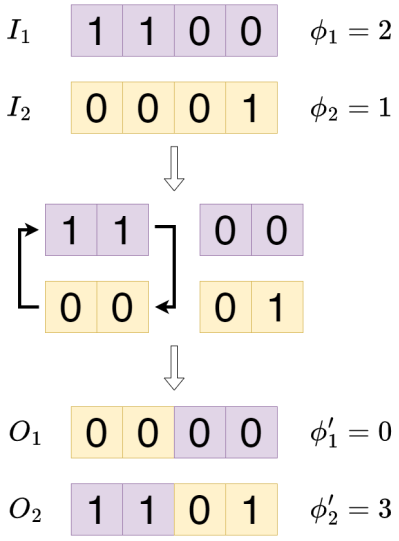


Figure 2.1: Single-point crossover

If we now use these offspring as-is to create the next generation, we would obtain the population shown in table 2.4:

Generation 1

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

Table 2.4: Population after applying the single-point crossover operator. Note that I_2 is the survivor of the previous generation picked in section 2.2.3 on page 9.

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

Table 2.5: 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

By the metrics in table 2.5, the population's average fitness rises from 1 to 1.25, with the fittest individual's score jumping from 2 to 3. This uplift highlights the crossover operator's role in directing the search towards enhanced solutions.

While the crossover operation elevates the average population fitness, incorporating a ***mutation*** operator can further boost genetic diversity[18] and sidestep early convergence to inferior solutions. The subsequent section delves into this operation.

2.2.4.2 Mutation

While the crossover operator effectively recombines existing genetic material, it is limited by the genetic diversity already present in the population. This can sometimes lead to premature convergence, especially for complex problems characterized by nu-

[18] See definition A.8 on page 98.

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 2.6: 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, $\cdot$ represents a "discarded" value (since according to the survival rate, only three offspring need to be produced). The discarded elements are usually selected using a selection operator alike to the one described in section 2.2.3 on page 9, but in this case, we selected them by hand since this is meant to be an illustrative example.

merous local optima, like the ***Rastrigin function***.[19]

To combat this and infuse fresh *diversity*, the *mutation* operator is employed. It introduces small, probabilistic changes to the genetic makeup of individuals.

Definition 2.8 (Mutation operator)**.** *The mutation operator introduces variations in an individual's genetic material based on a predefined probability, resulting in a new population. Formally, a mutation operator can be represented as:*

$$M(P : \mathbb{P}, \mu : \mathbb{R}, \dots) \to \mathbb{P}$$

Where:

- $\mathbb{P}$ *denotes the set of all possible populations.*
- $\mathbb{R}$ *signifies the set of real numbers.*
- P *stands for the current population.*
- μ *indicates the mutation rate – the chance of an individual undergoing mutation.*

Additional parameters vary depending on the specific mutation operator in play.

For example, the "One Max" problem might employ a ***bit-flip*** mutation.[20] This mutation scans every gene in an individual, flipping it based on a predetermined probability.

Let's say we apply this mutation with a rate of $\mu = 1$ to the population post-crossover (as discussed in section 2.2.4.1 on page 10). The resulting mutated population, $\mathbf{O}'$, is illustrated in table 2.7.

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 2.7: Illustration of the ***bit-flip*** mutation operator applied to the population resulting from the crossover operation in section 2.2.4.1 on page 10.

By analyzing the resulting offspring, as highlighted in table 2.8 on the next page, we can observe the mutation's role in enhancing diversity. For instance, no member of the original population had a 1 in the third position, rendering crossover incapable of producing such a gene configuration. However, post-mutation, three individuals exhibit this trait.

[19] Refer to appendix B.14 on page 109.
[20] For more details, refer to **??** on page ??.

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

Table 2.8: Population after applying the ***bit-flip*** mutation operator to the population resulting from the crossover operation in section 2.2.4.1 on page 10.

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

Table 2.9: Fitness of the population after applying the ***bit-flip*** mutation operator to the population resulting from the crossover operation in section 2.2.4.1 on page 10.

In summary, the mutation operator serves a pivotal role in genetic algorithms. It rejuvenates population diversity, mitigating early convergence to suboptimal solutions. By fostering exploration of the search landscape, it allows potentially superior traits to surface. Nonetheless, the mutation rate's calibration is a balancing act: excessively high rates might destabilize favorable traits, while overly conservative rates might inadequately deter premature convergence. The specific mutation operator and its associated rate significantly influence the genetic algorithm's exploratory efficiency.

Having concluded the variation phase, we can transition to the subsequent stage of the genetic algorithm.

2.2.5 Termination

The genetic algorithm assesses the termination criteria after generating each new population. If met, the algorithm ends and outputs the best-found individual. Otherwise, the generational cycle repeats.

Suppose our termination criterion is the identification of an individual with all ones, represented as 1111. This would mean the fitness function, $\phi_{\mathbf{G}}$, attains the value 4.

During our exploration, after applying the variation operators, we indeed discovered the individual 1111. This satisfies our termination criterion, prompting the algorithm to conclude its operation.

Yet, as illustrated in table 2.10, the algorithm has not scoured the entire search space. Instead, the genetic algorithm prioritizes a fitness-guided exploration over a complete one. Still, witnessing the increasing fitness of individuals over generations signals a trend towards an optimal or near-optimal solution.

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

Table 2.10: A map of the search space explored by the genetic algorithm. Dark gray cells denote the candidates the algorithm reviewed. The position of each individual corresponds to its binary representation, using the row for the first two bits and the column for the last two.

In small search arenas, the distinction between a genetic algorithm and random searches might appear negligible. However, as the search space expands, which we delve into later in this thesis, the difference becomes substantial.

Despite their stochastic nature, genetic algorithms do not always promise optimal outcomes. Their efficacy hinges on various components, including the fitness function, representation, variation operators, and selection techniques. This thesis delves deeper into these elements and analyzes their performance across diverse problems.

In essence, the termination phase is pivotal in dictating the final outcome of the genetic algorithm. By setting a clear termination condition, like achieving the highest fitness score, the algorithm efficiently navigates the search terrain. Although it might not exhaustively search, it employs a fitness-centric strategy to steer closer to optimal solutions. It's vital to acknowledge the intrinsic limitations of genetic algorithms. Yet, when configured right, their capability to overshadow random searches, especially in vast spaces, is undeniable. We'll further explore this subject in the subsequent segments of this thesis.

2.3 Genetic Programming

Genetic Programming (GP) [7, 9, 22, 27] has been carved out as a specialized subfield of EAs which emphasizes evolving a collection of computer programs tailored to address specific problems. While GP can be viewed as a natural evolution of GAs, it distinguishes itself by its primary goal: GAs fine-tunes parameters to optimize a given function, whereas GP is geared towards program induction.[21]

At their core, both GP and GA operate on similar foundations. They use a population-based strategy, employ a fitness function to assess individual performance, and apply genetic operators to produce new individuals.

Remark. *GP can be seen as a specific type of GA, where individuals are not just a matrix of genes but are represented as computer programs. These programs are usually structured as abstract syntax trees (ASTs).*

Within the population of GP, every individual represents a computer program, built from a library of primitives, distinguished as ***functions*** and ***terminals***. One way to represent this is as a composite pattern where functions are internal nodes, and terminals are the leaves. This structure is commonly referred to as an abstract syntax tree (AST). For a visual representation, refer to fig. 2.2.

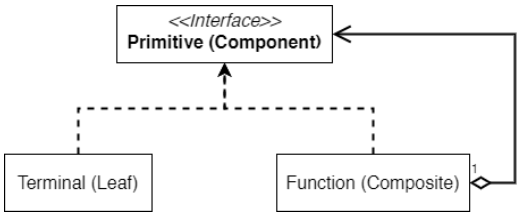


Figure 2.2: Representation of GP individuals as a composite structure.

The GP algorithm, delineated in listing 2.2, bears structural resemblance to the GA algorithm. The key differentiator rests in the specific genetic operations tailored for GP.

Listing 2.2– Structural blueprint of the Genetic Programming algorithm, underscoring its architectural alignment with the Genetic Algorithm.

```
var population = recursively construct random programs
population.forEach { it -> execute it then evaluate it }
while (termination criteria is not met) {
  val survivors = select survivors from population
  val parents = select parents from population
  val offspring = alter parents
  offspring.forEach { it -> execute it then evaluate it }
  population = survivors + offspring
}
```

[21] A more detailed definition of program induction can be found in definition A.17 on page 99.

```
return fittest from population
```

In the following sections, we will delve into the key components of the GP algorithm, clarifying these concepts with practical examples.

2.3.1 Representation and Evaluation

2.3.1.1 Representation

As with GAs, the representation of the individuals is one of the most important aspects of 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 classical 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 [7]. In this example, our goal is to use symbolic regression to approximate the function

$$f(x) = 5x^3 - 2x^2 + \sin(x) - 7;\ x \in [-1, 1] \tag{2.4}$$

using this function, we can generate a set of points that lies on the curve as shown in fig. 2.3 and table 2.11 on the next page.

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. $\{c \mid c \in [1,\ 7] \wedge c \in \mathbb{Z}\}$ (An ephemeral constant)[23]

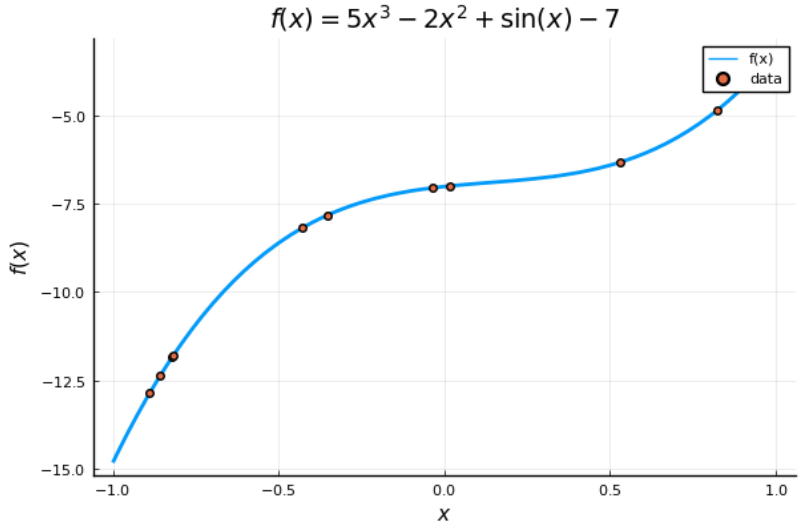


Figure 2.3: A set of points generated from the function $5x^3 - 2x^2 + \sin(x) - 7$

Using this set of functions and terminals, we can represent the program as a tree, as shown in fig. 2.4 on the next page.

[22] For example, by applying genetic operators.
[23] See definition A.5 on page 97.

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

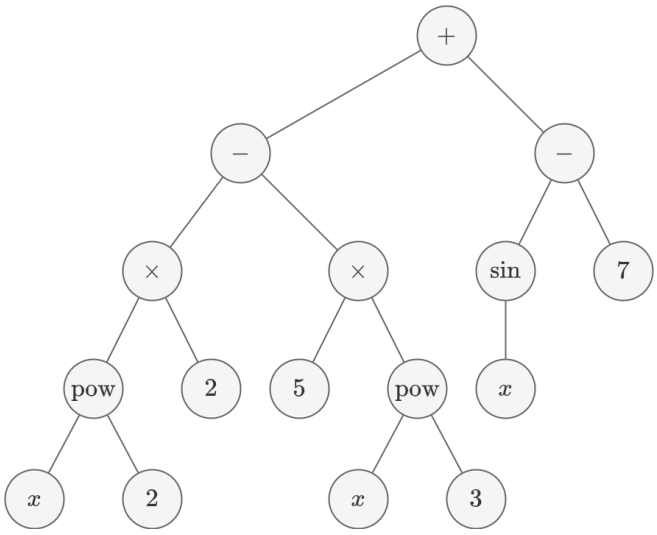


Figure 2.4: A possible tree representation of the program $5x^3 - 2x^2 + \sin(x) - 7$

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.

2.3.1.2 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 definition 2.2 on page 7, 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.

[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* [9], Angeline and Pollack's *Genetic Library Builder* [8, 10], or Rosca and Ballard's *Adaptive Representation* [11].

Lemma 2.1. *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:*

$$|\mathbb{T}_H(\mathcal{T},\, \mathcal{F})| = \begin{cases} |\mathcal{T}| & \text{if } H = 0 \\ \sum_{f \in \mathcal{F}} |\mathbb{T}_{H-1}(\mathcal{T},\, \mathcal{F})|^{A(f)} & \text{if } H > 0 \end{cases} \tag{2.5}$$

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:

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

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,[a] 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 $h + 1$:

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

□

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

Lemma 2.2. *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:*

$$|\mathbb{T}_{\leq H}(\mathcal{T}, \mathcal{F})| = \begin{cases} |\mathcal{T}| & \text{if } H = 0 \\ |\mathbb{T}_H(\mathcal{T}, \mathcal{F})| + |\mathbb{T}_{\leq H-1}(\mathcal{T}, \mathcal{F})| & \text{if } H > 0 \end{cases} \tag{2.6}$$

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 2.1. *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:*

$$|\mathbb{T}_{\leq H}(\mathcal{T}, \mathcal{F})| = \begin{cases} |\mathcal{T}| & \text{if } H = 0 \\ \left(\sum_{h=0}^{H-1} \sum_{f \in \mathcal{F}} |\mathbb{T}_h(\mathcal{T}, \mathcal{F})|^{A(f)} \right) + |\mathcal{T}| & \text{if } H > 0 \end{cases} \tag{2.7}$$

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 lemma 2.2 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 lemma 2.1 on the preceding page, we get:

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

By unrolling the recurrence relation, we get:

$$\begin{aligned} |\mathbb{T}_{\leq H}| &= \left(\sum_{f \in \mathcal{F}} |\mathbb{T}_{H-1}|^{A(f)} \right) + |\mathbb{T}_{\leq H-1}| \\ &= \left(\sum_{f \in \mathcal{F}} |\mathbb{T}_{H-1}|^{A(f)} \right) + \left(\sum_{f \in \mathcal{F}} |\mathbb{T}_{H-2}|^{A(f)} \right) + \cdots + \left(\sum_{f \in \mathcal{F}} |\mathbb{T}_1|^{A(f)} \right) + \left(\sum_{f \in \mathcal{F}} |\mathbb{T}_0|^{A(f)} \right) + |\mathbb{T}_0| \\ &= \left(\sum_{h=0}^{H-1} \sum_{f \in \mathcal{F}} \mathbb{T}_h(\mathcal{T}, \mathcal{F})|^{A(f)} \right) + |\mathcal{T}| \end{aligned}$$

□

Given a program viewed as a ***labeled tree***, we can determine the cardinality of the genetic programming algorithm's search space for a specific maximum height H. Considering the sets $\mathcal{T} = \{x, c\}$ and $\mathcal{F} = \{+, -, \times, /, \sin, \cos, \exp, \log\}$:

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

For an AST with a target program height of 4, we can set 5 as the maximum height for the programs in our search space, offering some flexibility to the generated programs. Given that c has 7 potential values, $|\mathcal{T}| = 8$, the search space's cardinality can be calculated accordingly.

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

Thus, the search space of the genetic programming algorithm is of the order of 10^{38} programs.[25] It should be easy to see that the size of the search space make it unfeasible to perform an exhaustive search.[26]

This is why we need to use a heuristic search algorithm, such as genetic programming. In fig. 2.5 we can see how the number of trees of height less or equal to h rapidly increases as h increases.

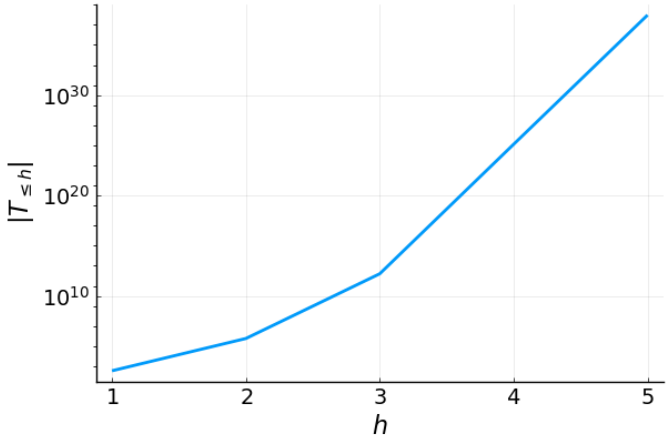


Figure 2.5: Total number of trees of height less or equal to h for $h \in \{0, \ldots, 5\}$ and $\mathcal{T} = \{x, c\}$ and $\mathcal{F} = \{+,\ -,\ \times,\ /,\ \sin,\ \cos,\ \text{pow}\}$. Note that the Y axis is in logarithmic scale.

2.3.1.3 Evaluation

To determine a program's fitness, various methods exist, but a prevalent choice is the ***mean squared error*** (MSE) between the points and the program.

Definition 2.9 (Mean Squared Error). *Given a vector of n predictions derived from n data points across all variables, where $\mathbf{y}_i$ represents the i-th observed value and $\hat{\mathbf{y}}_i$ is the i-th prediction, the MSE of the predictor is a function* $\text{MSE} : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ *defined by:*

$$\text{MSE}(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{n} \sum_{i=1}^{n} (\mathbf{y}_i - \hat{\mathbf{y}}_i)^2 \tag{2.8}$$

[25] This value was computed using the script shown in listing C.6 on page 123.

[26] If we approximate the time to evaluate a program to be 1 nanosecond, it would take approximately 3 sextillion years to evaluate all the programs in the search space.

The MSE is a benchmark for assessing an estimator's quality, especially in *machine learning* contexts.[27]

For our purposes, we employ the MSE to assess a program's fitness. Given a program P and two point sets, $\mathbf{x}$ and $\mathbf{y}$, as detailed in table 2.11 on page 16, and presuming $\mathsf{P}[\mathbf{x}]$ as the points produced by evaluating P on $\mathbf{x}$, with $\mathsf{P}(x)$ being the evaluation result for point x, the fitness of P can be described as:

$$\phi_{\mathsf{P}} = \text{MSE}(\mathbf{y}, \mathsf{P}[\mathbf{x}]) = \frac{1}{n}\sum_{i=1}^{n}(\mathbf{y}_i - \mathsf{P}(\mathbf{x}_i))^2 \tag{2.9}$$

In summation, when gauging a program's fitness in GP, error measures play a pivotal role, with MSE emerging as a primary standard. By contrasting a program's predictions with actual data, the MSE offers a valuable assessment of prediction accuracy. Its inherent simplicity effectively establishes a quality standard for future enhancements. Such an organized methodology guarantees that GP systematically traverses the solution domain, persistently optimizing candidate solutions.

2.3.2 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*** [7]. 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 listing 2.3.

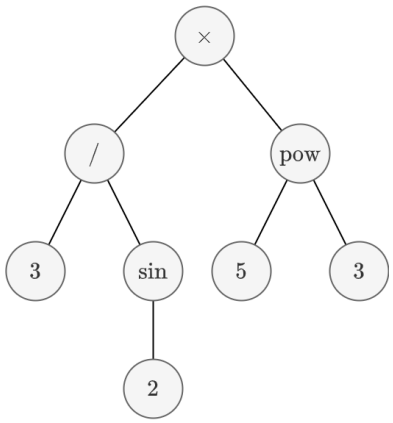
Listing 2.3– The grow method for generating random trees.

```
// Given:
val lo: ℕ // Minimum height
val hi: ℕ // Maximum height
require(lo ≤ hi)
val n = random.int(lo..hi) // Random height
fun grow(ts: Set<Terminal>, fs: Set<Function>, depth: ℕ) {
  require(ts != ∅ && fs != ∅)
  val c = ∅
  if (
    depth == n || (depth ≥ lo && random.double(0..1) ≤ ts.size / (ts.size + fs.size))
  ) {
  // d = n ∨ (d ≥ l ∧ random() < |𝒯| / (|𝒯| + |ℱ|))
    return random node from ts
  } else {
    val f = random node from fs
    for (i in 1..arity(f)) {
      c += grow(ts, fs, d + 1)
    }
    return a tree with f as root and c as children
  }
}
```

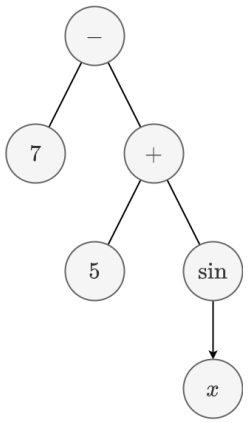
[27] While we favor the mean squared error for its straightforward nature and prominence in literature, other error measures like the *mean absolute error* (MAE) or the *cross-entropy* (CE) loss functions can also be considered.

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

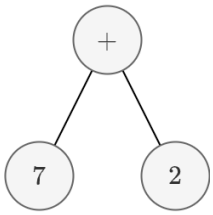
Now that the algorithm can generate random trees, it can generate a random population of trees by generating a random tree for each individual in the population. Assuming a population size of $p = 4$, a maximum height of $h = 3$, a minimum height of $l = 1$, and the primitives set defined in the previous 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 fig. 2.6.



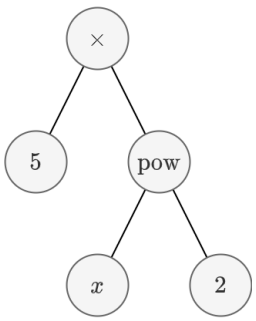
(a) $\mathbf{I}_1 = \dfrac{3}{\sin(2)} \times 5^3$



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



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



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

Figure 2.6: A population of random trees.

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.

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

With this, we can assign a fitness to each individual in the population as shown in table 2.12. A summary of the population's fitness is shown in table 2.13.

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 2.12: 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 2.13: Fitness of the individuals in generation 0

We can observe that the worst individual has an error significantly larger than the best individual. This is to be 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 fig. 2.7 on the facing page. It is clear from the figure that the worst individual is $\mathbf{I}_1$.

In summary, the initialization phase plays a pivotal role in setting the foundational landscape for the Genetic Programming process. Through randomized generation methodologies, like the grow method, the algorithm nurtures a diverse pool of candidate solutions. This diverse initial population enhances the algorithm's probability of exploring different regions of the solution space, thereby increasing the odds of locating an optimal solution. Yet, these generated programs are merely starting points. Their evaluation, determined through metrics such as the MSE, provides an empirical compass to guide the evolution process. Once the fitness landscape has been charted out, the algorithm is primed for the next pivotal phase: selection. In the upcoming section, we delve into the mechanics of selection, a process crucial for ensuring the survival and propagation of the fittest candidates.

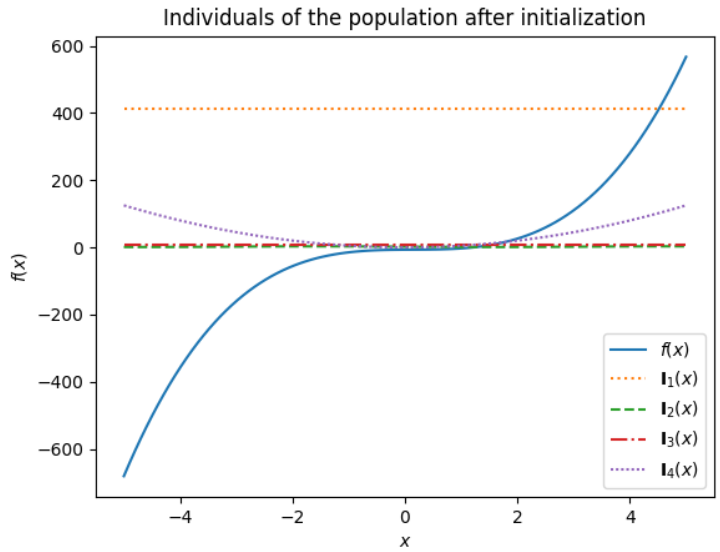


Figure 2.7: Graphical representation of the population in generation 0 compared to the expected output.

2.3.3 Selection

The selection process in GP is similar to that of GAs. 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 [33]. 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 eq. (2.3) on page 9, 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:

$$\phi'(I) = \left(\sum \Phi_{\mathbf{P}}\right) - \phi_I \tag{2.10}$$

Here, ϕ_I signifies the fitness of individual I, and $\Phi_{\mathbf{P}}$ represents the *batch fitness function* defined in definition 2.4 on page 7 applied to the population $\mathbf{P}$. We then define the selection probability for an individual $\mathbf{P}_i$ as:

$$p_i = \frac{\phi'(\mathbf{P}_i)}{\sum_{j=1}^{N} \phi'(\mathbf{P}_j)} \tag{2.11}$$

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 table 2.14 on the next page. 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.

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 fig. 2.8 on the following page.

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 2.14: Selection probabilities for the symbolic regression problem.

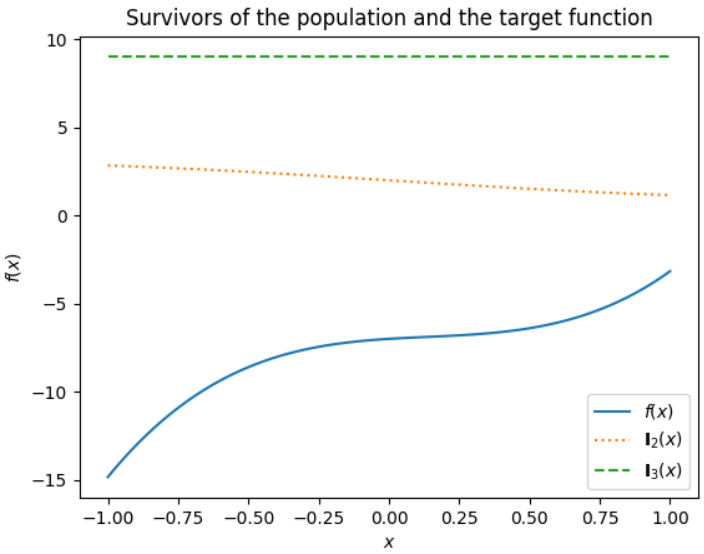


Figure 2.8: Comparison between the survivors, $\mathbf{I}_2 = 7 - (5 + \sin(x))$ and $\mathbf{I}_3 = 7 + 2$, and the target function, f.

Selection remains a cornerstone of evolutionary algorithms, ensuring that individuals possessing beneficial traits have a greater likelihood of transferring those traits to subsequent generations. While nuances distinguish the selection processes of Genetic Programming (GP) and Genetic Algorithms (GAs), the underlying principle remains: to direct the search towards more promising regions of the solution space. As we progress further, it's essential to recognize the interplay between selection and other operators, understanding that their combined effects mold the trajectory of the evolutionary process. The insights gleaned from this understanding will aid in fine-tuning algorithms and achieving better results in future endeavors.

2.3.4 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***.

2.3.4.1 Crossover

Variation operators in 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 GAs, described in section 2.2.4.1 on page 10, 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 chapter 6 on page 81, 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 **??** on page ??.

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 fig. 2.9.

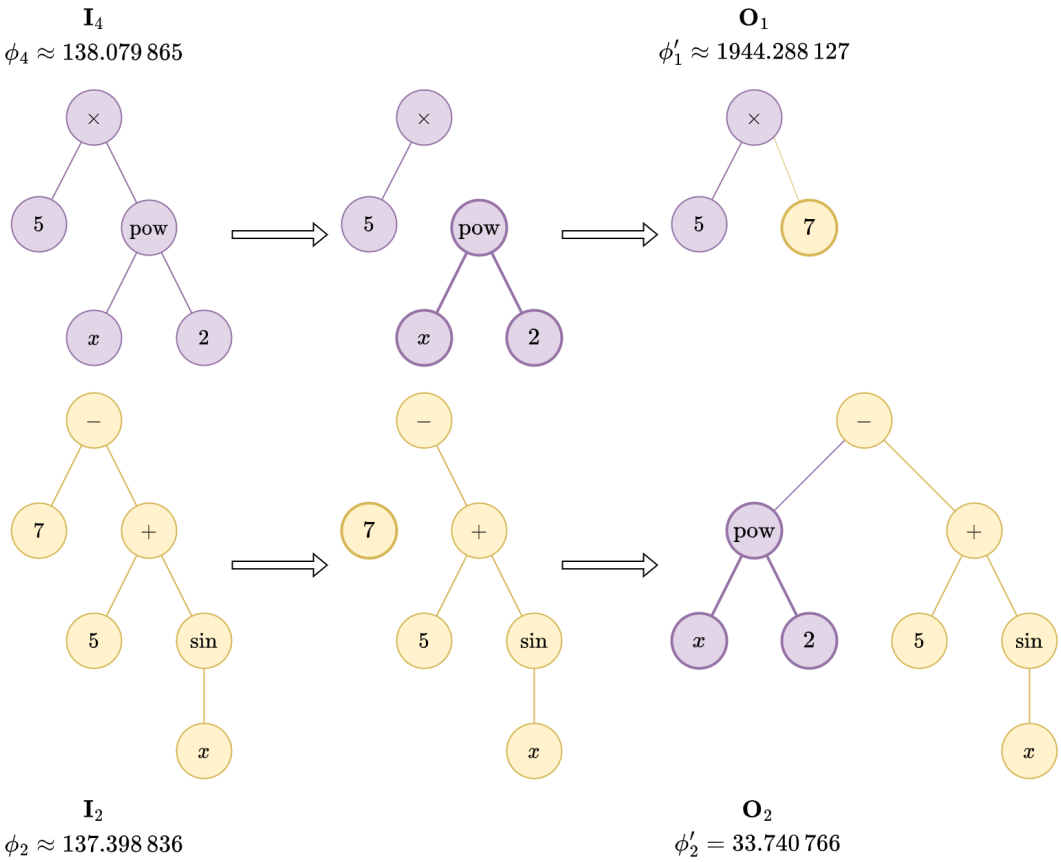


Figure 2.9: Crossing over of $\mathbf{I}_2 = 7 - (5 + \sin(x))$ and $\mathbf{I}_4 = 5x^2$, producing $\mathbf{O}_1 = x^2 - (5 + \sin(x))$ and $\mathbf{O}_2 = 5 \cdot 7$.

Following the application of the subtree crossover operator, the fitness of the individuals in the population is evaluated as shown in table 2.15 on the next page. A summary of the population's fitness is given in table 2.16.

	Fitness	**Individual**
Best	33.740 766	$\mathbf{O}_2(x)$
Worst	1 944.288 127	$\mathbf{O}_1(x)$
Average	611.837 999	
Standard deviation	776.701 997	

Table 2.16: Fitness summary of the population after applying the subtree crossover operator.

Generation 1

Individual	Program	Fitness
$\mathbf{I}_2(x)$	$7-(5+\sin(x))$	137.398 835
$\mathbf{I}_3(x)$	$7+2$	331.924 267
$\mathbf{O}_1(x)$	$5\cdot 7$	1 944.288 127
$\mathbf{O}_2(x)$	$x^2-(5+\sin(x))$	33.740 766

Table 2.15: 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 table 2.13 on page 22, 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%.

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

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 $76\,814.062\,197$ to $776.701\,997$, is also seen. This decrease of around 98.989% indicates that the population's diversity has reduced.

$$\frac{\sigma_i-\sigma_X}{\sigma_i}=\frac{76\,814.062\,197-776.701\,997}{76\,814.062\,197}\approx 98.989\%$$

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.

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

2.3.4.2 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*** [22, 77], 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 section 2.2.4.2 on page 11. 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 section 2.3.4.1 on page 24, 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 **??** on page ??. A summary of the fitness of the population is presented in **??** on page ??.

Generation 1

Individual	Program	Fitness
$M(\mathbf{I}_2(x)$	$7-(5+\cos(x))$	112.297 411
$M(\mathbf{I}_3(x)$	$6+2$	296.910 273
$M(\mathbf{O}_1(x)$	$6\cdot 7$	2 609.386 088
$M(\mathbf{O}_2(x)$	$x^3-(5+\sin(x))$	15.295 863

Table 2.17: Population after applying the node mutation operator.

	Fitness	Individual
Best	15.295 863	$M(\mathbf{O}_2(x)$
Worst	2 609.386 088	$M(\mathbf{O}_1(x)$
Average	758.472 409	
Standard deviation	1 073.402 832	

Table 2.18: Fitness summary of the population after applying the node mutation 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.

With this, we can conclude that after at the end of the generation, the population shows an improvement of 98.298% in fitness, and 98.603% in standard deviation.

$$\frac{\bar{\Phi}_i-\bar{\Phi}_M}{\bar{\Phi}_i}=\frac{(44\,551.063\,525-758.472\,409)}{44\,551.063\,525}\approx 98.298\%$$

$$\frac{\sigma_i-\sigma_M}{\sigma_i}=\frac{(76\,814.062\,197-1\,073.402\,832)}{76\,814.062\,197}\approx 98.603\%$$

fig. 2.10 on the following page shows the population after applying the node mutation operator. Two points should be clear from this figure. First, all individuals of the population are closer to the target function than the individuals in the initial population. Second, the new fittest individual is $\mathbf{O}_2$, this individual has a shape similar to the target function.

Since GP is still in a prototypal stage in Keen, it cannot produce reliable results on the symbolic regression problem, but the potential of this approach can be seen in DEAP's[28] documentation,[29] where an error of the order of 10^{-33} is achieved on 40 generations.

In summary, mutation plays a pivotal role in the genetic programming process. It injects fresh genetic material into the population, thereby promoting diversity and ensuring the continual exploration of the search space. As demonstrated, mutation not only alters the structure of individuals but can also lead to significant enhancements in fitness. By tactfully combining crossover and mutation operations, genetic programming can efficiently traverse the vast landscape of possible solutions. This underscores the importance of maintaining a delicate balance between exploration (introducing new genetic structures) and exploitation (refining existing successful solutions).

2.3.5 The Generalization Problem in GP

2.3.5.1 Understanding Generalization

Generalization in Genetic Programming (GP) relates to a program's proficiency in handling unseen data, not solely the data it evolved with. A truly generalized program goes beyond rote memorization of training data. It discerns underlying patterns, equipping it to make precise predictions or decisions on new, unencountered instances.

[28] DEAP will be formally introduced on chapter 3 on page 31

[29] https://github.com/DEAP/deap/blob/60913c5543abf8318ddce0492e8ffcdf37974d86/examples/gp/symbreg.py

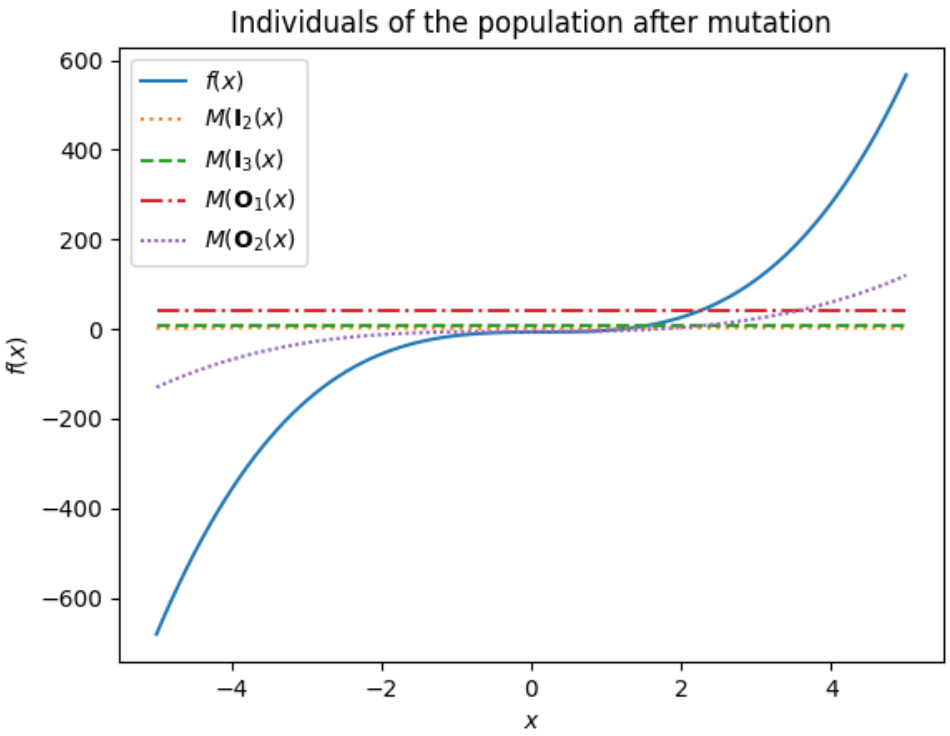


Figure 2.10: Population after applying the node mutation operator on the individuals obtained from the crossover operation (refer to table 2.15 on page 26).

2.3.5.2 Significance of Generalization in GP

Generalization holds a paramount position in GP. The overarching objective of evolving programs is to address real-world challenges. A program's utility diminishes if it can't generalize, limiting its effectiveness to familiar data. Particularly in predictive modeling, the emphasis isn't on a model's ability to recall past data, but on its precision with upcoming, unpredictable data. Therefore, the act of generalizing surpasses basic accuracy measures on training datasets, signaling a model's resilience and practical relevance.

2.3.5.3 The Overfitting Dilemma

Overfitting stands as an adversary to generalization. A program succumbs to overfitting when it molds excessively to training data, picking up on its noise and irregularities rather than its broad trends. Such overfitted programs may boast of stellar accuracy on training datasets but falter with unseen data. Within the GP context, visualizing overfitting can be akin to observing a program that's evolved into a convoluted form, brimming with redundant branches or logic catering to the peculiarities of training data.

2.3.5.4 Addressing the Generalization Problem

The academic landscape has brought forth numerous techniques to counter the generalization conundrum in GP, some such works include:

- Chen et al.'s approach hinged on feature selection tailored for high-dimensional symbolic regression [35].
- Kushchu advocated the adoption of multiple fitness functions, a measure aimed at bolstering generalization [15].
- Enhancing generalization through the Rademacher distribution was a strategy put forth by Chen et al. [41].

While this thesis will sidestep an in-depth exploration of the generalization issue in GP, it's pivotal to acknowledge its profound implications in GP and its standing as a fervently pursued research frontier.

Chapter 3

State of the Art: Relevant Work

This chapter delves into the current state of the art in evolutionary computation frameworks. While evolutionary algorithms themselves aren't new, their applications keep evolving[1] with technological advancements in programming languages and tools.

We'll spotlight five major frameworks: ***Agile Artificial Intelligence in Pharo***, ***DEAP***, ***Jenetics***, ***ECJ***, and ***GeneticSharp***. Each section provides an overview of the respective framework, complete with code samples, showcasing their syntax and distinct features. This exploration aims to give readers a broad understanding of the array of tools in this domain.

Moreover, this review sets the context for our forthcoming contribution: an evolutionary computation framework crafted in Kotlin. To relate, listing C.1 on page 117 features the ***Keen*** framework's approach to the ***OneMax*** problem, akin to the examples herein.

While we strive for objectivity, we recognize that personal biases may color our perceptions. We don't position ourselves as experts in the covered frameworks or languages, and there's a possibility that the provided implementations could be optimized further. We invite readers to delve into these frameworks firsthand and shape their own conclusions.

For transparency, it's worth noting that this review is impartial. We haven't received any compensation from, nor are we affiliated with, any of the framework developers aside from Bergel's ***Agile Artificial Intelligence in Pharo*** book. This chapter reflects our individual insights and experiences.

3.1 The One Max Problem

For the purpose of illustrating the use of the different frameworks, we will use the *One Max problem* introduced on section 2.2.1 on page 6.

The ***One Max 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:

$$\phi(x) = \sum_{i=1}^{n} x_i \tag{3.1}$$

The function $\phi(x)$ counts the number of ones in the string x. The maximum possible value of $\phi(x)$ 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.

[1] No pun intended.

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.

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.

3.2 Agile Artificial Intelligence in Pharo

Alexandre Bergel's ***"Agile Artificial Intelligence in Pharo"*** [39] delivers a comprehensive exploration of genetic algorithms, encapsulating their theory and application within the *Pharo* programming environment. This language, dynamic and reflective, finds its roots in *Smalltalk*. The source code of the framework is licensed under a *freeware license* and is available on *GitHub*.[2]

Bergel's tome not only introduces readers to genetic algorithms but also to neural networks and the concept of neuroevolution. The framework's design and the reasoning behind it are meticulously detailed, offering readers insights into its three main components: ***genetic operators***, ***selection operators***, and the ***evolution engine***. Accompanying this detailed framework are test cases. These serve dual roles: exemplifying the framework's application and guiding enthusiasts in developing their own robust and extensible frameworks.

The book also provides utilities for the visualization of genetic algorithms and neural networks. These are implemented using *Roassal*, a visualization engine for *Pharo*. The engine is capable of rendering a variety of graphs, including directed and undirected graphs, trees, and more relevant to this document, a summary of the evolution fitness over time.

To showcase the framework's utility, the OMP problem resolution via a genetic algorithm is provided:

Listing 3.1– A simple genetic algorithm using Bergel's framework.

```
1  engine := GAEngine new.
2  engine random: (Random seed: 11).
3  engine populationSize: 20.
4  engine numberOfGenes: 20.
5  engine createGeneBlock: [ :rand :index :ind |
6    (0 to: 1) atRandom: rand ].
7  engine fitnessBlock: [ :ind | ind count: [ :each | each = 1 ] ].
8  engine endIfFitnessIsAbove: 20.
9  engine run.
10 engine trace: 'Target fitness reached at generation '.
11 engine traceCr: engine logs last generationNumber.
12 engine trace: 'Best individual is: '.
13 engine traceCr: engine logs last fittestIndividual genes .
14 engine trace: 'with fitness: '.
15 engine traceCr: engine logs last bestFitness .
```

Breaking it down:

1 A new genetic algorithm engine is created.

2 The random seed is set to 11.

3-4 The population and number of genes are both set to 20.

5-6 The gene creation block is outlined for initial population generation, returning a random digit either 0 or 1.

[2] https://github.com/Apress/agile-ai-in-pharo/tree/master

7 The fitness block is then detailed, here counting the number of 1s in an individual.

8 The algorithm's termination condition is set, halting if an individual's fitness surpasses 20.

9 The algorithm is then executed.

10-18 The algorithm's logs are then printed to the console. Note that the engine keeps a log of the evolution process.

While Bergel's framework, as presented in the book, stands as a clear and potent genetic algorithm tool for *Pharo*, its primary intent is instructional. As such, it may not rival the robustness of seasoned frameworks like DEAP or *Jenetics*.

3.3 Distributed Evolutionary Algorithms in *Python* (DEAP)

Distributed Evolutionary Algorithms in Python (DEAP) [58] 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) [63].

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. In contrast, the `Toolbox` serves as a comprehensive repository for various operators necessary in evolutionary algorithms, including the evaluation, selection, mutation, and crossover functions.

The following illustrates the use of DEAP to solve the OMP problem using a genetic algorithm:

Listing 3.2– A simple genetic algorithm using DEAP.

```
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
creator.create("Individual", list, fitness=creator.FitnessMax)
toolbox = base.Toolbox()
toolbox.register("attr_bool", random.randint, 0, 1)
toolbox.register("individual", tools.initRepeat, creator.Individual, toolbox.attr_bool, n=
↪  20)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
toolbox.register("evaluate", lambda i: (sum(i),))
toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("mutate", tools.mutFlipBit, indpb=0.05)
toolbox.register("select", tools.selTournament, tournsize=3)
TARGET_FITNESS = 20
if __name__ == "__main__":
    random.seed(11)
    pop = toolbox.population(n=20)
    hof = tools.HallOfFame(1)
    stats = tools.Statistics(lambda i: i.fitness.values)
    stats.register("max", max)
    gen = 0
    while True:
        gen += 1
        offspring = algorithms.varAnd(pop, toolbox, cxpb=0.5, mutpb=0.2)
        fits = toolbox.map(toolbox.evaluate, offspring)
        for fit, ind in zip(fits, offspring):
            ind.fitness.values = fit
        pop = toolbox.select(offspring, k=len(pop))
        hof.update(pop)
        record = stats.compile(pop)
```

```
28          if record['max'][0] >= TARGET_FITNESS:
29              break
30      print(f"Target fitness reached at generation {gen}.")
31      print(f"Best individual is: {''.join(map(str, hof[0]))}")
32      print(f"with fitness: {hof[0].fitness.values[0]}")
```

When executed, the program outputs the following:

```
Target fitness reached at generation 12.
Best individual is: 11111111111111111111
with fitness: 20.0
```

This example demonstrates the use of DEAP to implement a simple genetic algorithm, with the added functionality to stop evolution once a target fitness is achieved.

Here is a breakdown of the code:

1-2 DEAP uses a meta-factory to *dynamically* create user-defined classes. Here, we create two classes: `FitnessMax` for the fitness (single-objective, to be maximized) and `Individual` to represent an individual in the population. The second argument to `create` is the base class, meaning that `FitnessMax` is a subclass of `base.Fitness` and `Individual` is a subclass of `list`.

3 The `Toolbox` is created, it is used to store most of the functions and arguments required to perform the genetic algorithm.

4-5 We register two functions in the toolbox: `attr_bool` which generates a random binary number (0 or 1), and `individual` which creates a new individual using the `attr_bool` function. The individual consists of 20 binary numbers (genes).

6-7 We then register two more functions: `population` which creates a population of individuals, and `evaluate` which evaluates an individual's fitness as the sum of its genes.

8-10 We register three more functions in the toolbox: `mate` for performing *two-point crossover*, `mutate` with a *bit-flip mutator* with a 5% probability, and `select` for performing tournament selection with a tournament size of 3.

11 We set the target fitness to 20 (since we aim to maximize the sum of the genes, the maximum possible fitness is 20)

12-32 In the main section of the code, we seed the random number generator to ensure reproducibility, create a population of 20 individuals, and set up the *Hall of Fame* and a statistics object to keep track of the maximum fitness in the population.

19-29 A while loop is used to run the genetic algorithm until the target fitness is reached. Inside the loop:

- 21-22 We apply crossover and mutation to the population using the `varAnd` function,[3] and then evaluate the fitness of the offspring
- 23-24 We assign the newly computed fitness values to the individuals.
- 25 We replace the old population with the selected individuals from the offspring.
- 26-27 We update the Hall of Fame and compile the statistics.
- 28-29 We check if the maximum fitness has reached the target fitness. If it has, we break the loop and the algorithm stops.

30-32 Finally, we print the results.

This example illustrates the key aspects of using DEAP: creating custom classes, setting up a toolbox, defining and registering functions, and manually controlling the loop of the genetic algorithm.

DEAP provides robust support for multi-objective algorithms and parallelization, which are common requirements in complex optimization problems. The library also includes a set of benchmark functions and examples to help users understand the various algorithms' behavior and performance.

[3] This function simply applies the variation operators to the population.

DEAP is widely regarded as one of the most comprehensive and cutting-edge genetic algorithm frameworks currently available, encompassing a wide range of algorithms, including GAs, GP, Evolutionary Strategies (ES), and more. It boasts a wealth of documentation and a robust community of users and contributors. However, certain facets of DEAP may pose challenges for users, as reflected in the preceding code example.

- **Toolbox Convention:** DEAP's verbose code and toolbox usage can confuse those new to evolutionary algorithms.
- **Flexibility vs. Usability:** DEAP offers flexibility but its dependence on code injection complicates static analysis and IDE integration.
- **Production Risks:** DEAP's code injection can introduce vulnerabilities if not handled properly.
- **Python's Typing:** The dynamic typing in Python, while flexible, can lead to runtime errors and harder-to-understand code.
- **Dependency on NumPy:** DEAP requires the NumPy library, introducing an additional dependency to manage.
- **Library Mastery:** Extending DEAP's capabilities demands a deep understanding of its architecture and often manual algorithm modifications.
- **Abbreviated Variables:** DEAP's use of abbreviated variable names, coupled with Python's keyword arguments, can hinder code interpretation without documentation.

Despite this, DEAP's flexibility and modularity have made it a popular choice among researchers and practitioners in the field of evolutionary computation.

3.4 Jenetics: Java Genetic Algorithm Library

Jenetics [77] is a robust EC library developed in *Java*. It is a comprehensive framework that provides a *wide range* of genetic algorithms and operators. The library is open-source and available under the *MIT* license [19].

Jenetics structure revolves around two main concepts: `Phenotype`[4] and `Engine`. The `Phenotype` is a representation of a single candidate solution to a given optimization problem. Meanwhile, the `Engine` is the core of the library, which controls the evolution process. It is responsible for the initialization of the population, the execution of the evolution, and the termination of the evolution once the termination condition is met (this uses a very similar approach to the one we presented in section 3.2 on page 32).

One thing that sets *Jenetics* apart from other libraries is the use of the *Java's Stream API* [74] to make a seamless integration between it and the `EvolutionStream` class used by the `Engine`. This means that the evolution has access to all the features provided by the Stream API, such as parallelization, filtering, and mapping.

Below, we demonstrate the use of *Jenetics* to solve the OMP with a genetic algorithm:

Listing 3.3– Solution to the OMP using *Jenetics*

```
public class OneMax {
  public static void main(String[] args) {
    RandomRegistry.random(new Random(11));
    final var engine =
        Engine.builder((Genotype<BitGene> gt) ->
                        gt.chromosome().as(BitChromosome.class).bitCount(),
                    BitChromosome.of(20, 0.5))
            .maximizing()
            .populationSize(20)
            .alterers(new Mutator<>(0.05), new SinglePointCrossover<>(0.5))
            .build();
    Phenotype<BitGene, Integer> best = engine.stream()
```

[4] See definition A.15 on page 99

```
13                                          .limit(Limits.byFitnessThreshold(19))
14                                          .collect(EvolutionResult.toBestPhenotype());
15      System.out.println("Target fitness reached at generation: " + best.generation());
16      System.out.println("Best individual is: " + best.genotype());
17      System.out.println("with fitness: " + best.fitness());
18    }
19  }
```

When executed, the program outputs the following:

```
Target fitness reached at generation: 40
Best individual is: [00001111|11111111|11111111]
with fitness: 20
```

This example demonstrates *Jenetics*' usage for implementing a straightforward genetic algorithm, with the added ability to stop evolution once steady fitness is achieved.

Here is an explanation of the code:

1-2 First, a class containing a `main` method is established. This forms the launching point of the program.

3 The `RandomRegistry` is then utilized to preset the random seed, a step that ensures consistency and replicability of program outputs.

4-11 The core configuration of the evolutionary engine occurs next, tailored specifically to the problem at hand.

5-7 An `Engine` is instantiated using the `Engine.builder` method, requiring two arguments: a `Function` and a `Genotype`. The `Function` assesses the fitness of a `Genotype` by tallying the quantity of `1`s in the `Genotype`. A `Genotype` is created via the `BitChromosome.of` method, which mandates the `length` of the chromosome and the `probability` of a `1` appearing within the chromosome. Here, `BitChromosome` is a distinct version of the `Chromosome` interface, used to depict a `Genotype` composed of `BitGenes`.

8 The `maximizing` method signals that the objective is to boost fitness to its maximum possible value.

9 The `populationSize` method sets the size of the evolutionary population.

10 The `alterers` method sets the genetic operations to be used in the evolution. Here, the `Mutator` and `SinglePointCrossover` are selected. The `Mutator` behaves similarly to a *bit-flip mutation* operator in this context.

11 Finally, the `build` method assembles the functional `Engine`.

12-14 Execution of the evolution process follows.

12 The `stream` method generates an `EvolutionStream` from the `Engine`, allowing for control over the evolution process.

13 The `limit` method sets the termination condition. The evolution halts once a `Phenotype` possessing a fitness **exceeding** 19 is located.

14 Lastly, the `collect` method gathers the evolution's output, in this case, capturing the optimal `Phenotype` discovered.

15-17 The final stage involves printing the evolution results to the console for review and analysis.

This example showcases the key aspects of using *Jenetics*: creating genotypes, setting up an engine, and controlling the evolution process.

Jenetics provides extensive support for multi-objective algorithms and parallel execution, which are common necessities in intricate optimization problems.

Recognized as one of the most comprehensive and cutting-edge genetic algorithm libraries today, *Jenetics* receives wide-ranging acclaim. However, certain aspects of *Jenetics* could be challenging for its users.

- **Java Limitations:** *Jenetics* is hindered by Java's verbosity and complexity. As a Java library, *Jenetics* unavoidably incorporates some of Java's limitations, notably its verbosity and complexity.
- **Documentation Gaps:** The library's documentation is not exhaustive and sometimes outdated. The library's documentation could be more comprehensive. Currently, some examples are outdated or have redundant implementations.
- **Seed Replicability:** The `RandomRegistry`'s claim on replicability conflicts with actual outcomes, indicating a documentation deficiency. The insufficient documentation is most evident in the use of the `RandomRegistry` to set the random seed, as seen in example line 3. The documentation claims this ensures the replicability of the program's output, but practical execution proves otherwise. More complete code samples and concrete use cases would ameliorate this issue.
- **Flexibility vs. Usability:** *Jenetics* offers adaptability but leans on builders and factories, sometimes obscuring the code. There is a delicate balance between the framework's flexibility and its usability. *Jenetics*' notable adaptability leans heavily on builders and factories, which can occasionally obfuscate the code.
- **Complex Problem Handling:** Addressing intricate problems in *Jenetics* introduces multiple complexities in the code. Complex problem-solving with *Jenetics* can quickly complicate the code, introducing *codecs*, *proxies*, and the `Problem` interface.
- **Visualization Gap:** The library lacks a comprehensive mechanism for visualizing the evolutionary process. There's a lack of a robust mechanism for visualizing the evolution process.
- **Well typed Jenetics programs can raise errors on runtime:** *Jenetics* sometimes requires explicit type casting, which introduces errors at runtime.

Despite these shortcomings, *Jenetics* is a powerful and flexible library that can be used to solve a wide variety of optimization problems. The library is actively maintained, and it is one of the most popular evolutionary computation libraries available today.

3.5 ECJ: A Java-based Evolutionary Computation Research System

The ***Evolutionary Computation in Java*** (ECJ) [60] is a powerful and flexible framework for conducting research and experiments in the field of evolutionary computation. ECJ has been under active development for more than two decades, and during this period, it has grown into one of the most comprehensive open-source[5] libraries for evolutionary computation.

What sets ECJ apart from other libraries is its ability to handle a broad range of evolutionary computation paradigms. It offers support for genetic algorithms, genetic programming, multi-objective optimization, co-evolution, and many more. It also comes with built-in functionality for distributed computing, allowing researchers to harness the power of large-scale computing clusters for their experiments.

The ECJ library is designed with flexibility and extensibility in mind. It employs a design pattern that uses parameter files for configuration, thus enabling researchers to customize the algorithms according to their specific needs without having to modify the source code. The library is highly modular, and its components can be easily replaced or extended.

The following code sample demonstrates how to solve the *OneMax* problem using ECJ:[6]

Listing 3.4– ECJ's solution to the OMP

```
1 public class OneMax extends Problem implements SimpleProblemForm {
2   @Override
3   public void evaluate(final EvolutionState state,
4                        final Individual ind,
5                        final int subpopulation,
6                        final int threads) {
7     if (ind.evaluated) {
8       return;
```

[5] No license is provided with the ECJ source code.
[6] We couldn't find a way to set the random seed in ECJ, so the results are not reproducible.

```
9       }
10      if (!(ind instanceof BitVectorIndividual)) {
11        state.output.fatal("Individual must be a BitVectorIndividual")
12      }
13      int sum = 0;
14      // This is necessary to safe-cast ind to BitVectorIndividual
15      BitVectorIndividual ind2 = null;
16      if (ind instanceof BitVectorIndividual) {
17        ind2 = (BitVectorIndividual) ind;
18      }
19      if (ind2 != null) {
20        for (int x = 0; x < ind2.genome.length; x++) {
21          sum += (ind2.genome[x] ? 1 : 0);
22        }
23
24        if (!(ind2.fitness instanceof SimpleFitness)) {
25          state.output.fatal("Fitness must be a SimpleFitness")
26        }
27        ((SimpleFitness) ind2.fitness)
28            .setFitness(state,
29                /* fitness = */ sum / (double) ind2.genome.length,
30                /* isIdeal = */ sum == ind2.genome.length);
31        ind2.evaluated = true;
32      }
33    }
34  }
```

An explanation of the code:

1 The class `OneMax` extends `Problem` and implements `SimpleProblemForm`, which is ECJ's interface for problems that can be solved with generational evolutionary algorithms.

2-6 The `evaluate` method is overridden from the `Problem` class. It is responsible for evaluating the fitness of an `Individual` (a candidate solution in the population). The arguments include the current state of the evolution, the individual to be evaluated, the subpopulation to which the individual belongs, and the number of the thread executing this method (useful in a multithreaded setting).

7-9 The method checks if the individual has already been evaluated. If it has, then the method immediately returns to avoid unnecessary computation.

10-12 If the individual is not an instance of `BitVectorIndividual` (which represents individuals whose genome consists of bits), a fatal error is reported. The OneMax problem assumes a bit string representation of individuals.

13-18 Variable `sum` is initialized to keep track of the total number of 1s in the individual's bit string representation. The individual is then safe-casted to `BitVectorIndividual`. This is necessary because the `ind` argument is of type `Individual`, but we need to work with its `BitVectorIndividual` specifics (such as its `genome` property).

20-22 If the casting is successful, a loop iterates through the genome (bit string) of the individual, incrementing `sum` for every bit set to 1.

24-26 If the fitness of the individual is not an instance of `SimpleFitness` (which represents a single floating-point fitness value), a fatal error is reported.

27-30 The individual's fitness is then set to the proportion of bits set to 1 in the bit string (`sum / (double) ind2.genome.length`). If all bits are set to 1 (i.e., `sum` equals the length of the genome), then the individual is marked as ideal.

31 Finally, `ind2.evaluated` is set to `true`, indicating that the individual has been evaluated.

To run the OneMax problem, we need to create a parameter file that specifies the problem, the evolutionary algorithm, and the parameters of the algorithm. The following is a sample parameter file for the OneMax problem:

Listing 3.5– Configuration file for the OMP

```
breedthreads = 1
evalthreads = 1
seed.0 = 11
eval.problem = ec.app.onemax.OneMax
pop.subpop.0.species.ind = ec.vector.BooleanVectorIndividual
pop.subpop.0.species.genome-size = 20
pop.subpop.0.species.fitness = ec.FitnessSimple
pop.subpop.0.size = 20
pop.subpop.0.species.pipe = ec.vector.breed.VectorMutationPipeline
pop.subpop.0.species.pipe.source.0 = ec.vector.breed.VectorCrossoverPipeline
pop.subpop.0.species.pipe.source.0.prob = 0.5
pop.subpop.0.species.pipe.source.0.source.0 = ec.select.TournamentSelection
pop.subpop.0.species.pipe.source.0.source.1 = ec.select.TournamentSelection
eval.problem.terminateFitness = 20
```

This is a parameter file for ECJ, which specifies various configuration settings for an evolutionary computation run.

1-2 `breedthreads` and `evalthreads` determine the number of threads to be used for breeding and evaluation operations, respectively. Both are set to 1, which means a single-threaded operation.

3 `seed.0` sets the initial seed for the random number generator. Different seeds will lead to different runs, even with identical configuration parameters.

4 `eval.problem` sets the problem to be solved, in this case, `ec.app.onemax.OneMax`, which refers to the OneMax problem class implemented earlier.

5-6 `pop.subpop.0.species.ind` sets the type of individual in the population. `ec.vector.BooleanVectorIndividual` means that each individual will be a Boolean vector, which is appropriate for the OneMax problem. `pop.subpop.0.` ⌟ `species.genome-size` sets the length of the Boolean vector to 20.

7-8 `pop.subpop.0.species.fitness` sets the type of fitness to be used. `ec.FitnessSimple` represents a single floating-point fitness value. `pop.subpop.0.size` sets the population size to 20.

9 `pop.subpop.0.species.pipe` defines the pipeline for breeding new individuals. `ec.vector.breed.VectorMutationPipeline` means a mutation operation will be applied.

10 `pop.subpop.0.species.pipe.source.0` sets the source of individuals for the mutation operation to be a crossover operation
(`ec.vector.breed.VectorCrossoverPipeline`), with a crossover probability of 0.5.

12-13 `pop.subpop.0.species.pipe.source.0.source.0` and `pop.subpop.0.species.pipe.source.0.source.1` set the source of individuals for the crossover operation to be tournament selection (`ec.select.TournamentSelection`).

14 `eval.problem.terminateFitness` sets the fitness level at which the evolutionary run should terminate. When an individual reaches a fitness of 20, the run will stop, implying that a perfect solution (all bits set to 1) has been found for the OneMax problem.

To run the OneMax problem, we need to execute the following command:

Listing 3.6– Running the OMP with ECJ

```
java -cp .
ec.Evolve -file .\ec\app\onemax\one_max.properties
```

This command will run the OneMax problem with the parameter file `one_max.properties` in the package `ec.app.onemax`. According to the official ECJ documentation, this command should be able to solve the OneMax problem, however, we were unable to get it to work. The following is the output of the command:

Listing 3.7– Output of the command in listing 3.6

```
# Code omitted for brevity
Exception in thread "main" ec.util.ParamClassLoadException:
No class name provided.
PARAMETER: state
        at ec.util.ParameterDatabase.getInstanceForParameter(ParameterDatabase.java:493)
        at ec.Evolve.initialize(Evolve.java:479)
        at ec.Evolve.initialize(Evolve.java:412)
        at ec.Evolve.main(Evolve.java:758)
```

The error message indicates that the parameter `state` is missing. This parameter is required by ECJ, but is not specified in the parameter file. We tried to add the parameter to the file, but the error persisted. We also tried to run the OneMax problem with the parameter file provided by the ECJ documentation, but the error persisted. We were unable to find a solution to this problem.

While ECJ stands as one of the most comprehensive and sophisticated Evolutionary Computation (EC) frameworks available, it presents users with a daunting learning curve. We've identified several areas of concern that contribute to this complexity:

- **Verbose and obscure code:** The framework's verbosity hinders code comprehension. To illustrate, the `eval` method in the `OneMax` class spans 31 lines, yet only 3 of these lines (20-22) actually pertain to the OneMax problem itself—the rest are framework-related.
- **Complex configuration:** Setting up the framework involves numerous layers of boilerplate code, necessitating a deep understanding of the underlying concepts and terminology. Lines such as `pop.subpop.0.species.pipe.source.0.source.0` can be particularly opaque to newcomers.
- **Outdated documentation and implementation:** The `README.md` file has not been updated in 4 years, and the OneMax class equivalent was last refreshed 5 years ago. Even the most recent version (27) of the framework dates back 4 years, rendering much of the documentation and implementation obsolete.
- **Legacy Java syntax:** ECJ's reliance on older Java syntax, specifically the lack of generics, results in frequent casting and `instanceof` checks. This not only makes the code more brittle but also complicates understanding.
- **Exclusion from popular package managers:** The absence of ECJ from widely-used package managers like *Maven* or *JitPack* imposes additional burdens on integration with other projects, as manual download and compilation become necessary.
- **Limited IDE compatibility:** The design of ECJ emphasizes command line compilation and execution, undermining the utility of modern Integrated Development Environments (IDEs) and complicating debugging efforts.
- **Inconvenience for integration:** Rather than being crafted for use as a library, the framework is designed with the expectation of direct source code inclusion. This requirement to import source code into users' own projects increases the difficulty of integration.
- **Generic name complicates research:** The term "ECJ" is sufficiently common to cause confusion when seeking information on the framework. Queries such as "ECJ" or "ECJ Java" yield results concerning the *European Court of Justice* or the *Eclipse Compiler for Java*, respectively, rather than the intended framework.
- **Limited community and resources:** As of 2023, ECJ lacks a robust community of developers or users. The dearth of online resources like tutorials or Stack Overflow discussions compounds the challenges for new users seeking to leverage the framework effectively.
- **Non-utilization of modern Java features:** Beyond the lack of generics, the framework fails to harness modern Java features like lambda expressions, lists, or Optional types. These features, by enabling more concise and readable code, can significantly enhance error prevention.

In conclusion, *the Evolutionary Computation in Java* (ECJ) framework, while extensive and capable, poses considerable barriers to entry for users due to its complexity, outdated documentation, and heavy reliance on legacy *Java* syntax. Furthermore, despite its versatility in supporting various evolutionary computation paradigms and distributed computing, the steep learning curve, absence from popular package managers, and limited compatibility with modern IDEs make it challenging for newcomers and experienced developers alike. The verbose code, intricate configuration requirements, and limited use of modern *Java* features also contribute to the difficulty in understanding and integrating the ECJ into projects. Our exploration further exposed issues with the execution of sample problems, highlighting the need for updated documentation and maintenance. However, ECJ's modular design, extensibility, and breadth of capabilities maintain its status as a valuable tool for research in the field of evolutionary computation. Future efforts could focus on modernizing the framework, improving documentation, and fostering an active user community to mitigate the identified issues and enhance its usability.

3.6 GeneticSharp: Comprehensive Overview of a .NET Genetic Algorithm Library

GeneticSharp is a high-performance, extensible, cross-platform *C#* Genetic Algorithm library tailored for multi-threading. This open-source library, designed to streamline the integration of genetic algorithms into applications, is licensed under the *MIT License* [19].

GeneticSharp revolves around three foundational concepts: `Chromosome`, `Population`, and `GeneticAlgorithm`:

- `Chromosome` serves as the foundation for various chromosome types, responsible for gene storage and genetic operator management.
- `Population` maintains a collection of chromosomes undergoing evolution.
- `GeneticAlgorithm` acts as the algorithm's primary driver, encompassing its main loop.

Distinctively, *GeneticSharp* capitalizes on parallel computing, accelerating fitness evaluation, crossover, and mutation. Furthermore, its extension-oriented design simplifies the addition of new operators or further customizations to the genetic algorithm.

Consider the following example, which demonstrates solving the OMP using *GeneticSharp*:

Listing 3.8– Solving OMP with *GeneticSharp*

```
public sealed class BinaryChromosome : BinaryChromosomeBase {
    public BinaryChromosome(int length) : base(length) {
        CreateGenes();
    }
    public override IChromosome CreateNew() {
        return new BinaryChromosome(Length);
    }
}
public class OneMaxFitness : IFitness {
    public double Evaluate(IChromosome chromosome) {
        return chromosome.GetGenes().Count(gene => (int)gene.Value == 1);
    }
}
public class ReproducibleRandom : RandomizationBase {
    private static readonly object GlobalLock = new();
    private static readonly ThreadLocal<Random?> ThreadRandom = new(NewRandom);
    private static Random? Instance => ThreadRandom.Value;
    private static Random NewRandom() {
        lock (GlobalLock) {
            return new Random(11);
        }
    }
    public override int GetInt(int min, int max) {
```

```
24          Debug.Assert(Instance != null, nameof(Instance) + " != null");
25          return Instance.Next(min, max);
26      }
27      public override float GetFloat() {
28          Debug.Assert(Instance != null, "ReproducibleRandom.Instance != null");
29          return (float)Instance.NextDouble();
30      }
31      public override double GetDouble() {
32          Debug.Assert(Instance != null, "ReproducibleRandom.Instance != null");
33          return Instance.NextDouble();
34      }
35  }
36  const int chromosomeLength = 20;
37  const int populationSize = 20;
38  RandomizationProvider.Current = new ReproducibleRandom();
39  var selection = new EliteSelection();
40  var crossover = new UniformCrossover();
41  var mutation = new FlipBitMutation();
42  var fitness = new OneMaxFitness();
43  var chromosome = new BinaryChromosome(chromosomeLength);
44  var population = new Population(populationSize, populationSize, chromosome);
45  var ga = new GeneticAlgorithm(population, fitness, selection, crossover, mutation) {
46      Termination = new FitnessThresholdTermination(chromosomeLength)
47  };
48  ga.Start();
49  Console.WriteLine($"Target fitness reached at generation: {ga.GenerationsNumber}");
50  var bestChromosome = ga.BestChromosome as BinaryChromosome;
51  Console.WriteLine($"Best individual is: {bestChromosome}");
52  Console.WriteLine($"with fitness: {bestChromosome?.Fitness}");
```

Dissecting the code reveals the following:

1-8 The `BinaryChromosome` class is sealed[7] to inhibit subclassing and encapsulates a chromosome with binary genes. Sealing is an advisable practice here due to the constructor's invocation of the virtual `CreateGenes` method, thus eliminating potential ambiguities from subclass method overrides.

2-4 The constructor, accepting a length parameter, invokes the base constructor. Given that the base may call a virtual method, sealing thwarts unforeseen behaviors from subclasses unknowingly overriding it.

4-7 `CreateNew` method proffers a fresh `BinaryChromosome` instance retaining the original length.

9-13 `OneMaxFitness` class quantifies a chromosome's fitness.

10-12 `Evaluate` method tallies genes equating to 1 (equivalent to binary `true`), serving as the OMP's fitness metric.

14-35 The `ReproducibleRandom` class furnishes reproducibly random numbers, critical for consistent genetic algorithm outcomes across iterations.

15-17 A global lock alongside a thread-local `Random` instance (`ThreadRandom`) ensures synchronized seeding across threads for consistent randomness.

18-22 `NewRandom` method spawns a `Random` object seeded with 11, bolstered by the `lock` keyword for thread-safety.

23-34 Methods `GetInt`, `GetFloat`, and `GetDouble` yield values from the `Random` instance.

36-37 Constants pertinent to the genetic algorithm, such as chromosome length and population size, are delineated.

[7] Details on definition A.18 on page 99.

38 Randomization defaults to the `ReproducibleRandom` class.

39-44 Configuration lines dictate the genetic algorithm's components, encompassing selection (elite), crossover (uniform), mutation (flip bit), and the fitness function (`OneMaxFitness`).

45-47 The genesis population and genetic algorithm instance are instantiated.

46 The algorithm's cessation is predicated on a chromosome attaining peak fitness, tantamount to its length.

48 The genetic algorithm is set in motion.

49-52 Outputs include the generation count to attain target fitness, the paramount chromosome, and its corresponding fitness.

Despite its many advantages, *GeneticSharp* also has certain limitations:

- **Overwhelming verbosity:** The library's detailed nature can overshadow its functionality, especially in contrast to succinct frameworks like Jenetics.
- **Externalized fitness function:** The architecture demands that fitness functions be isolated in distinct classes. This design choice might make trivial problems seem more convoluted than when using, for example, higher-order functions.
- **Sparse documentation:** Compared to competitors like DEAP, the library lags in both instructional content and practical examples. Novices might find this shortage hampers their initial engagement with the library.
- **Parallelization trade-offs:** Incorporating parallel functionality amplifies the complexity of defining custom genetic operators. Although parallel processing boosts execution speed, developers need to grapple with the intricacies it introduces.
- **Advanced C# intricacies:** Features unique to advanced *C#* such as the `sealed` attribute can disorient beginners. New users might be deterred by such advanced constructs, especially without prior experience in *C#*.
- **Developer-centric focus:** *GeneticSharp* caters predominantly to application creation, potentially sidelining academic researchers. Scholars deeply entrenched in genetic algorithm studies might find the library's emphasis misaligned with their interests.
- **Budding community support:** The fledgling nature of the *GeneticSharp* community in 2023 could impede access to resources and prompt support. This nascent state might affect user engagement, library updates, and peer assistance.
- **Conventional algorithm scope:** The library's focus remains anchored to traditional genetic algorithms, omitting newer evolutionary algorithm derivations. Practitioners might find the lack of support for paradigms like genetic programming limiting.

In summary, *GeneticSharp* offers a dynamic, parallelized platform for crafting genetic algorithms in *C#*. Its customizability and developer-centric approach make it a compelling choice for those aiming to harness genetic algorithms within their software projects. Yet, users must be cognizant of its more intricate API, the nuances introduced by parallelization, and potential challenges arising from a smaller community and limited documentation. Ultimately, while *GeneticSharp* solidifies its position as a valuable tool within the *.NET* arena, its optimal utilization depends on aligning its capabilities with the user's specific requirements and familiarity with genetic algorithms.

3.7 Other Libraries

In the preceding sections, we detailed a select set of premier frameworks within EC. However, the scope of our research extended beyond this list to a wider array of libraries. This comprehensive survey was instrumental in shaping our proposed *Kotlin*-based EC framework. Through understanding the strengths and peculiarities of each library, we aimed to incorporate their best features while sidestepping prevalent pitfalls.

Below, we provide a concise overview of some additional frameworks:

- **EvolvingObjects** (EO) [61]: An object-oriented *C++* framework. It offers various evolutionary algorithms and operators, including GAs, ES, and Particle Swarm Optimization (PSO). Notably, certain aspects, like GP, are not currently supported.
- **Inspyred** [64]: A Python-focused framework inspired by De Jong [18]. It clearly distinguishes between algorithmic computations and problem-specific ones. While it supports a range of evolutionary algorithms, GP is not among them.

- **Pyevolve**: A Python-based framework that provides a gamut of evolutionary algorithms, including GAs and GP. However, it is currently inactive.
- **PGAPack** [71]: Crafted in *C*, it is a parallel genetic algorithm library. It offers compatibility with *Fortran* and *C++* and a myriad of parallel architectures, it also features a *Python* interface [72].
- **pagmo** [48]: A *C++* library that prioritizes parallel optimization. It delivers a cohesive interface for various optimization algorithms and provides a *Python* interface [50].
- **easy_ga** [59]: Developed in *Rust*, this framework simplifies GA prototyping, primarily focusing on traditional GAs.
- **genevo** [62]: Another *Rust* framework, it majorly supports classic GAs.
- **Evolutionary Computation Framework** (ECF) [43]: ECF is a *C++* framework that provides a comprehensive set of evolutionary algorithms. Its depth rivals that of DEAP and ECJ. Nonetheless, it faces challenges like limited documentation and a challenging learning curve.

The breadth and capabilities of these libraries highlight the vast potential within the domain of genetic algorithms. Our exhaustive exploration empowers us to develop a *Kotlin*-based framework that amalgamates the strengths of each while pioneering unique features.

Chapter 4

The Keen Framework

4.1 Introduction

The field of *evolutionary computation* (EC) is rich with diverse methodologies and algorithms, each proving effective for complex optimization challenges. However, applying these techniques in practice often involves repetitive coding and lacks a straightforward, user-friendly interface. To tackle these issues, this chapter presents ***Keen***, a versatile genetic algorithms framework created in *Kotlin*.

Keen aims to make evolutionary algorithms more accessible and user-friendly. It primarily focuses on streamlining the implementation of *genetic algorithms* (GAs) and *genetic programming* (GP), while also providing a foundation for extending to other evolutionary algorithms. This approach is intended to encourage experimentation and innovation in the EC field, facilitating the development of novel algorithms and techniques.

This chapter delves into the *Keen framework*, detailing its structure, key components, and functionalities. It covers the range of genetic operators in Keen, including selection, mutation, and crossover methods. The chapter also explores GAs and GP techniques within Keen, highlighting their unique features and potential applications.

Additionally, the chapter explores how Keen is built for easy expansion and adaptation, encouraging innovation and evolution in line with the EC field's growth. We hope that Keen will foster wider application of these algorithms.

4.2 Architecture

Keen, organizes its architectural blueprint into five distinguishable modules: *Evolution*, *Genetic Material*, *Genetic Operators*, *Utility*, and *Programs*.

The ***Utility*** module serves as the library's toolbox, providing various auxiliary classes and functions that the remaining modules can utilize. As its content is primarily supplementary, it won't be thoroughly examined within this document.

The ***Genetic Material*** module encapsulates the foundational elements defining the characteristics of our evolving entities or `Individuals`. Its core constituents are the `Individual` and `Genotype` classes, accompanied by the encompassing `Chromosome` and `Gene` hierarchies.

Within the ***Genetic Operators*** module, we define the different genetic operators that the *Evolution Engine* leverages to stimulate the evolutionary process. This module accommodates classes embodying crossover, mutation, and selection operators, systematically grouped into corresponding categories: **Crossover**, **Mutation**, and **Selection**.

The ***Evolution*** module houses classes that construct the engine of evolution itself, along with essential classes promoting the smooth progression of the evolution process.

Finally, the emerging ***Programs*** module demonstrates genetic programming techniques with various programs created using this approach. Since it's still in the early development phase, this document will not explore it extensively.

A majority of the *Keen* classes are parameterized by the type of data they handle and the gene type composing the population of individuals, like an integer or an integer gene. This design choice lends *Keen* a wide range of flexibility, enabling users to tailor the genetic material for the evolution engine.Moreover, thanks to *Kotlin*'s type inference, we can generally omit type parameters, thereby boosting code legibility.

Keen upholds the principle of immutability, rendering most of its classes immutable. This provides an assurance of robustness and predictability, as objects remain unaltered post-creation, enhancing parallelization potential by allowing shared memory use without the need for synchronization mechanisms.

Furthermore, *Keen* offers factory methods for most classes, fostering an intuitive and straightforward programming experience. It employs language-oriented programming techniques to form a domain-specific language (DSL), streamlining object creation with more readable code. This approach is evident in libraries such as *Kotlinx.html* [47], *Kotlin Telegram Bot* [46], and *Kotest* [66]. While we promote factory methods as the primary mode of object creation, the design caters to manual object creation, granting users control over the process when needed.

Here's an illustrative DSL example, where we create a `Genotype` for the *One Max* problem. Here, each `Chromosome` represents a sequence of bits.

Listing 4.1– Usage of the *Keen* Domain Specific Configuration Language to create a `Genotype` for the *One Max* problem.

```
genotypeOf {        // Creates a genotype
    chromosomeOf {  // Creates a chromosome
        booleans {  // of booleans
            size = 20
            trueRate = 0.15
        }
    }
}
```

The approach we see here closely mirrors the use of configuration files in ECJ. However, as our DSL is defined entirely within *Kotlin*, it offers significant advantages. These include type safety, robust Integrated Development Environment (IDE) support, and the ability to leverage the full capabilities of the language in defining our genotypes.

Furthermore, we claim that our approach enhances the readability of the code. Compared to using ECJ's configuration files, the DSL presents a more intelligible and cleaner interface. By defining the configurations in the same language and even within the same files as the rest of the code, we significantly improve code readability. This consolidation offers a more streamlined, unified development environment that facilitates easier understanding and manipulation of the codebase.

In the following sections, we will delve deeper into the pivotal classes within these modules and examine their functions and significance.

4.3 Genetic Algorithms

As we examine the framework in detail, it's essential to understand its key components. Although the general concepts of GAs are outlined in section 2.2 on page 6, here we focus on their specific implementation and design within our framework. We'll look at three main areas: the structure of the ***genetic material***, the workings of the ***evolution engine***, and the function of the ***evolution listeners***. Each part will be explained in detail, showing how they work together to make the GAs effective for the framework's objectives.

4.3.1 Genetic Material

The genetic material stands at the heart of GAs, representing the problem domain's primary aspect. Consequently, it's essential to structure this material efficiently and flexibly.

Our approach is "gene-centric". Rather than relying on indirect representations, the genetic material is depicted as a collection of genes. This approach makes our API design more flexible. For example, the genetic material can be a single gene, a group of genes (known as a chromosome), or multiple groups of genes (a genotype).

Furthermore, the genetic material is immutable. This immutability ensures thread safety and adheres to the "copy-on-write" principle, both of which are indispensable in providing a robust framework.

Gene The gene stands as the fundamental unit of genetic material within our framework, representing the minutest unit that can undergo manipulation.

Fundamentally, each gene must be capable of: 1. *mutation*, 2. *reproduction*, and 3. *information storage*. To promote extensibility, genes are designed to handle minimal operations, passing most tasks to operators (refer to section 4.4 on page 51). This approach ensures the framework's adaptability across diverse genetic materials, including binary, integer, real, and custom types.

Thus, we suggest the following interface for gene implementation:

Listing 4.2– Gene interface

```
interface Gene<T, G> where G : Gene<T, G> {
    val value: T
    fun mutate(): G
    fun duplicateWithValue(value: T): G
}
```

Chromosome A chromosome is defined as an iterable collection of genes. Its primary role is to offer a streamlined way to manipulate a group of genes. It's crucial to keep the chromosome's responsibilities minimal to foster adaptability across diverse genetic materials.

To this end, we suggest the following interface for each chromosome:

Listing 4.3– Chromosome interface

```
interface Chromosome<T, G> : Collection<G> where G : Gene<T, G> {
    val genes: List<G>
    fun duplicateWithGenes(genes: List<G>): Chromosome<T, G>
    interface Factory<T, G> where G : Gene<T, G> {
        fun make(): Chromosome<T, G>
    }
}
```

Incorporated within is the `Factory` interface, dedicated to the creation of new chromosomes. This is harnessed by the framework whenever there's a need to generate new chromosomes.

Genotype Like a chromosome, the genotype in our system is an iterable collection of chromosomes. While chromosomes and genotypes have similar functions, the genotype has a wider scope. In our framework, it's important to differentiate between the two, as this allows for handling multiple chromosomes at once. This separation enhances our framework's flexibility and supports the use of more complex genetic materials.

Utilizing parametric polymorphism, we negate the necessity for a hierarchical type relationship between genotypes and chromosomes. Consequently, the genotype can be represented by a concrete class, as demonstrated below:

Listing 4.4– Genotype class

```
data class Genotype<T, G>(val chromosomes: List<Chromosome<T, G>>) :
    Collection<Chromosome<T, G>> where G : Gene<T, G> {

    class Factory<DNA, G : Gene<DNA, G>> {
        fun make(): Genotype<DNA, G> { ... }
    }
```

```
}
```

Embedded within is the `Factory` class, designated for the generation of new genotypes.

Note this implementation does not depend on the `Chromosome` type, thus allowing for the use of any chromosome implementation that adheres to the *Gene* type. For example, one could have:

Listing 4.5– Genotype class with custom chromosome

```
genotypeOf {
    chromosomeOf {
        intImpl1 { ... }
    }
    chromosomeOf {
        intImpl2 { ... }
    }
}
```

where `intImpl1` and `intImpl2` are custom implementations of the `Chromosome` interface. Thus allowing more flexibility in the design of the genetic material.

Individual An individual in our framework represents the actual expression of the genetic material. It acts as the practical form of the genotype and symbolizes a potential solution. Essentially, an individual is defined by two key elements:

1. A *fitness* value,
2. The corresponding *genotype*.

In light of this, the structure for the individual is delineated as:

Listing 4.6– Individual class

```
data class Individual<T, G>(
    val genotype: Genotype<T, G>,
    val fitness: Double = Double.NaN
) where G : Gene<T, G> { ... }
```

4.3.2 Ranker

The ***ranker*** is a novel component of the framework responsible for ranking the population. The main purpose of this component is to compare individuals and determine their relative fitness. The ranker is a crucial component of the evolutionary process, as it is the primary source of information for the selection process.

The interface for the rankers is illustrated in listing 4.7.

Listing 4.7– Ranker interface

```
interface IndividualRanker<T, G> where G : Gene<T, G> {
    operator fun invoke(first: Individual<T, G>, second: Individual<T, G>): Int
    fun fitnessTransform(fitness: List<Double>)
}
```

Here, the ***ranker*** acts as a function that receives two individuals and returns an integer representing their relative fitness.

The ***ranker*** is also responsible for transforming the fitness values of the population in case that it is necessary. For example, the *roulette wheel* selection assigns a higher probability of being selected to individuals with higher fitness values, this is not suitable for minimization problems, hence the fitness values must be transformed.

The framework currently provides two rankers: the *Fitness Max Ranker*, that ranks individuals in descending order of fitness, and the *Fitness Min Ranker*, that ranks individuals in ascending order of fitness.

4.3.3 Evolution Engine

The ***evolution engine*** is the core of our framework, managing the population's evolution. It includes all necessary functions for the evolutionary process. This engine combines the basic structure of the evolutionary algorithms and uses functions from related parts like the ***genetic material*** and ***genetic operators***.

Specifically, the engine manages:

1. Population initialization;
2. Population evaluation;
3. Selection of individuals for the forthcoming generation;
4. Genetic operator application on chosen individuals;
5. Evaluation of the newly formed individuals;
6. Supplanting the old population with the new;
7. Iteration from step 3 until meeting termination criteria.

Furthermore, the engine offers hooks for both pre and post-processing steps related to the population and individuals. These hooks facilitate user-driven customization of the evolution trajectory. Although this feature remains outside this work's scope, its availability merits acknowledgment.

The engine's architecture is depicted in listing 4.8.

Listing 4.8– Evolution engine structure

```
class EvolutionEngine<T, G>(
    populationConfig: PopulationConfig<T, G>,
    selectionConfig: SelectionConfig<T, G>,
    alterationConfig: AlterationConfig<T, G>,
    evolutionConfig: EvolutionConfig<T, G>,
) : Evolver<T, G> where G : Gene<T, G> {

    private var state: EvolutionState<T, G> = EvolutionState.empty(ranker)

    override fun evolve(): EvolutionState<T, G> {
        do {
            state = iterateGeneration(state)
        } while (limits.none { it(state) })
        return state
    }

    fun iterateGeneration(state: EvolutionState<T, G>): EvolutionState<T, G> {
        val interceptedStart = interceptor.before(state)
        val initialPopulation = startEvolution(interceptedStart)
        val evaluatedPopulation = evaluatePopulation(initialPopulation)
        val parents = selectParents(evaluatedPopulation)
        val survivors = selectSurvivors(evaluatedPopulation)
        val offspring = alterOffspring(parents)
        val nextPopulation = survivors.copy(
            population = survivors.population + offspring.population
        )
```

```
        val nextGeneration = evaluatePopulation(nextPopulation)
        val interceptedEnd = interceptor.after(nextGeneration)
        return interceptedEnd.copy(generation = interceptedEnd.generation + 1)
    }
    ...

    class Factory<T, G>(...) where G : Gene<T, G> {
        ...
        fun make() = EvolutionEngine(
            populationConfig = PopulationConfig(genotypeFactory, populationSize),
            selectionConfig = SelectionConfig(
                survivalRate, parentSelector, survivorSelector
            ),
            alterationConfig = AlterationConfig(alterers),
            evolutionConfig = EvolutionConfig(
                limits, ranker, listeners, evaluator.creator(fitnessFunction), interceptor
            )
        )
    }
}
```

One of the key design decisions of the engine is to make it store as little information as possible. This decision is motivated by the fact that the engine is the most complex component of the framework, and thus, it should be as lightweight as possible to reduce its reasons for change and increase its maintainability.

The only information stored by the engine is it's current state, which is composed of the current generation, the population, and the ranker used to evaluate the population. This state is passed as a parameter to all the functions that need it, and the result of these functions is a new state.

Instead of depending on the engine to maintain information about the evolution process, the framework relies on the ***evolution listeners*** to store information about the evolution process.

The supplementary `Factory` class streamlines the engine configuration process. Although its utilization is recommended for engine instantiation, it remains optional, ensuring flexibility for users.

Finally, the `Evolver` interface is designed to accommodate varying engine implementations, with the `EvolutionEngine` class being the exclusive existing implementation to date.

4.3.4 Evolution Listeners

The *Keen* framework introduces an `EvolutionListener` interface designed with an *observer pattern* to offer users insights into the evolution process in real-time. This interface provides hooks for various events in the evolution cycle, ranging from its commencement and culmination to the beginning and conclusion of each phase, including generation, evaluation, and selection, among others.

The primary objective of the `EvolutionListener` is to empower users to oversee the evolution process, leveraging this data for diverse purposes. Users can employ this interface to conduct a detailed analysis of the evolution process, generate comprehensive reports based on its results, or persistently store the best-performing individual of each generation.

Within the *Keen* framework, several concrete implementations of this interface are available:

- `EvolutionPlotter`: Visualizes the best, worst, and average fitness throughout the generations in a plot format.
- `EvolutionPrinter`: Outputs essential evolution data every n generations, detailing aspects such as fitness metrics, the top-performing individual, and the progression of generations.
- `EvolutionSummary`: Prints a comprehensive summary post-evolution, encapsulating fitness statistics, leading individuals, generation count, time metrics for each evolution phase, and the evolution's total duration.

Remark. *The **Evolution plotter** uses Lets-Plot Skia Frontend [65] to render the plots, which is a multi-platform plotting library for Kotlin, meaning that the plots can be rendered in any platform supported by Kotlin Multi-Platform.*

The implementation of this interface is a novel feature of the *Keen* framework, and it is not present in the frameworks that were used as a reference for this work.

4.4 Genetic Operators

Genetic operators are the driving force behind the evolution process in any genetic algorithm or genetic programming paradigm. These operators stimulate change in the population and guide the search towards optimal solutions.

This section explains the basic genetic operators in *Keen*, including *Selection*, *Mutation*, and *Crossover*.

Each of these categories encompasses several techniques, providing *Keen* with a versatile repertoire of genetic operators. The subsequent subsections provide a comprehensive discussion on each of these operators, elucidating their working principles and application contexts.

Before delving into the various genetic operators, we must take into consideration the special case where individuals are programs (specifically trees). Keen represents this individuals in a way that each gene is a program tree, this means that operators that work on the chromosome level will not affect the individuals. For this reason, Keen provides special operators for this case, which are described in section 4.5.2 on page 63.

4.4.1 Selection

Building upon the concepts presented in definition 2.5 on page 9, the ***selector operator*** is mathematically represented as $\Sigma : \mathbb{P} \times \mathbb{N} \times \cdots \to \mathbb{P}$. At its core, the selector operator processes a given population P, selects n individuals based on specific criteria, and outputs a subset of the initial population. To ensure adaptability, the design challenge centers around establishing a versatile interface to cater to diverse selection strategies.

A promising approach consists of introducing an interface encompassing a method that requires the population and desired number of individuals as inputs. We can incorporate any supplementary parameters as properties of a specific selector object. This conceptualization can be defined as:

Listing 4.9– Selector interface

```
interface Selector<T, G> where G : Gene<T, G> {
    operator fun invoke(
        state: EvolutionState<T, G>, outputSize: Int
    ): EvolutionState<T, G>
}
```

In this model, the `Selector` interface gets parameterized according to the *value* and *gene* types. The incorporated `invoke` method explicitly mentions the state and output size parameters, which are essential for the selection process. Thanks to operator overloading, this framework allows direct function invocation through the selector object, depicted as `selector(...)`.[1]

Definition 4.1 (Random Selector)**.**

$$\Sigma_{\text{random}}(P : \mathbb{P},\ n : \mathbb{N}) \to \mathbb{P} \tag{4.1}$$

Each individual is then selected based on a uniform probability distribution:

$$\rho_i = \frac{1}{|P|} \tag{4.2}$$

[1] Syntactic sugar for `selector.invoke(...)`.

Although the `RandomSelector` is not frequently mentioned in literature, its straightforward nature provides a valuable touchstone for assessment.

Definition 4.2 (Roulette Wheel Selector)**.** *The **roulette wheel selector** ensures selection chances proportional to an individual's fitness. It's defined as:*

$$\Sigma_{\text{roulette}}(P : \mathbb{P},\, n : \mathbb{N},\, b : \{0,1\}) \to \mathbb{P} \tag{4.3}$$

where P represents the population, n is the individuals' count to select, and b is a boolean for preceding selection with a sort. Let ϕ_i be the fitness of the i^{th} individual. Suppose we have a function $t : \mathbb{R}^n \to \mathbb{R}^n$ that transforms the fitness into a value that assigns greater probabilities to the fittest individuals according to the ranking strategy.
The transformed fitness, meaning the fitness adjusted to the current ranking strategy, is defined as: $\Phi' = t(\Phi)$. Consider $\Phi'_{\text{min}} = \max\{\min(\Phi'_i),\, 0\}$ the minimum fitness in the population, and $\phi'_i = \Phi'_i - \max\{\min(\Phi'_i),\, 0\}$ its adjusted fitness. The selection probability p_i for the i^{th} individual is:

$$\rho_i = \frac{\phi'_i}{\sum_{j=1}^{|P|} \phi'_j} \tag{4.4}$$

Selection is based on a random number between 0 and 1, selecting i if the summed probabilities up to i surpass this number.

Remark. *The fitness adjustment, ϕ'_i, ensures non-negative probabilities. However, it makes the least fit individual unselectable, with a zero probability.*

This approach, while ensuring a selection chance for almost every individual, can face challenges. If the fitness values are too close to each other, the selection probability will be too similar, leading to a lack of diversity in the selected individuals. This issue is further discussed in section 5.4 on page 74.

Definition 4.3 (Tournament Selector)**.** *The **tournament selector** uses a tournament selection method and is defined as:*

$$\Sigma_{\text{tournament}} :\, \mathbb{P} \times \mathbb{N} \times \mathbb{N} \to \mathbb{P};\; (P,\, n,\, k) \mapsto \Sigma_{\text{tournament}}(P,\, n,\, k) \tag{4.5}$$

In this definition, P is the population, n is the number of individuals to be selected, and k is the size of each tournament. The selection process works as follows:

1. *Select a random group of k individuals from the population.*
2. *Choose the individual with the highest fitness score from this group.*
3. *Repeat steps 1 and 2 until n individuals have been selected.*

Remark. *The selection dynamic is profoundly influenced by the tournament size, k. Opting for a larger k leans towards elitism, predominantly favoring individuals with superior fitness scores. In contrast, a smaller k value promotes diversity, which helps prevent the genetic algorithm from converging too early.*

For an exhaustive delve into the intricacies and theoretical frameworks surrounding the tournament selection modality, the reader is directed to Blickle et al.'s seminal paper [12].

4.4.2 Mutation

Mutation is a crucial operator in evolutionary computation, acting as the main source of genetic variation. This operation sporadically alters parts of a solution, commonly referred to as a chromosome or individual. Such alterations ensure that the genetic material remains diverse and rich—qualities vital for producing robust and adaptable solutions. Even though mutation is often linked with genetic algorithms, its importance is felt throughout various evolutionary computation methods, from genetic programming to evolutionary strategies.

The methodology of mutation differs depending on the solution space's nature and the problem's unique demands. This section will examine various mutation strategies, highlighting their unique characteristics and applications.

From our previous discussion in section 2.2.4.2 on page 11, a mutation operator can be articulated as a function $M : \mathbb{P} \times [0, 1] \times \cdots \to \mathbb{P}$. This operator selects individuals from a population based on a probability, μ, and then alters a portion of its genetic composition.

The approach adopted by *Keen* mirrors that of selection operators. The proposed interface is as follows:

Listing 4.10– Mutator interface (Note: For clarity, certain parts of the code have been omitted.)

```
interface Mutator<T, G> where G : Gene<T, G> {
    val individualRate: Double
    val chromosomeRate: Double
    operator fun invoke(
        state: EvolutionState<T, G>, outputSize: Int
    ): EvolutionState<T, G>
    fun mutateIndividual(individual: Individual<T, G>): Individual<T, G>
    fun mutateChromosome(chromosome: Chromosome<T, G>): Chromosome<T, G>
}
```

This design contemplates mutations at different stages of the solution blueprint, focusing primarily on the chromosome level. It provides default `inkove` and `mutateIndividual` methods (omitted from the code for the sake of brevity), directing the mutation task to the `mutateChromosome` method. This design choice facilitates ease for users: they only need to implement the `mutateChromosome` method when devising a new mutator, but still retain the freedom to override other methods if necessary.

Definition 4.4 (Random Mutator)**.** *The **random mutator** serves as a rudimentary mutation operator, undertaking random alterations to the genes of a given chromosome. Formally, it is articulated as:*

$$M_{\text{random}}(P : \mathbb{P},\ \mu : [0, 1],\ \rho_{\mathbf{c}} : [0, 1],\ \rho_g : [0, 1]) \to \mathbb{P} \tag{4.6}$$

Here, P symbolizes the population, μ represents the likelihood of an individual undergoing mutation, $\rho_{\mathbf{c}}$ is the probability of a chromosome being selected for mutation, and ρ_g is the probability of a gene being selected for mutation.

Remark. *To underpin this mutator's generic nature, each gene is equipped with a `mutate` function. This function, when invoked, produces a randomly mutated gene of the same type as the original. By default, the function generates a new gene mirroring the initial gene's value. This design choice provides users with the flexibility to determine if a specific mutation operator should employ this function. As we'll observe, certain mutation operators opt not to utilize it.*

Definition 4.5 (Bit Flip Mutator)**.** *The **bit flip mutator** is a mutation operator tailored for binary genetic representations. It can be mathematically formulated as:*

$$M_{0/1}(P : \mathbb{P},\ \mu : [0, 1],\ \rho_{\mathbf{c}} : [0, 1],\ \rho_g : [0, 1]) \to \mathbb{P} \tag{4.7}$$

In this equation, P symbolizes the population, and μ is the predefined probability determining the likelihood of a given individual undergoing mutation, $\rho_{\mathbf{c}}$ is the probability of a chromosome being selected for mutation, and ρ_g is the probability of a gene being selected for mutation.

Consider the following problem: given a set of cities and the distances between them, find the shortest possible route that visits each city precisely once, subsequently returning to the starting city. This problem is known as the *Travelling Salesman Problem* (TSP), and it is a well-known combinatorial optimization problem. The TSP is NP-hard, meaning that no known algorithm can solve it in polynomial time. However, evolutionary algorithms have demonstrated their efficacy in solving this problem [5, 39].

However, the TSP's nature poses a challenge for mutation operators. The order of visiting cities is crucial, and altering the order of cities can lead to invalid solutions. For this reason, mutation operators for TSP must generate valid permutations (without repeating cities). Keen provides two mutation operators for problems of this nature: `SwapMutator` and `InversionMutator`.

Definition 4.6 (Swap Mutator)**.** *The **swap mutator**, denoted by*

$$M_{\text{swap}}(P : \mathbb{P},\ \mu : [0,\ 1],\ \rho_{\mathbf{c}} : [0,\ 1],\ \rho_g : [0,\ 1]) \to \mathbb{P} \tag{4.8}$$

is an operator that, with given probabilities, chooses a chromosome from a population and swaps the positions of two genes within it. In the equation above, P represents the population, μ denotes the probability of mutation, $\rho_{\mathbf{c}}$ quantifies the likelihood of chromosome selection, and ρ_g determines the gene selection probability.
Its versatility lies in its applicability to any chromosome, regardless of its specific genetic representation.

Definition 4.7 (Inversion Mutator)**.** *The inversion mutator works by selecting a random subset of genes from a chromosome and subsequently inverting their sequence. Formally, the operator can be represented as:*

$$M_{\text{inv}}(P : \mathbb{P},\ \mu : [0,\ 1],\ \rho_{\mathbf{c}} : [0,\ 1],\ \rho_{ib} : [0,\ 1]) \to \mathbb{P} \tag{4.9}$$

Where:

- *P represents the population.*
- *μ denotes the mutation probability.*
- *$\rho_{\mathbf{c}}$ is the chromosome-wise mutation rate.*
- *ρ_{ib} is the inversion boundary probability. This probability is used to determine the starting and ending index of the inversion.*

4.4.3 Crossover

Crossover, also termed recombination, serves as an indispensable operator in a plethora of evolutionary computation algorithms. Its primary function is to generate novel candidate solutions by amalgamating attributes from multiple input solutions. This operation, inspired by genetic mechanisms observed in biological reproduction, is pivotal for striking a balance between exploration and exploitation within an algorithm's search space. By fostering a diverse set of solution structures and retaining beneficial traits, crossover steers the evolutionary process closer to optimal or near-optimal solutions.

Within the *Keen* framework, a diverse array of crossover techniques has been conceptualized and developed, each epitomizing a distinctive recombination approach. This section endeavors to furnish an exhaustive overview of these methodologies, elucidating their mechanics, applications, and subtle intricacies.

4.4.3.1 Generalized Crossover Algorithm

A novel feature of *Keen* is its innovative approach to crossover. Diverging from conventional frameworks that typically restrict crossover to two-parent models, *Keen* provides users the flexibility to engage multiple parents in the crossover process. This multi-parent crossover approach aligns with contemporary research findings [13, 28, 37], which advocate for the benefits of such strategies.

To facilitate this advanced methodology, *Keen* introduces a *generalized crossover* algorithm. This algorithm forms the basis for multi-parent crossover operations, offering a template for users to develop custom crossover strategies.

Listing 4.11– Generalized crossover algorithm in *Keen*, serving as a foundation for multi-parent crossover.

```
val parents = Random.subsets(population, numParents, exclusivity)
val recombined = []
while (|recombined| < outputSize) {
    recombined += crossover(parents.random())
}
```

This algorithm initiates by selecting parent groups from the population, governed by the `numParents` parameter. The `exclusivity` parameter dictates whether parents can be chosen multiple times. Subsequently, these selected parents are recombined to generate offspring, until the desired number of offspring (`outputSize`) is reached. This algorithm is deliberately designed to be agnostic to the number of parents, thus allowing for versatile implementation of various multi-parent crossover methods.

The subset selection algorithm, an integral part of the generalized crossover process, is detailed in listing 4.12.

Listing 4.12– Subset selection algorithm utilized in the generalized crossover process.

```
fun <T> Random.subsets(
    elements: List<T>,
    size: Int,
    exclusive: Boolean,
): List<List<T>> {
    val subsets = []
    val remainingElements = elements.shuffled
    while (remainingElements is not empty) {
        val subset = if (exclusive) {
            remainingElements.take(size)
        } else {
            List(size) { it ->
                if (it == 0) remainingElements.removeFirst()
                else remainingElements.random().also {
                    element -> remainingElements.remove(element)
                }
            }
        }
        subsets += subset
        remainingElements = remainingElements.drop(size)
    }
    return subsets
}
```

In this algorithm, the elements are first shuffled to ensure randomness. Subsets of elements are then formed based on the specified size. If `exclusive` is `true`, unique elements are chosen for each subset. Otherwise, the first element of each subset is unique, while the rest are randomly selected and removed from the remaining pool to avoid repetition. This approach guarantees that every element is included in at least one subset, thereby ensuring comprehensive coverage and variety in the crossover process.

4.4.3.2 Selection Strategies

Definition 4.8 (n-Combine Crossover)**.** *The n-combine crossover operator is defined as a function that accepts n inputs and applies a designated combiner function to these inputs, yielding a single combined offspring. Formally, the combine crossover can be expressed as:*

$$X_{n\text{-}comb}(P : \mathbb{P}_T,\ f : T^n \to T,\ \rho_i : [0,\ 1],\ \rho_{\mathbf{c}} : [0,\ 1],\ \rho_g : [0,\ 1],\ e : \{0,\ 1\}) \to \mathbb{P}_T \tag{4.10}$$

where:

- P *represents the population of individuals of type* T*.*
- f *is the combiner function, taking* n *inputs to produce an output.*
- ρ_i *is the probability of applying the crossover to an individual in the population.*
- $\rho_{\mathbf{c}}$ *is the probability of the combiner function affecting a chromosome.*
- ρ_g *is the gene-level application probability of the function.*
- e *is a boolean value that determines whether the same individual can be used more than once as a parent.*

Remark. *The core principle of the combine crossover lies in the multi-parent crossover concept, wherein the arity of the combiner function aligns with the number of input parents.*

Notably, the *Combine Crossover* operator in *Keen* is designed as an **open class**. This design choice allows for extensive customization, enabling developers to extend and tailor the operator according to specific needs. An illustrative example is the `AverageCrossover` operator, which calculates the arithmetic mean of the parent inputs, demonstrating the versatility and adaptability of the *Combine Crossover* in various genetic algorithm scenarios.

Definition 4.9 (Single-Point Crossover)**.** *The single-point crossover operator takes a pair of input chromosomes and randomly selects a crossover point on them. It then exchanges the subsequences of genes situated after this point between the two input chromosomes to produce two outputs. More formally, a single-point crossover is represented as:*

$$X_{\text{sp}}(P : \mathbb{P},\ \rho_i : [0,\ 1],\ \rho_{\mathbf{c}} : [0,\ 1],\ e : \{0,\ 1\}) \to \mathbb{P} \tag{4.11}$$

where:

- P *represents a population of binary chromosomes.*
- ρ_i *symbolizes the likelihood of applying the crossover to an individual.*
- $\rho_{\mathbf{c}}$ *symbolizes the likelihood of applying the crossover to a chromosome.*
- e *is a boolean value that determines whether the same individual can be used as both parents.*

Remark. *The simplicity of single-point crossover makes it a popular choice in many EC applications. However, it might not always ensure adequate exploration of the search space, especially in problems where gene positions have strong interactions. In these scenarios, permutation crossover methods like ordered crossover and partially mapped crossover might be more effective.*

For better clarity on the single-point crossover operation, see the graphical representation in fig. 2.1 on page 11, which showcases the random selection of a crossover point and the subsequent exchange of gene subsequences.

Note: There exists a generalized version of single-point crossover named *multi-point crossover*. This method selects multiple crossover points on the input chromosomes and exchanges the subsequences of genes situated after these points between the two input chromosomes to produce two outputs. However, this method is currently not supported in *Keen*.

Definition 4.10 (Ordered Crossover)**.** *The ordered crossover operator takes two parent chromosomes. Through a series of steps, it ensures that the offspring chromosomes inherit the order of sequences or genes from both parents. Formally, the ordered crossover is represented as:*

$$X_{\text{ox}}(P : \mathbb{P},\ \rho_{\mathbf{i}} : [0,\ 1],\ \rho_{\mathbf{c}} : [0,\ 1],\ e : \{0,\ 1\}) \to \mathbb{P} \tag{4.12}$$

where:

- P *denotes a population of ordered chromosomes.*
- $\rho_{\mathbf{i}}$ *is the probability of applying the crossover to an individual.*
- $\rho_{\mathbf{c}}$ *represents the chance of employing the ordered crossover on a chromosome.*
- e *is a boolean value that determines whether the same individual can be used more than once as a parent.*

There are multiple implementations of the ordered crossover operator. The following displays the variant used in the *Keen* framework:

Listing 4.13– OX algorithm as implemented in the *Keen* framework. For simplicity, we represent chromosomes as arrays, meaning that `output1[i] = input1[i]` signifies that the `i`-th gene in `output1` will be assigned the value of the `i`-th gene in `input1`. `input1` and `input2` are the parent chromosomes, while `output1` and `output2` are the offspring chromosomes. All arrays are assumed to be of the same size.

```
val (index1, index2) = random.indices in input1
val crossSection = input1[index1..index2]
for (i in 0..input1.size) {
    if (i < index1 or i >= index2) {
        if (input2[i] not in crossSection) {
```

```
            output1[i] = input2[i]
        }
        if (input1[i] not in crossSection) {
            output2[i] = input1[i]
        }
    }
}
return output1, output2
```

In this approach, a random section from the first parent chromosome is copied to an offspring chromosome in the same position. Subsequently, genes from the second parent chromosome, not in the copied section, fill the offspring in the same order. The second offspring chromosome uses the second parent chromosome for the copied section and the first parent for the remaining genes. For a visual elucidation of this process, refer to fig. 4.1.

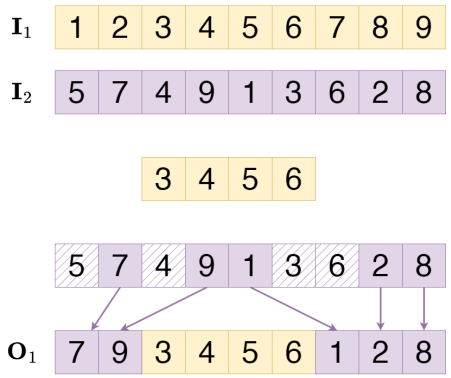


Figure 4.1: Ordered Crossover (OX) operator.

Definition 4.11 (Partially Mapped Crossover)**.** *The partially mapped crossover (PMX) operates by choosing a random substring of one input chromosome and then mapping the position of these genes on the other input chromosome. This procedure ensures output obtain genes from both inputs, maintaining the original order. Formally, PMX can be articulated as:*

$$X_{\text{pmx}}(P : \mathbb{P},\ \rho_{\mathbf{i}} : [0,\ 1],\ \rho_{\mathbf{c}} : [0,\ 1],\ e : \{0,\ 1\}) \to \mathbb{P} \tag{4.13}$$

where:

- P *denotes a population of sequenced chromosomes.*
- $\rho_{\mathbf{i}}$ *stands for the likelihood of the crossover being applied to an individual.*
- $\rho_{\mathbf{c}}$ *signifies the probability of triggering the PMX operation on a chromosome.*
- e *is a boolean value that determines whether the same individual can be used more than once as a parent.*

The implementation of PMX in the *Keen* framework operates as follows:

Listing 4.14– PMX algorithm as described in the *Keen* framework.

```
val (output1, output2) = input1, input2
val (lo, hi) = random.indices in input1
val crossSection1 = input1[lo..hi]
val crossSection2 = input2[lo..hi]
for (i in 0..input1.size) {
    if (i < lo or i >= hi) {
        while (output1[i] in crossSection2) {
```

```
            val index = crossSection2.indexOf(output1[i])
            output1[i] = crossSection1[index]
        }
        while (output2[i] in crossSection1) {
            val index = crossSection1.indexOf(output2[i])
            output2[i] = crossSection2[index]
        }
    }
}
return output1, output2
```

The essence of the above implementation is to select a random section from the first input chromosome and copy it to the output chromosome in the same position. Then, the genes from the second input chromosome, not in the copied section, fill the output in the same position as the first input. Finally, the remaining genes are mapped from the second input chromosome to the output chromosome, ensuring that the output chromosome is a valid permutation.

For a clearer visual representation of this entire process, refer to fig. 4.2.

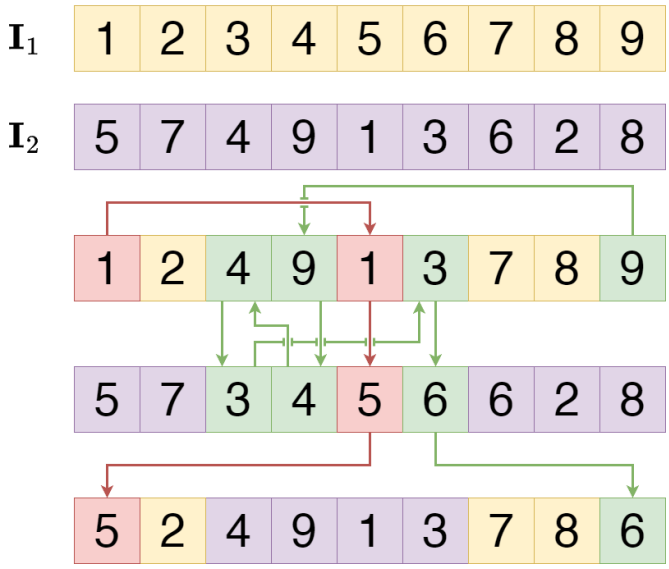


Figure 4.2: Partially Mapped Crossover (PMX) operator.

Definition 4.12 (Position Based Crossover)**.** *The position based crossover operator works with two input chromosomes. It strategically copies certain positions from one input, while the rest of the genes are filled in from the other input without duplicating any genes. Formally, the position based crossover is illustrated as:*

$$X_{\text{pbx}}(P : \mathbb{P},\ \rho_i : [0,\ 1],\ \rho_c : [0,\ 1]) \to \mathbb{P} \tag{4.14}$$

where:

- P *signifies a population of ordered chromosomes.*
- ρ_i *denotes the probability of utilizing the crossover for an individual.*
- ρ_c *signifies the likelihood of adopting the position based crossover for a chromosome.*

The way PBX is implemented in the *Keen* framework is as follows:

Listing 4.15– PBX algorithm as executed in the *Keen* framework. For simplicity, we represent chromosomes as arrays, meaning that `output1[i] = input1[i]` signifies that the `i`-th gene in `output1` will be assigned the value of the `i`-th gene in `input1`. `input1` and `input2` are the parent chromosomes, while `output1` and `output2` are the offspring chromosomes. All arrays are assumed to be of the same size.

```
val positions = random.indices in input1
val output = input1
for (i in positions) {
    output[i] = input1[i]
}
var j = 0
for (i in 0..input2.size) {
    if (input2[i] not in output) {
        output[positions[j++]] = input2[i]
        if (j >= positions.size) {
            break
        }
    }
}
return output
```

In the PBX method, select positions from the first input chromosome are directly copied to the output. The remaining genes from the second input chromosome are then filled in the order they appear, skipping any that are already present in the output. A visual representation of this process can be seen in fig. 4.3

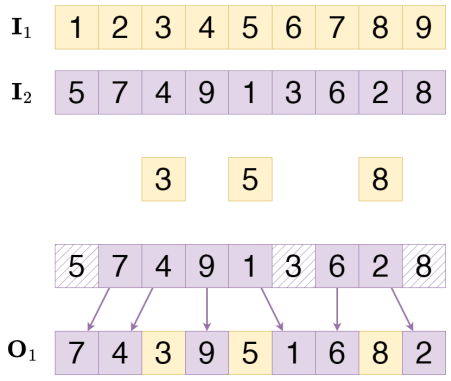


Figure 4.3: Position Based Crossover (PBX) operation.

4.5 Genetic Programming

The basic concepts of Genetic Programming (GP) are detailed in section 2.3 on page 14, but its implementation in the *Keen* framework provides a new perspective. This section highlights the initial implementation of GP in Keen, focusing on its early development stage. As Keen evolves, GP is expected to become a key feature, enhancing the connection between theoretical principles and practical advancements. Despite being in the early phases, Keen's GP implementation already demonstrates the framework's flexibility and commitment to offering a complete set of tools for evolutionary computation.

4.5.1 Primitive Set

The quintessential component of any GP implementation is the set of *primitives* that define the language of the GP system. The *Keen* framework's GP implementation is no exception, and its primitives are defined in the `prog` package.

The `prog` package exposes three main components: ***reducible*** expressions, ***program*** trees, and ***generator*** functions.

4.5.1.1 Reducible Expressions

Within the *Keen* framework, "reducible expressions" serve as the building blocks for creating more complex mathematical or logical constructs. Primitives in *Keen* are captured using the `Reducible` interface. This interface outlines a single method, `invoke`, which accepts a list of arguments and yields a single value. Drawing parallels, the `invoke` method resembles a function call, and the `Reducible` interface stands as a function signature.

The architecture of the proposed `Reducible` interface can be appreciated in listing 4.16.

Listing 4.16– The `Reducible` interface

```
interface Reducible<T> : Node<Reducible<T>> {
    operator fun invoke(environment: Environment<T>, args: List<T>): T
}
```

Remark. *A crucial feature of the `Reducible` interface in Keen is its handling of the **environment**, a structure that holds variable values for expressions. Initially, Keen used a global environment for storing variable values, but this method lacked the desired flexibility. The framework has since shifted to using a local environment for each expression, enabling variable scopes to be defined differently for each case. This adjustment enhances the capability to construct more complex expressions and solve intricate problems, aligning more closely with how variables operate in programming languages, where they are confined to their defined scopes.*
It's important to note that this modification is still under development, and as such, the current variable implementation in Keen is not yet fully operational. Addressing and refining this aspect of the framework is a priority for upcoming versions.

The `Reducible` interface extends `Node<Reducible<T>>`, meaning every reducible expression also acts as a node in an expression tree. This setup is a key part of *Keen*'s GP method, as it allows for the creation of expressions with various levels of complexity.

Moving ahead, *Keen* divides reducible expressions into two basic categories: ***terminals*** and ***functions***. Terminals are reducible expressions that function as leaf nodes in an expression tree, while functions represent reducible expressions that correspond to intermediate nodes in the tree.

Listing 4.17– Prototype of the `Terminal` interface

```
interface Terminal<T> : Reducible<T>, Leaf<Reducible<T>> {
    fun create(): Terminal<T>
}
```

Listing 4.18– Prototype of the `Fun` open class

```
open class Fun<T>(
    val name: String, override val arity: Int, val body: (List<T>) -> T
) : Reducible<T>, Intermediate<Reducible<T>> {
    override fun invoke(environment: Environment<T>, args: List<T>): T { /* ... */ }
}
```

Note that both the `Terminal` interface and the `Fun` class are parameterized by a type `T`. This type represents the return type of the `invoke` method, and is used to enforce type safety when creating expressions.

The `Fun` class is introduced to associate a function with a name and an arity. This is important as the genetic operators tailored for GP require this information to function correctly. An important part of our design approach is using an open class for the `Fun` class, this allows for the creation of functions with a generic nature via instantiation, while also allowing for the creation of more specific functions via inheritance, as shown in listing C.3 on page 120.

4.5.1.2 Program trees

The `Reducible` interface and its subsets–terminals and functions–are foundational components that aid in the construction of program trees in *Keen*. The framework represents these program trees as immutable tree structures, a design decision rooted in advantages such as thread safety, predictability, and reduced chances of accidental data modification, in particular, making program trees immutable one can assure that the arity of a node is always correct.

In essence, a program tree is characterized by a reducible expression situated at its root, which in turn references other programs as its descendants. This design is portrayed in the blueprint provided in listing 4.19.

Listing 4.19– Prototype of the `Program` class

```
class Program<V>(
    override val value: Reducible<V>,
    override val children: List<Program<V>>,
) : Tree<Reducible<V>, Program<V>> {
    /* ... */
    operator fun invoke(environment: Environment<V>, vararg args: V): V =
        value(environment, children.map { it(environment, *args) })
}
```

The `Program` class employs a type `V` as a parameter, symbolizing the return type of the `invoke` method intrinsic to the `Reducible` interface. Such an approach ensures the types of the are consistent throughout the program tree. Notably, the `invoke` function adopts a recursive mechanism. It calls upon the `invoke` method of the root reducible expression, subsequently passing the results derived from the recursive invocations of its children.

Forming immutable trees, particularly in recursive architectures, can be a challenging endeavor. To counteract this complexity, the `Tree` interface introduces a method to instantiate trees by conducting a top down tree construction. The underlying algorithm, designed to transform a list of nodes into trees, is presented in listing 4.20.

Listing 4.20– Algorithm to create a tree from a list of nodes

```
val stack = []
nodes.reversed.forEach { node ->
    val children = stack.take(node.arity)
    stack.removeAll(children)
    val node = createNode(node.value, children)
    stack += node
}
return stack.first
```

This algorithm takes a list of nodes and creates a tree by traversing the list in reverse order. For each node, it takes the required number of children from the top of the stack, removes them from the stack, and creates a new node with the given value and children. Finally, it returns the top of the stack, which is the root of the tree.

4.5.1.3 Generation Methods

Every individual within the population should consistently be a valid program throughout the evolution process. To guarantee the syntactic correctness of all newly created or varied individuals, the GP algorithm must generate an initial population of syntactically accurate individuals. The designated algorithm for producing this initial population is termed the *generation method* or *generator function*.

Koza's influential work [7] introduced two pivotal algorithms for generating the initial population: the *full* and *grow* methods.

The ***full*** method creates a tree with a specified maximum depth D, ensuring that all terminal nodes are located at depth D.

Contrastingly, the ***grow*** method crafts a tree with a maximum depth of D and a minimum depth of d, positioning all terminal nodes at depths ranging between d and D. This method was elaborated upon in section 2.3.2 on page 20.

It's noteworthy that the ***full*** method can be viewed as a specific instance of the ***grow*** method when $d = D$. Given this perspective, we can further conceptualize the ***grow*** method as a "depth-conditional tree generation" strategy. The detailed algorithm for this is presented in listing 4.21.

Listing 4.21– Depth-conditional tree generation

```
fun generateRecursive(
    intermediates: List<Intermediate>,
    leafs: List<Leaf>,
    depth: Int,
    maxHeight: Int,
    condition: (maxHeight: Int, depth: Int) -> Boolean,
    leafFactory: (Leaf) -> Tree,
    intermediateFactory: (Intermediate, List<Tree>) -> Tree,
) {
    val children = []
    val node = if (condition(maxHeight, depth)) {
        leafFactory(random node from leafs)
    } else {
        val intermediate = random node from intermediates
        repeat intermediate.arity times {
            children += generateRecursive(
                intermediates,
                leafs,
                depth + 1,
                maxHeight,
                condition,
                leafFactory,
                intermediateFactory
            )
        }
        intermediateFactory(intermediate, children)
    }
    return node
}
```

The `generateRecursive` function accepts a list of intermediate nodes, a list of leaf nodes, a maximum height, a current depth, and a condition function. The condition function receives the maximum height and the current depth of a node, returning a boolean value. If the condition is met (true), a leaf node is formed; if not (false), an intermediate node is created. This function is invoked recursively to form the descendants of the current node, halting the recursion when the condition is true or when the maximum height is attained. Additionally, the `generateRecursive` function takes in two other functions dedicated to the creation of both leaf and intermediate nodes, enabling its adaptability for any tree type, including program trees.

A key characteristic of this algorithm is its bottom-up approach to tree generation, starting from terminal nodes and finishing at the root node. This method allows for the creation of immutable trees, eliminating the need to make temporary trees.

The `generate` function operates by calling the `generateRecursive` function, with a maximum height randomly determined between d and D.

Using this method, the grow and full techniques can be illustrated as:

Listing 4.22– Grow and full methods

```
fun <T> generateProgramGrowing(
    terminals: List<Terminal<T>>,
```

```
    functions: List<Fun<T>>,
    min: Int,
    max: Int,
) = generateProgram(terminals, functions, min, max) { h, d ->
    d == h || d >= min
        && random.double() < terminals.size / (terminals.size + functions.size)
}

fun <T> generateProgramFull(
    terminals: List<Terminal<T>>, functions: List<Fun<T>>, min: Int, max: Int,
) = generateProgram(terminals, functions, min, max) { h, d -> h == d }

fun <T> generateProgram(
    terminals: List<Terminal<T>>,
    functions: List<Fun<T>>,
    min: Int,
    max: Int,
    condition: (Int, Int) -> Boolean,
) = Tree.generate(
    nodes = terminals to functions,
    depths = min to max,
    condition = condition,
    leafFactory = ::Program,
    intermediateFactory = ::Program
)
```

The `generateProgramGrowing` function creates a program tree using lists of terminal and intermediate nodes, along with minimum and maximum heights. It uses these criteria to decide how to build the tree:

- If the current depth is the same as the maximum height, it stops growing the tree.
- If the current depth is at least the minimum height and a random number between 0 and 1 is less than the ratio of terminal to intermediate nodes, it also stops growing.
- Otherwise, it continues growing the tree.

The `generateProgramFull` function is similar, but it only stops growing the tree when the current depth reaches the maximum height.

Koza also suggested that population members could be derived using a *ramped half-and-half* technique. Here, the initial population is formed utilizing both the *full* and *grow* strategies, with the method for each individual being selected at random. To implement this, *Keen* offers the `generateProgramWith` function, receiving a list of generation strategies, a compilation of terminals and functions, and a minimum and maximum height. This function randomly chooses a generation method to produce the program tree. By defining the function this way, integrating new generation techniques into the library becomes seamless.

In this method, the *ramped half-and-half* technique is just a specific use of the `generateProgramWith` function. This version limits the list of generation strategies to only the *full* and *grow* methods.

We would like to note that this approach is novel to the *Keen* framework, as other GP libraries usually limits the generation method to a single strategy (usually the grow ramped half-and-half method).

4.5.2 Genetic Operators

The preceding sections introduced genetic operators within the realm of genetic algorithms. Yet, when we transition to Genetic Programming (GP), these operators assume a different character. In GP, the evolutionary landscape is punctuated by program trees or structures, making the operators' responsibilities more multifaceted. They must not only bring about genetic variations but also uphold the structural, syntactic, and semantic sanctity of the programs they modify.

The upcoming subsections focus on the unique genetic operators developed for GP in the *Keen* framework, explaining their functions, how they work, and their importance.

Remark. *This operators are still in a prototypal stage, and their implementation is prone to change in future versions of the framework.*

Definition 4.13 (Subtree Crossover)**.** *The subtree crossover is a genetic operator designed for tree-structured chromosomes. For two given trees, it randomly chooses a node or subtree in each. The subtrees anchored at these points are then interchanged, birthing two new outputs. The operator can be mathematically expressed as:*

$$X_{subtree}(P : \mathbb{P}, \rho_{\mathbf{i}} : [0,\ 1], \rho_{\mathbf{c}} : [0,\ 1], \rho_{\mathbf{g}} : [0,\ 1], e : \{0,\ 1\}) \to \mathbb{P} \tag{4.15}$$

Here's a brief rundown of the parameters:

- P*: A population of program trees.*
- $\rho_{\mathbf{i}}$*: Probability of an individual undergoing crossover.*
- $\rho_{\mathbf{c}}$*: Chance of initiating the subtree crossover.*
- $\rho_{\mathbf{g}}$*: Likelihood of selecting a gene for the crossover process.*
- e*: A flag indicating if an individual can participate in crossover more than once.*

Remark. *Keen's rendition of the subtree crossover currently equips all tree nodes with equal selection probabilities. This strategy may evolve in subsequent versions to encompass a more sophisticated selection mechanism.*

An extensive explanation of this operator was provided in section 2.3.4.2 on page 26.

Definition 4.14 (Point Mutation)**.** *The point mutation operator in GP works by picking a random node in a program tree and replacing it with another node that has the same number of child nodes (arity). This swap creates a variation of the original program while keeping the overall tree structure intact.*
The operation can be mathematically represented as:

$$M_{point}(P : \mathbb{P},\ \mu_{\boldsymbol{i}} : [0,\ 1],\ \mu_{\boldsymbol{c}} : [0,\ 1],\ \mu_{\boldsymbol{g}} : [0,\ 1]) \to \mathbb{P} \tag{4.16}$$

Explaining the parameters:

- P*: The population of program trees.*
- $\mu_{\boldsymbol{i}}$*: The chance of mutating an individual tree.*
- $\mu_{\boldsymbol{c}}$*: The chance of mutating a chromosome within a tree.*
- $\mu_{\boldsymbol{g}}$*: The probability of selecting a particular gene for mutation.*

A key feature of Point Mutation in *Keen* is its ability to maintain the overall structure of program trees while introducing genetic changes.

Remark. *The focus on matching arity during node replacement is important. This method ensures the mutated tree retains the structure of the original tree, reducing the chances of runtime errors or unclear meanings that could hinder the evolutionary process.*

Definition 4.15 (Subtree Mutation)**.** *The subtree mutation operator functions by picking a node from a program tree and replacing the subtree from that node with a new one. This method maintains the overall structure of the program but adds new genetic elements, potentially improving the program's performance.*
The mathematical representation of this operation is:

$$M_{subtree} : \mathbb{P} \times [0,\,1] \times [0,\,1] \times [0,\,1] \to \mathbb{P};\ (P,\,\mu_{\boldsymbol{i}},\,\mu_{\boldsymbol{c}},\,\mu_{\boldsymbol{g}}) \mapsto M_{subtree}(P,\,\mu_{\boldsymbol{i}},\,\mu_{\boldsymbol{c}},\,\mu_{\boldsymbol{g}}) \tag{4.17}$$

Explaining the parameters:

- P*: The population of program trees.*
- $\mu_{\boldsymbol{i}}$*: The chance of a tree undergoing mutation.*
- $\mu_{\boldsymbol{c}}$*: The likelihood of choosing a chromosome for mutation.*
- $\mu_{\boldsymbol{g}}$*: The probability of selecting a gene for mutation.*

4.6 Extensibility

4.6.1 Structured Extensibility Through Interfaces

Keen employs well-defined interfaces to encapsulate essential functionalities. These interfaces act as templates, enabling developers to easily extend and modify the framework. The process is straightforward: to enhance or customize capabilities, one simply implements the provided interfaces.

4.6.2 Factory Method Pattern in Genetic Construction

The Factory Method Pattern plays a pivotal role in *Keen*, facilitating the dynamic creation of genetic material. By decoupling the framework from specific implementations, this pattern provides the flexibility to introduce varied genetic construction techniques seamlessly.

Consider the `Chromosome.Factory` interface and the `Chromosome.AbstractFactory` abstract class that extends it. These provide a standardized mechanism for generating chromosome objects, ensuring easy adaptability to diverse and complex problem domains such as crash reproduction (refer to chapter 6 on page 81)

For illustration, consider the `SimpleChromosome`:

Listing 4.23– Implementation of the *SimpleChromosome* using the `Chromosome.Factory` interface.

```
data class SimpleChromosome(override val genes: List<SimpleGene>) :
    Chromosome<Int, SimpleGene> {
    // ... Other implementations ...

    class Factory(override var size: Int, private val geneFactory: () -> SimpleGene) :
        Chromosome.AbstractFactory<Int, SimpleGene>() {
        override fun make() = SimpleChromosome((0..<size).map { geneFactory() })
    }
}
```

Using the `SimpleChromosome` is straightforward:

Listing 4.24– Utilization of the *SimpleChromosome* within an evolutionary engine setup.

```
data class SimpleGene(val dna: Int) : Gene<Int, SimpleGene> {
    // ... Implementation details for SimpleGene ...
}

fun main() {
```

```
    val engine = evolutionEngine(
        ::fitnessFn,
        genotypeOf {
            chromosomeOf {
                // Random genes between 0 and 100
                SimpleChromosome.Factory(10) { SimpleGene((0..100).random()) }
            }
        }
    ) {
        // ... Other configurations ...
    }
    val result = engine.evolve()
    // ... Use the result ...
}
```

We would like to mention that this is not a totally novel approach, as it is inspired by the work of Wilhelmstötter's *Jenetics* [77].

4.6.3 Observer Pattern for Evolution Monitoring

A key design pattern employed in *Keen* is the Observer Pattern. This pattern is fundamental for tracking the status of the evolutionary process. Its primary advantage lies in decoupling the core evolutionary engine from the monitoring components. This separation facilitates the extension and customization of monitoring functionalities without affecting the underlying evolutionary mechanisms.

As discussed in section 4.3.3 on page 49, the `EvolutionListener` interface acts as a cornerstone for this pattern. It offers a standardized approach to observe and respond to the evolutionary process. Developers can implement this interface to devise custom listeners, which can then be integrated with the evolutionary engine, enhancing the flexibility and adaptability of the monitoring process.

4.6.4 Modularity in Design

The architecture of *Keen* is inherently modular, offering developers the flexibility to craft and integrate new algorithms with ease. This modularity is exemplified in the way genetic algorithms can be customized. Developers can, for instance, design a genetic algorithm that iteratively modifies the population until certain criteria are met, utilizing the existing functions that define the genetic algorithm's stages.

This modular approach extends to the enhancement of evolutionary process notifications. For example, developers can design a custom listener with a hook to receive an alert when the population achieves a predefined fitness level. This ability to modify and extend the notification system underscores the framework's versatility in adapting to varied evolutionary scenarios.

Another, arguably more complex, usage example could be the implementation of Evolutionary Strategies. Evolutionary Strategies are similar to Genetic Algorithms, but they use a different approach to the selection process. Without going into the details of the algorithm, the main difference between the two is that in Evolutionary Strategies, the operators used to alter the population –selection, mutation and crossover– act differently on the individuals, and the existence of additional parameters that affect the selection process. Notwithstanding these differences, the core of the algorithm is the same, and the implementation of the Evolutionary Strategy is straightforward. The following code snippet shows a possible implementation of a simplified evolutionary strategy using Keen:

Listing 4.25– Implementation of an Evolutionary Strategy using Keen (assuming the existence of an evolution engine).

```
var state = EvolutionState.empty()
do {
    val initialPopulation = engine.startEvolution(state)
    val evaluatedPopulation = engine.evaluate(initialPopulation)
```

```
    // We assume this function selects individuals based on the strategy parameters
    val parents = evStratSelection(evaluatedPopulation)
    val nextPopulation = engine.alterOffspring(offspring)
    // We assume this function updates the parameters needed by the evolutionary strategy
    updateStrategyParameters()
    state = state.copy(population = nextPopulation)
} while (engine.limits.none { it -> it(state) })
```

Note that most of the code is the same as the one used in the genetic algorithm, the only difference is the selection process and the update of the parameters. This is possible thanks to the modularity of the framework, which allows the developer to reuse the existing functions and classes to create new algorithms.

4.7 *StraitJakt*

In the field of evolutionary computation, the complexity and non-deterministic nature of these processes underscore the importance of robust data validation, both before and after execution. As frameworks expand in scale, ensuring the integrity of operations becomes crucial. A common issue in genetic programming, for instance, is the inadvertent generation of invalid trees. Such errors may arise from factors like the use of improper operators or the unintended creation of cyclic dependencies.

One challenge with implementing these necessary validations is the resulting code complexity. Intensive validation checks can obscure the core logic, reducing the code's readability and maintainability. Moreover, in scenarios where multiple validations are required for a single operation, the standard practice of throwing exceptions for each failed check is inefficient. Consider the example of a chromosome with several invalid genes; a more practical approach would be to collate and return a comprehensive list of errors rather than issuing separate exceptions for each anomaly.

To address these challenges, we developed *StraitJakt*, a specialized library designed to streamline the process of constraint validation within evolutionary algorithms. While *StraitJakt* is tailored for integration with the *Keen* framework, its versatile design allows for application in other projects as well. In *Keen*, *StraitJakt* plays a pivotal role in validating the numerous constraints essential for the proper functioning of the algorithms.

StraitJakt was initially developed as part of the *Keen* framework, but as its complexity grew, it was separated into its own project to maintain a clear separation of concerns.

The library works by defining *constraint blocks*, which are collections of validation checks. Each constraint block defines a scope that enables a special syntax for the validation checks within it. The syntax is designed to be as close as possible to the natural language used to describe the constraints. The syntax defined by the constraint blocks can be seen in listing 4.26.

Listing 4.26– Constraint block syntax

```
constraints {
    "The value of x must be greater than or equal to 10" {
        // By default, the exception type is defined by the constraint
        // In this case, BeAtLeast creates an IntConstraintException due to the type of x
        x must BeAtLeast(10)
    }
    // Optionally, a custom exception constructor can be provided
    "The value of y must not be negative"(::CustomConstraintException) {
        y mustNot BeNegative
    }
}
```

Here, the `constraints` block defines a scope for the validation checks. Then, each constraint is defined by a string that describes the constraint and a lambda that contains the validation checks. The validation checks are defined using the `must` and `mustNot` keywords, which are followed by the constraint to be checked.

Constraints are defined by a `Constraint` object, which is a class that implements the `Constraint` interface. The `Constraint` interface defines two functions: `validator` and `generateException`. The `validator` function is used to check if the constraint is satisfied, and the `generateException` function is used to generate an exception if the constraint is not satisfied.

Once defined, the block will be evaluated as follows:

1. A `Jakt.Scope` is created to enable the special syntax for the validation checks.
2. For each constraint, a `Jakt.Scope.StringScope` is created to provide access to the `must` and `mustNot` keywords.
3. If a validation check fails, an exception is generated and stored in the `Jakt.Scope`.
4. Once all the constraints have been evaluated, if:
   (a) No exceptions were generated, the code continues normally.
   (b) Exceptions were generated, a `CompositeException` is thrown containing all the exceptions generated.

StraitJakt makes heavy use of operator overloading, extension functions, and trailing lambdas to provide a concise syntax for the validation checks. An implementation that does not use these features would look like the one shown in listing C.7 on page 123.

The main purpose of *StraitJakt* is to provide an expressive way of validating multiple constraints at once, even though in some cases *Keen* uses it to validate a single constraint. This is done to maintain a consistent API for all the constraints, even though some of them only require a single validation check.

A similar result could be achieved using the *Arrow* library [54], which provides an `ensure` function that allows us to validate a single constraint. However, *Arrow* introduces non-standard[2] functional programming concepts that may be unfamiliar to some users. Given that *Keen* is focused on Evolutionary Algorithms, we tried to use a more standard approach to avoid introducing unnecessary complexity. Moreover, we argue that the syntax provided is more expressive than the one provided by *Arrow*[3] since the scope of each constraint is defined by a string that describes the constraint.

Listing 4.27– Example of the use of *Arrow* to validate multiple constraints, note that the code could be improved to avoid code duplication and reduce verbosity.

```
fun validateX(): Either<IllegalArgumentException, Unit> = either {
    ensure(x >= 10) { IllegalArgumentException("
    ↪  The value of x must be greater than or equal to 10") }
}

fun validateY(): Either<CustomConstraintException, Unit> = either {
    ensure(y >= 0) { CustomConstraintException("The value of y must not be negative") }
}

val exceptions = mutableListOf<Exception>()

fold(
    { validateX() },
    { it: IllegalArgumentException -> exceptions.add(it) },
    {}
)
fold(
    { validateY() },
    { it: CustomConstraintException -> exceptions.add(it) },
    {}
)
```

[2] Non-standard in the context of the *Kotlin* language.
[3] For the case of validating multiple pre-conditions and post-conditions.

```
if (exceptions.isNotEmpty()) {
    throw CompositeException(exceptions)
}
```

An example of the use of *StraitJakt* within *Keen* can be seen in listing C.8 on page 124.

As a final note, it is important to mention that *StraitJakt* is heavily inspired by *Kotest*'s String Spec DSL and the `should` and `shouldNot` keywords [66].

Chapter 5

Case Study: Real Function Optimization

5.1 Introduction

Real function optimization is a prevalent task in numerous fields, from *data science* and *machine learning* (ML) to *operations research* and *engineering*. It presents a common class of challenges that can be effectively addressed using GAs.

This chapter embarks on a practical exploration of *Keen*. We aim to illustrate the robust capabilities of *Keen* by employing it to solve various classic optimization problems, the details of which are provided in appendix B on page 101.

Please note that this chapter doesn't intend to delve into the formulation of these optimization problems; the discussion is primarily centered around how *Keen* can be leveraged as a solution tool. For a comprehensive understanding of *Keen*, its design, and its features, refer to chapter 4 on page 45.

The ensuing sections will navigate through the problem descriptions, the solutions employed using *Keen*, the corresponding results, and the consequent analysis. In the following sections, we will systematically explore and demonstrate the efficiency and versatility of *Keen* in tackling a range of optimization problems.

5.2 Problem Description

This chapter focuses on optimizing various real functions, detailed in appendix B on page 101, chosen for their complexity and prevalence in optimization studies. Our objective is to identify the **global minimum** for these functions, addressing what is known as the global optimization challenge.

The core problem involves finding a point $x^* \in \mathbb{R}^n$ for a function $f : \mathbb{R}^n \to \mathbb{R}$, where $f(x^*)$ is the lowest possible value of f. This is termed the **global minimum** of f, and the task of locating this minimum is the *function optimization problem*.

The search space, or the realm of potential solutions, varies by function. For instance, the search space for the *Cross-in-Tray* function is $[-10,\ 10]^2$, while for the *Easom* function, it's $[-100,\ 100]^2$.

Understanding the concepts of sets, cardinalities, and functions is essential due to the diverse nature of our functions and their search spaces. These foundational concepts aid in grasping the search space's complexity and are crucial for employing the *Keen* framework in these optimization tasks.

The following definitions, theorems, and corollaries will elucidate the necessary background.

Definition 5.1 (Set cardinality inequality)**.** *For any two sets A and B, if there exists an injective function $f : A \to B$, then $|A| \leq |B|$.*

Theorem 5.1 (Schröder–Bernstein theorem)**.** *Given two sets A and B, if there exist two injective functions $f : A \to B$ and $g : B \to A$, then there exists a bijective function $h : A \to B$.*

Corollary 5.1.1. *For any two sets A and B, if $|A| \leq |B|$ and $|B| \leq |A|$, then $|A| = |B|$.*

Proof. The conditions $|A| \leq |B|$ and $|B| \leq |A|$ imply that there exist injective functions $f : A \to B$ and $g : B \to A$, respectively. By the Schröder–Bernstein Theorem (theorem 5.1 on the preceding page), the existence of these injective functions guarantees a bijective function $h : A \to B$.
A bijective function is one that is both injective and surjective. This means that every element of A is mapped to a unique element in B and every element of B is the image of some element in A.
Therefore, there can't be more elements in A than in B (or vice versa), as this would contradict the surjectivity or the injectivity of the function h. Thus, by the definition of set cardinality (definition 5.1 on the previous page), we conclude that $|A| = |B|$. □

Theorem 5.2. *Let $(x, y) \in \mathbb{R}^2$ with $x \neq y$. Then, the cardinality of the interval $[x, y]$, denoted as $|[x, y]|$, is the same as the cardinality of $\mathbb{R}$. Thus, $|[x, y]|$ is uncountable.*

Proof. Let us define a function $f : [x, y] \to \mathbb{R}$ defined as $f(z) = \tan\left(\frac{\pi(z-x)}{y-x} - \frac{\pi}{2}\right)$ which maps the interval $[x, y]$ bijectively onto $\mathbb{R}$.
This means that there is an injective function $f : [x, y] \to \mathbb{R}$ and an injective function $g : \mathbb{R} \to [x, y]$.
By the definition of cardinality inequality (definition 5.1 on the preceding page), we conclude that $|[x, y]| \leq |\mathbb{R}|$ and $|\mathbb{R}| \leq |[x, y]|$.
By theorem 5.1 on the previous page, we conclude that $|[x, y]| = |\mathbb{R}|$. □

These results make it clear that, in most cases, the search space for our real function optimization problems will have an uncountable number of potential solutions. This vastness and complexity underline the necessity for robust and efficient optimization methods like those provided by *Keen*.

5.3 Solution

To solve this problem, we aim to optimize certain functions using a GA. This approach will leverage a double-valued chromosome where each gene represents the x_i values. The fitness function encapsulates the function we aim to optimize. Let's break down the process:

5.3.1 Chromosome Structure

Each gene in the chromosome will be a double-valued gene, with a specified range of valid values depending on the optimization function. For the Bukin function N.6 (appendix B.4 on page 102), valid values range between $[-15, -5]$ for the first gene, and $[-3, 3]$ for the second.

5.3.2 Variation Operators

Regardless of the specific function we're optimizing, we will consistently employ the following variation operators:

- Recombination: *average crossover* –a specialized form of the n-combine crossover operators that computes a gene-wise average of n parents to produce a single offspring. In this case, we don't require more than two parents, due to the representation of the chromosome, but alternative implementations may take advantage of utilizing more than two parents.
- Mutation: *random mutation* –for each gene, a random real number is generated within the valid range for that gene.

For selection, we compare the performance of *random selector*, *tournament selector*, and *roulette selector* to showcase both the impact of selection on the optimization process, as the overall performance and consistency of *Keen*'s optimization capabilities.

5.3.3 Implementation of the Optimization Process

To demonstrate the implementation within the *Keen* framework, we utilize the *Bukin function N.6* as an example. *The Bukin function N.6* is defined and implemented in the context of *Keen* as follows:

Listing 5.1– Implementation of the *Bukin function N.6* in Kotlin for the *Keen* framework.

```
fun bukinN6(genotype: Genotype<Double, DoubleGene>) = genotype.flatten().let { (x, y) ->
    100 * (y - 0.01 * x.pow(2) + 1).pow(2) + 0.01 * (x + 10).pow(2)
}
```

In this implementation, the Bukin function takes a genotype as an input, which is then transformed into a pair of x and y values. The fitness value is calculated using the defined *Bukin function N.6* formula.

To streamline the creation of the evolution engine across various optimization functions, we abstract its definition:

Listing 5.2– Kotlin function for creating an evolution engine tailored to a specific optimization function in *Keen*.

```
fun createEngine(
    function: (Genotype<Double, DoubleGene>) -> Double,
    selectOp: Selector<Double, DoubleGene>,
    summary: EvolutionSummary<Double, DoubleGene>,
    vararg ranges: ClosedRange<Double>
) = evolutionEngine(function, genotypeOf {
    chromosomeOf {
        doubles {
            this.ranges += ranges
            this.size = 2
        }
    }
}) {
    ranker = FitnessMinRanker()
    populationSize = 500
    parentSelector = selectOp
    survivorSelector = selectOp
    alterers += listOf(
        RandomMutator(individualRate = 0.1),
        AverageCrossover(chromosomeRate = 0.3)
    )
    listeners += summary
    limits += listOf(
        SteadyGenerations(generations = 50),
        GenerationLimit(generations = 500)
    )
}
```

The above code snippet outlines a function for initializing an evolution engine. This engine is configured to optimize a given fitness function, employing a genotype composed of two double-valued genes, as specified by the provided value ranges.

The key configurations of the engine include:

- Setting the population size to 500 individuals.
- Employing a *Fitness Minimum* ranker to prioritize optimal fitness values.
- Utilizing alterers such as *Random Mutator* (mutation rate: 0.1) and *Average Crossover* (crossover rate: 0.3), chosen based on experimental tuning.
- Implementing a *Steady Generations* limit, halting the evolution process after 50 consecutive generations without improvement.
- Applying a *Generation Limit* of 500 to prevent infinite loops in the evolutionary cycle.

With the evolution engine established, the next step involves defining a function to execute the GA for the intended optimization function:

Listing 5.3– Kotlin function for running the GA using the Bukin function N.6 in the *Keen* framework.

```
fun main() {
    val summary = EvolutionSummary<Double, DoubleGene>(Duration::inWholeMicroseconds)
    val engine = createEngine(
        ::bukinN6,
        TournamentSelector(),
        summary,
        -15.0..-5.0,
        -3.0..3.0
    )
    engine.evolve()
    summary.display()
}
```

This function sets up the evolution engine for the Bukin function N.6. Upon completion of the GA process, it displays the results, with the summary including elapsed time measurements in microseconds. This precision is particularly relevant as some durations may be as short as 10^{-3} milliseconds.

If reproducibility is desired, the following line can be added to the beginning of the `main` function:

Listing 5.4– Setting the seed for the random number generator in *Keen*.

```
Domain.random = Random(420)
```

A key challenge with our problem and solution is that they aren't readily addressed by the frameworks discussed in this thesis. The issue stems from our need for unique ranges for each gene, a feature not directly available in these frameworks. While it's possible to adapt other frameworks to meet this requirement, doing so would demand considerable customization to achieve the desired functionality.

For example, implementing the solution in *Jenetics* would require the creation of a custom class to simulate the behavior of double-valued genes with different ranges. Additionally, we'd need to define classes for the various optimization functions, potentially utilizing *functional interfaces*. The core of the solution involves creating a class that inherits from the `Problem` interface, necessitating the definition of `fitness` and `codec` methods. The `fitness` method calculates the fitness for a genotype, while the `codec` method generates a `Codec` using the custom class, facilitating the creation of `Genotype` instances. This `Codec` is essential for configuring the GA `Engine` to execute the genetic algorithm.

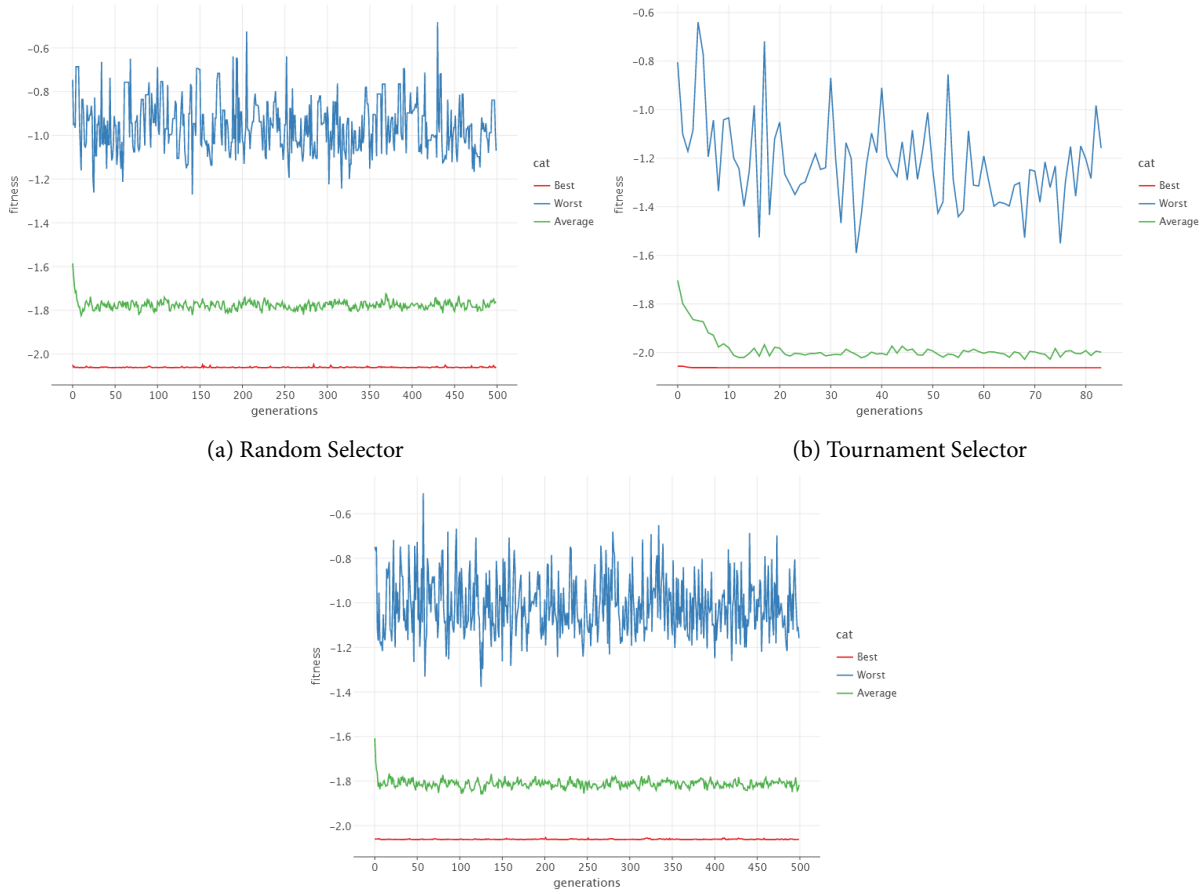
For *DEAP*, adapting the solution might be somewhat simpler. A custom crossover function would be needed for the average crossover, and another for the random mutation, but the existing *gaussian mutation* operator could be suitable for our problem, perhaps with minor adjustments. This reduces the complexity compared to *Jenetics*, but still demands a thorough understanding of *DEAP* and some customization.

Adapting both examples to the frameworks would necessitate thorough expertise in their use and considerable effort. Additionally, the available documentation for these frameworks may not be comprehensive enough, potentially leading to a need for extensive experimentation to achieve the desired implementation.

5.4 Results and Discussion

This section elucidates the optimization outcomes for 20 classical functions, analyzed using three distinct selection strategies: Random, Tournament, and Roulette. The following tables and figures provide an in-depth view of the time efficiency and effectiveness of each phase in the evolutionary process, along with the resultant optimization performance.

The comparative analysis of the selectors, as depicted in fig. 5.1 on the next page, highlights the superior performance of the



(a) Random Selector

(b) Tournament Selector

(c) Roulette Selector

Figure 5.1: Fitness evolution for the Cross-in-tray function under different selection strategies.

Tournament Selector over the Random and Roulette Selectors. Notably, the Random and Roulette selectors exhibit comparable outcomes, which can be attributed to the inherent nature of the Roulette Selector. This selector functions as a probabilistic version of the Random Selector, where the selection probability is guided by individual fitness weights. As the evolutionary process progresses, the variance in these weights diminishes, leading to Roulette Selector's behavior increasingly resembling that of the Random Selector.

It is crucial to observe that while both Random and Roulette selectors occasionally approach or even achieve the global optimum, their inherent randomness impedes consistent maintenance of the optimum solution. Conversely, the Tournament Selector, with its more deterministic and competitive selection mechanism, not only reaches but also reliably sustains the optimal solutions once discovered. From this we can conclude that a stagnation termination criterion is not suitable for the Random and Roulette selectors, as they are not able to maintain the optimal solution for a long enough period of time, instead a target fitness criterion may be more appropriate.

Now we will present the results of the optimization process of each selector for each function. Each experiment was run 4 times, dropping the first iteration to avoid JVM warmup effects. The results presented are the average of the remaining 3 iterations.

5.4.1 Random Selector

Table 5.1: Time spent in each phase of the evolution process with a Random Selector.

Function	Selection (ms)	Total (ms)
Ackley	2.33×10^{-2}	6.68×10^{2}
Beale	4.10×10^{-3}	5.83×10^{2}
Booth	4.02×10^{-3}	5.79×10^{2}
Bukin N.6	4.54×10^{-3}	5.75×10^{2}
Cross-in-tray	3.94×10^{-3}	5.89×10^{2}
Easom	3.97×10^{-3}	5.68×10^{2}
Eggholder	3.96×10^{-3}	5.73×10^{2}
Goldstein-Price	3.96×10^{-3}	5.77×10^{2}
Himmelblau	4.04×10^{-3}	5.94×10^{2}
Holder Table	3.96×10^{-3}	5.87×10^{2}
Levi	4.21×10^{-3}	6.54×10^{2}
Matyas	4.67×10^{-3}	6.99×10^{2}
McCormick	4.41×10^{-3}	6.41×10^{2}
Rastrigin	4.17×10^{-3}	6.20×10^{2}
Rosenbrock	4.10×10^{-3}	5.87×10^{2}
Schaffer N.2	3.96×10^{-3}	5.63×10^{2}
Schaffer N.4	3.96×10^{-3}	5.69×10^{2}
Sphere	4.09×10^{-3}	6.15×10^{2}
Styblinski-Tang	5.10×10^{-3}	7.34×10^{2}
Three-Hump Camel	4.18×10^{-3}	7.68×10^{2}
Average	5.13×10^{-3}	6.17×10^{2}
S. Deviation	4.18×10^{-3}	5.76×10^{1}

Table 5.2: Results of the optimization process with a Random Selector.

Function	Generations	Fittest X	Fittest Y	Error
Ackley	5.00×10^{2}	-1.91×10^{-2}	-1.62×10^{-2}	1.37×10^{-1}
Beale	5.00×10^{2}	1.18×10^{0}	7.01×10^{-1}	7.15×10^{-1}
Booth	5.00×10^{2}	4.25×10^{-1}	3.56×10^{0}	1.16×10^{0}
Bukin N.6	5.00×10^{2}	-9.85×10^{0}	-2.74×10^{-2}	3.87×10^{-3}
Cross-in-tray	5.00×10^{2}	1.27×10^{0}	-1.32×10^{0}	2.69×10^{-3}
Easom	5.00×10^{2}	3.24×10^{0}	2.90×10^{0}	5.39×10^{-1}
Eggholder	5.00×10^{2}	3.21×10^{1}	1.47×10^{2}	5.74×10^{2}
Goldstein-Price	5.00×10^{2}	-5.80×10^{-3}	-8.87×10^{-1}	9.36×10^{0}
Himmelblau	5.00×10^{2}	3.01×10^{0}	6.05×10^{-1}	1.27×10^{1}
Holder Table	5.00×10^{2}	7.53×10^{-2}	-6.00×10^{0}	1.58×10^{1}
Levi	5.00×10^{2}	1.11×10^{0}	1.15×10^{0}	1.42×10^{-1}
Matyas	5.00×10^{2}	-5.46×10^{-2}	-1.13×10^{-1}	2.37×10^{-3}
McCormick	5.00×10^{2}	3.11×10^{-2}	-1.09×10^{0}	7.68×10^{-1}
Rastrigin	5.00×10^{2}	4.19×10^{-2}	-1.32×10^{0}	1.06×10^{1}
Rosenbrock	5.00×10^{2}	6.12×10^{-1}	4.23×10^{-1}	2.27×10^{-1}
Schaffer N.2	5.00×10^{2}	5.18×10^{-1}	1.32×10^{0}	8.40×10^{0}
Schaffer N.4	5.00×10^{2}	1.46×10^{0}	6.48×10^{-1}	2.13×10^{-2}
Sphere	5.00×10^{2}	-1.49×10^{-2}	4.34×10^{-2}	3.66×10^{-3}
Styblinski-Tang	5.00×10^{2}	-1.12×10^{0}	-4.18×10^{-1}	1.78×10^{1}
Three-Hump Camel	5.00×10^{2}	4.07×10^{-2}	-1.64×10^{-4}	3.81×10^{-3}
Average	5.00×10^{2}	—	—	3.26×10^{1}

Continued on next page

Table 5.2 continued from previous page

Function	Generations	Fittest X	Fittest Y	Error
S. Deviation	0.00×10^{0}	—	—	1.24×10^{2}

5.4.2 Tournament Selector

Table 5.3: Time spent in each phase of the evolution process with a *Tournament Selector*.

Function	Selection (ms)	Total (ms)
Ackley	2.31×10^{-1}	1.80×10^{2}
Beale	3.02×10^{-2}	1.89×10^{2}
Booth	3.76×10^{-2}	2.15×10^{2}
Bukin N.6	3.02×10^{-2}	1.98×10^{2}
Cross-in-tray	3.21×10^{-2}	1.56×10^{2}
Easom	3.04×10^{-2}	1.57×10^{2}
Eggholder	3.09×10^{-2}	1.99×10^{2}
Goldstein-Price	3.12×10^{-2}	1.69×10^{2}
Himmelblau	3.13×10^{-2}	1.58×10^{2}
Holder Table	3.01×10^{-2}	2.03×10^{2}
Levi	5.91×10^{-2}	4.50×10^{2}
Matyas	4.73×10^{-2}	6.81×10^{2}
McCormick	3.25×10^{-2}	1.63×10^{2}
Rastrigin	3.24×10^{-2}	2.51×10^{2}
Rosenbrock	3.03×10^{-2}	1.78×10^{2}
Schaffer N.2	3.00×10^{-2}	1.44×10^{2}
Schaffer N.4	3.03×10^{-2}	1.68×10^{2}
Sphere	3.28×10^{-2}	4.72×10^{2}
Styblinski-Tang	6.09×10^{-2}	1.57×10^{2}
Three-Hump Camel	3.25×10^{-2}	4.71×10^{2}
Average	4.52×10^{-2}	2.48×10^{2}
S. Deviation	4.37×10^{-2}	1.43×10^{2}

Table 5.4: Results of the optimization process with a *Tournament Selector*.

Function	Generations	Fittest X	Fittest Y	Error
Ackley	1.09×10^{2}	-3.79×10^{-7}	-2.10×10^{-7}	1.22×10^{-6}
Beale	1.10×10^{2}	2.96×10^{0}	4.90×10^{-1}	7.21×10^{-4}
Booth	1.12×10^{2}	1.00×10^{0}	3.00×10^{0}	0.00×10^{0}
Bukin N.6	1.19×10^{2}	-1.00×10^{1}	6.17×10^{-5}	1.11×10^{-8}
Cross-in-tray	1.22×10^{2}	1.34×10^{0}	4.49×10^{-1}	1.87×10^{-6}
Easom	9.42×10^{1}	3.14×10^{0}	3.14×10^{0}	1.29×10^{-8}
Eggholder	1.23×10^{2}	7.64×10^{1}	4.61×10^{2}	6.89×10^{1}
Goldstein-Price	9.01×10^{1}	-6.10×10^{-11}	-1.00×10^{0}	7.68×10^{-14}
Himmelblau	9.96×10^{1}	2.99×10^{0}	1.99×10^{0}	2.22×10^{-7}
Holder Table	1.38×10^{2}	2.68×10^{0}	3.22×10^{0}	2.02×10^{-5}
Levi	1.25×10^{2}	1.00×10^{0}	1.00×10^{0}	1.45×10^{-7}
Matyas	3.49×10^{2}	-6.16×10^{-10}	8.73×10^{-10}	1.66×10^{-18}
McCormick	8.42×10^{1}	-5.47×10^{-1}	-1.54×10^{0}	7.70×10^{-5}
Rastrigin	1.84×10^{2}	-4.16×10^{0}	1.16×10^{0}	2.83×10^{1}
Rosenbrock	9.25×10^{1}	9.73×10^{-1}	9.49×10^{-1}	1.87×10^{-3}
Schaffer N.2	7.94×10^{1}	8.04×10^{-9}	1.80×10^{-8}	0.00×10^{0}

Continued on next page

Table 5.4 continued from previous page

Function	Generations	Fittest X	Fittest Y	Error
Schaffer N.4	9.84×10^{1}	-2.19×10^{-3}	-3.59×10^{-5}	1.46×10^{-8}
Sphere	2.91×10^{2}	-2.97×10^{-36}	-4.45×10^{-36}	8.61×10^{-71}
Styblinski-Tang	8.90×10^{1}	-2.90×10^{0}	-2.90×10^{0}	3.91×10^{1}
Three-Hump Camel	2.94×10^{2}	3.87×10^{-35}	-2.91×10^{-35}	8.18×10^{-69}
Average	1.40×10^{2}	—	—	6.82×10^{0}
S. Deviation	7.61×10^{1}	—	—	1.75×10^{1}

5.4.3 Roulette Selector

Table 5.5: Time spent in each phase of the evolution process with a Random Selector.

Function	Selection (ms)	Total (ms)
Ackley	9.94×10^{-2}	6.27×10^{2}
Beale	7.46×10^{-2}	6.29×10^{2}
Booth	6.45×10^{-2}	6.17×10^{2}
Bukin N.6	6.45×10^{-2}	6.26×10^{2}
Cross-in-tray	6.35×10^{-2}	6.54×10^{2}
Easom	6.01×10^{-2}	6.16×10^{2}
Eggholder	6.21×10^{-2}	4.80×10^{2}
Goldstein-Price	6.41×10^{-2}	6.30×10^{2}
Himmelblau	6.38×10^{-2}	6.30×10^{2}
Holder Table	6.18×10^{-2}	6.49×10^{2}
Levi	6.81×10^{-2}	7.41×10^{2}
Matyas	6.87×10^{-2}	7.10×10^{2}
McCormick	1.62×10^{-1}	1.50×10^{3}
Rastrigin	6.40×10^{-2}	6.28×10^{2}
Rosenbrock	6.44×10^{-2}	6.35×10^{2}
Schaffer N.2	6.44×10^{-2}	6.28×10^{2}
Schaffer N.4	6.46×10^{-2}	6.29×10^{2}
Sphere	6.81×10^{-2}	7.14×10^{2}
Styblinski-Tang	6.99×10^{-2}	6.92×10^{2}
Three-Hump Camel	6.82×10^{-2}	6.81×10^{2}
Average	7.21×10^{-2}	6.86×10^{2}
S. Deviation	2.21×10^{-2}	1.94×10^{2}

Table 5.6: Results of the optimization process with a Random Selector.

Function	Generations	Fittest X	Fittest Y	Error
Ackley	5.00×10^{2}	-6.50×10^{-2}	1.70×10^{-2}	3.47×10^{-1}
Beale	5.00×10^{2}	1.81×10^{0}	1.74×10^{-1}	9.55×10^{-1}
Booth	5.00×10^{2}	1.12×10^{0}	2.97×10^{0}	1.84×10^{-1}
Bukin N.6	5.00×10^{2}	-9.85×10^{0}	-3.02×10^{-2}	2.18×10^{-3}
Cross-in-tray	5.00×10^{2}	4.52×10^{-1}	4.57×10^{-1}	2.00×10^{-3}
Easom	5.00×10^{2}	3.10×10^{0}	3.16×10^{0}	8.30×10^{-3}
Eggholder	2.12×10^{2}	-3.89×10^{1}	3.54×10^{2}	3.42×10^{2}
Goldstein-Price	5.00×10^{2}	-4.49×10^{-2}	-9.90×10^{-1}	4.14×10^{0}
Himmelblau	5.00×10^{2}	1.47×10^{0}	1.12×10^{0}	2.70×10^{1}
Holder Table	5.00×10^{2}	2.57×10^{0}	-9.37×10^{0}	1.19×10^{0}
Levi	5.00×10^{2}	9.88×10^{-1}	7.47×10^{-1}	2.03×10^{-1}

Continued on next page

Table 5.6 continued from previous page

Function	Generations	Fittest X	Fittest Y	Error
Matyas	5.00×10^{2}	-4.70×10^{-2}	-6.89×10^{-2}	2.70×10^{-4}
McCormick	5.00×10^{2}	-6.20×10^{-2}	-1.01×10^{0}	5.11×10^{-1}
Rastrigin	5.00×10^{2}	1.65×10^{0}	8.58×10^{-1}	8.92×10^{0}
Rosenbrock	5.00×10^{2}	7.48×10^{-1}	6.04×10^{-1}	1.89×10^{-1}
Schaffer N.2	5.00×10^{2}	2.99×10^{-1}	-1.03×10^{0}	2.10×10^{1}
Schaffer N.4	5.00×10^{2}	2.49×10^{-1}	1.27×10^{0}	3.18×10^{-2}
Sphere	5.00×10^{2}	-3.47×10^{-2}	-5.68×10^{-3}	8.54×10^{-3}
Styblinski-Tang	5.00×10^{2}	-2.84×10^{0}	-2.84×10^{0}	3.84×10^{1}
Three-Hump Camel	5.00×10^{2}	-1.15×10^{-2}	4.76×10^{-2}	6.70×10^{-3}
Average	4.85×10^{2}	—	—	2.22×10^{1}
S. Deviation	6.26×10^{1}	—	—	7.41×10^{1}

A primary observation from the analysis is the minimal standard deviation in selection times across all selectors, suggesting remarkable stability and consistency in the selection phase. Notably, the Random Selector exhibits the lowest average selection time, attributed to its lack of computational requirements for individual selection. In contrast, both the Tournament and Roulette selectors show comparable average selection times, with the Roulette selector marginally slower. This slight increase in time for the Roulette selector stems from its necessity to calculate fitness weights for each individual - a more computationally demanding task than the Tournament selector's simple fitness comparison.

When examining average evolution times, we find that they are relatively similar across all selectors, approximately in the order of 10^{-1} seconds. The Tournament selector, however, demonstrates a marginal speed advantage. This efficiency is linked to its ability to avoid iterating through all generations to find an adequate solution, as it effectively maintains the optimal solution once discovered. The Roulette selector, in comparison, is slightly slower than the Random selector due to the additional computational load of fitness weight calculations.

Focusing on the evolution outcomes, a distinct advantage of the Tournament selector emerges. It uniquely halts the evolutionary process before reaching the maximum generation limit, achieving 50 steady generations earlier. This efficiency is rooted in its capability to sustain an optimal solution once it is identified, unlike the other selectors.

Further analysis reveals that the Tournament selector consistently yields the lowest average error, indicating its superiority in finding the most optimal solutions. The Roulette and Random selectors exhibit similar average errors, with the Roulette selector slightly outperforming the Random one. This observation is reinforced by the lower standard deviation in error for the Roulette selector, implying greater consistency in its results.

Notable exceptions to the Tournament selector's low error performance are observed with the Rastrigin, Egg Holder, and Schaffer functions. These functions are characterized by multiple, evenly distributed local optima, posing a significant challenge for the algorithm to escape once convergence is achieved. Consequently, this leads to the Tournament selector's inability to locate the global optimum in these cases. In contrast, functions like Booth, with a single global optimum and a limited set of local optima, present a less complex landscape, enabling easier escape and convergence to the global optimum.

5.5 Conclusion

In conclusion, the comprehensive analysis conducted in this section offers insightful revelations into the performance dynamics of different selection strategies within the *Keen* framework. The Tournament Selector emerges as a notably efficient strategy, particularly in maintaining optimal solutions and achieving the lowest average error across various functions. Its proficiency in dealing with a spectrum of optimization challenges underscores its adaptability and robustness. Conversely, the Random and Roulette Selectors, while occasionally attaining the global optimum, demonstrate limitations in sustaining optimal solutions over extended periods. This observation suggests the suitability of a target fitness criterion for these selectors, contrary to the stagnation termination criterion effective for the Tournament Selector. The consistent selection times and evolutionary outcomes observed across all selectors reflect the underlying stability of the *Keen* framework. However, the variable performance in dealing with functions characterized by multiple local optima highlights the importance of selector choice, tailored to the specific nature of the optimization problem at hand. Overall, these findings contribute significantly to understanding

the strengths and limitations of various selection strategies in genetic algorithm optimization, providing valuable guidance for their application in diverse computational contexts.

Chapter 6

Case Study: Crash Reproduction

6.1 Introduction

The realm of automatic test case generation in software engineering has seen significant advancements, particularly with the integration of Generative Pre-trained Transformer (GPT) models. Tools like *ChatGPT*, *GitHub Copilot*, and *JetBrain's AI Assistant* exemplify the progress in this field. A notable development in this area is *ChatUniTest* by Xie et al. [52], a GPT-based tool for automated Java test case generation. It has shown promise when compared to established tools like *Evosuite* [29] and *Randoop* [20]. Despite these advancements, concerns about the environmental impact of large-scale language models, such as significant water and electricity consumption, cannot be overlooked [44, 70].

Parallel to general test case generation, crash reproduction—a niche yet critical area in software engineering—focuses on generating test cases to replicate specific crashes for debugging purposes. This task involves navigating extensive search spaces and understanding intricate codebase semantics. Bergel & Slater's *Beacon* [40], an automated tool for Python, leverages a genetic algorithm to create Minimal Crash Reproduction (MCR) test cases. Their work demonstrates the efficacy of this approach across various exceptions and program types. However, a detailed performance analysis and comparison with other state-of-the-art tools are absent in their study.

A promising avenue within this domain is Linear Genetic Programming (LGP), characterized by representing individuals as linear instruction sequences. The output of the final instruction in this sequence defines the individual's output. Building upon the concept introduced by Bergel & Slater, this chapter presents a novel LGP-based approach for generating MCR test cases in Kotlin. We aim to conduct an exhaustive performance analysis of our methodology. Due to the nascent stage of Kotlin-specific crash reproduction tools, direct performance comparisons with other established tools will not be pursued. Instead, our focus will be on assessing our approach's effectiveness in replicating crashes induced by diverse exceptions in various program types.

6.2 Problem Description

The primary goal of this study is to demonstrate a complex application of the *Keen* framework, focusing on the challenging domain of crash reproduction in Kotlin programs. Specifically, our objective is to replicate stack traces accurately while ensuring that the resultant Kotlin programs are not only syntactically correct but also concise, ideally minimized in terms of code length.

This task presents itself as a maximization problem, where the aim is to maximize the similarity between the original program's stack trace and that of the generated program. The inherent complexity lies in producing a Kotlin program that not only triggers a stack trace akin to the original but does so with the minimal number of code lines.

Given the vastness of the search space, encompassing all conceivable Kotlin programs, practical constraints are necessary to render this problem solvable. To this end, we confine our search to Kotlin programs with a maximum length of 10 lines, comprising a predefined set of functions capable of handling up to three arguments.

It is important to note, however, that even this "constrained" search space remains infinitely large. This infinity is evident when considering a function with a single real number argument ($\mathbb{R}$), representing an uncountably infinite set.

Similar to our previous case studies, the effectiveness of the *Keen* framework in navigating this immensely large search space will be evaluated. This will be accomplished by comparing the results achieved through the framework against those obtained from a random search algorithm, highlighting *Keen*'s capabilities in efficiently addressing complex problems in the realm of crash reproduction.

Specifically, we will try to reproduce two test cases:

1. **TC1**: A program that throws an `IllegalArgumentException` with the message "The number is greater than 100"; and
2. **TC2**: A program that throws an `IllegalArgumentException` that has been thrown by a function with the name `throwIAE`.

The concrete functions used in this study are presented in listing C.4 on page 120.

6.3 Solution

Our objective is to design a genetic algorithm that creates a program P', which, when executed, throws an exception E' with a stack trace S' closely resembling the stack trace S of a given program P throwing exception E. To achieve this, we utilize the flexibility of the *Keen* framework, integrating dynamic manipulation of Kotlin code through reflection. This approach allows our LGP model to interact directly with Kotlin code.

6.3.1 Genetic Representation

In our genetic algorithm, a program is conceptualized as a list of functions. Correspondingly, we define a function as a gene and a program as a chromosome. To adapt the *Keen* framework to this novel genetic structure, we extend its capabilities by implementing the `Gene` and `Chromosome` interfaces.

The following Kotlin code illustrates the custom gene implementation, representing a function in our genetic model:

Listing 6.1– Caption: Implementation of a custom gene to represent a Kotlin function.

```
class KFunctionGene(override val value: KFunction<*>, val functions: List<KFunction<*>>) :
    Gene<KFunction<*>, KFunctionGene> {
    val arity = value.parameters.size
    override fun duplicateWithValue(value: KFunction<*>) = KFunctionGene(value, functions)
    override fun generator() = functions.random(Domain.random)
    operator fun invoke(args: List<*>) = value.callBy(value.parameters.zip(args).toMap())
}
```

The following code demonstrates the custom chromosome implementation, which represents a Kotlin program:

Listing 6.2– Caption: Implementation of a custom chromosome to represent a Kotlin program.

```
class KFunctionChromosome(override val genes: List<KFunctionGene>)
    : Chromosome<KFunction<*>, KFunctionGene> {
    override fun duplicateWithGenes(genes: List<KFunctionGene>) =
        KFunctionChromosome(genes)
    class Factory(override var size: Int, val geneFactory: () -> KFunctionGene) :
        Chromosome.AbstractFactory<KFunction<*>, KFunctionGene>() {
        override fun make() = KFunctionChromosome((0..<size).map { geneFactory() })
    }
}
```

These straightforward representations lay the groundwork for solving intricate problems. One of *Keen*'s key strengths is its user-centric design, enabling easy customization of genetic material to address various computational challenges.

6.3.2 Fitness Function

The fitness function ϕ, pivotal to our genetic algorithm, is formulated to evaluate how closely the generated program P' replicates the exception characteristics and stack trace of a given program P. Let E denote the exception thrown by P with stack trace S. The fitness function is defined as:

$$\phi(E, S) = 2 \cdot \delta_C(E) + \delta_M(E) + 2 \cdot \Delta_F(S) \tag{6.1}$$

where:

- $\delta_C(E)$ represents the *class equality* between the exception E and the exception thrown by P'. It is assigned a value of 1 if both exceptions are of the same class, and 0 otherwise.
- $\delta_M(E)$ denotes the *message equality* between the exception E and the exception thrown by P'. It evaluates to 1 if both exceptions share identical messages, and 0 otherwise.
- $\Delta_F(S)$ signifies the *function presence* in the stack trace comparison between S and the stack trace of P'. It takes a value between 0 and 1, representing the proportion of functions in S present in the stack trace of P'.

This fitness function aims to maximize the congruence between the stack traces and exceptions of the original and the generated programs. The first two components, $\delta_C(E)$ and $\delta_M(E)$, focus on the fidelity of the exception type and message in P' relative to P. The third component, $\Delta_F(S)$, assesses the stack trace's accuracy in reflecting the original program's execution path. The assigned weights to these components mirror their respective importance in achieving a faithful reproduction. The chosen values are adaptations from Bergel & Slater's methodology [40], which in turn are inspired by the work of Soltani et al. [36].

The optimal score of the fitness function is 5, achievable when P' throws an exception mirroring the class and message of E and its stack trace S' includes all functions in S from the original program.

The fitness function is then implemented as in the following pseudocode:

Listing 6.3– Implementation of the fitness function.

```
fun fitness(genotype) = genotype.let { functions ->
    try {
        variables = []
        functions.forEach { f ->
            variables += f(variables.takeLast(f.arity))
        }
        0.0
    } catch (E) {
        deltaC = classOf(E) == classOf(exception)
        deltaM = E.message == exception.message
        DeltaF = E.stackTrace.count { it.function in functions } / E.stackTrace.size
        2 * deltaC + deltaM + 2 * DeltaF
    }
}
```

This fitness function implementation works as follows:

- It iteratively executes each function in the list (genotype), passing the last 'arity' number of arguments from a dynamically maintained list 'variables'.
- If no exception is thrown during execution, the function returns a fitness score of 0.0.
- In case of an exception, the function calculates three components: class equality (δ_C), message equality (δ_M), and function presence (Δ_F), which contribute to the fitness score.
- The final score is computed as $2 \cdot \delta_C + \delta_M + 2 \cdot \Delta_F$, encapsulating the similarity of the thrown exception and its stack trace to those of the target exception.

Additionally, each defined function will return either an integer or *nothing* to avoid type mismatches during execution. Note that, given our approach, since the fitness function will catch any exception thrown by the program, the program will always terminate gracefully, regardless of the exception type, meaning that even if type mismatches occur, the program will still calculate a fitness score correctly.

This approach, while simplified, is expected to generate sufficiently challenging test cases to assess the genetic algorithm's effectiveness in crash reproduction scenarios.

6.3.3 Genetic Operators

The genetic algorithm for our crash reproduction problem employs a set of specific genetic operators to navigate the search space effectively. These operators are crucial for guiding the algorithm towards optimal solutions.

Selection Operator: We use a *Tournament Selector* with a tournament size of 3. This selector picks three individuals randomly and selects the best among them, based on their fitness scores.

Crossover Operators: To test the effectiveness of different crossover strategies, we examine two distinct crossover operators:

- **Single-Point Crossover:** This operator selects a random point within the chromosome and swaps the genes between the two parents at that point. It is useful for exchanging genetic material between the parents.
- **Uniform Crossover:** This operator iterates through the chromosome and returns a gene from either parent with equal probability. This operator is not natively supported by the *Keen* framework. However, we can implement it succinctly using a Combine Crossover like so:

Listing 6.4– Implementation of the uniform crossover operator using the *Keen* framework.

```
val crossover = CombineCrossover({ it.random(Domain.random) })
```

Mutation Operators: To investigate the effectiveness of different mutation strategies, we examine three distinct mutation operators:

- **Random Mutation:** This operator targets a randomly chosen gene in the chromosome, replacing it with a new randomly generated gene. It introduces diversity by altering gene values.
- **Swap Mutation:** This operator selects two genes at random within the chromosome and swaps their positions. It is useful for rearranging the existing genetic structure without introducing new genes.
- **Inversion Mutation:** This operator inverts the order of a randomly chosen subset of genes within the chromosome. It is effective for changing the gene sequence, which might lead to significant variations in the resultant program behavior.

Through the comparison of these genetic operators, we aim to evaluate their respective contributions to the overall effectiveness of the genetic algorithm in reproducing crash scenarios accurately.

6.3.4 Program Minimization

The genetic algorithm aims to produce a program that closely replicates the target program's exception and stack trace. However, the resulting program may not be the most concise in terms of code length. To refine the output, we introduce a program minimization procedure, designed to shorten the program while maintaining its essential functionality.

This minimization process is executed as a post-processing step after the genetic algorithm concludes. It follows a systematic approach to reduce the program's length:

1. **Iterative Reduction:** The algorithm begins by sequentially removing functions, starting from both the end and the beginning of the program.
2. **Fitness Evaluation:** After each removal, the altered program's fitness score is reassessed.
3. **Decision Making:** If the fitness score remains consistent with the original score, the function removal is deemed non-impactful, and the reduction is retained. This step ensures that only non-critical functions are eliminated.

4. **Termination:** The process continues until any further removals lead to a change in the fitness score, suggesting a functional alteration in the program. At this point, the minimization ceases.

The resulting program, shorter yet functionally akin to the original output, is then presented as the final solution.

It is important to note that while this method efficiently reduces program length, it does not guarantee the shortest possible solution. The primary goal is to strike a balance between brevity and functionality, yielding programs that are sufficiently concise for effective evaluation in crash reproduction scenarios. The effectiveness of this minimization technique serves as a testament to the genetic algorithm's capability in generating robust and near-optimal solutions.

6.3.5 Implementation

The genetic algorithm for crash reproduction is encapsulated in a tool we have developed, named *Tracer*. This tool leverages the capabilities of the *Keen* framework to efficiently navigate the problem space. Below is a snippet demonstrating the core implementation of the evolution engine within *Tracer*:

Listing 6.5– Caption: Implementation of the genetic algorithm using the *Keen* framework.

```
evolutionEngine(
    ::fitnessFunction, // Defines the fitness function
    genotypeOf { // Constructs the genotype
        chromosomeOf {
            KFunctionChromosome.Factory(10) { // Sets the chromosome size
                KFunctionGene(functions.random(Domain.random), functions) // Creates a gene
            }
        }
    }
) {
    populationSize = 1000 // Specifies the size of the population
    alterers += listOf(mutator, SinglePointCrossover(0.3))
    limits += listOf(TargetFitness(5.0), GenerationLimit(1000))
    listeners += this@Tracer.listeners // Adds event listeners for monitoring the algorithm
}
```

This implementation highlights several key components:

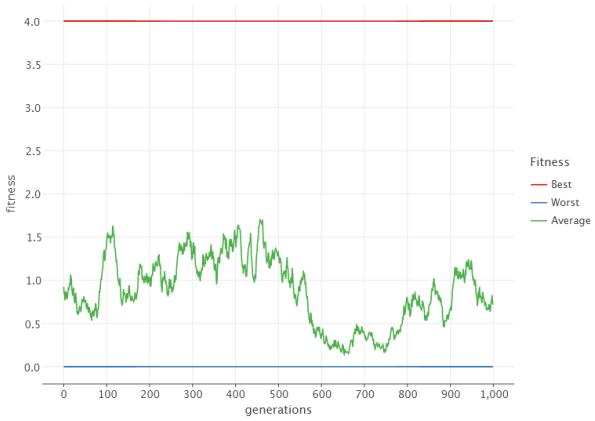
- **Fitness Function:** The `::fitnessFunction` evaluates the proximity of the generated program's behavior to the target.
- **Genotype and Chromosome Structure:** The algorithm creates chromosomes composed of function genes, reflecting the potential solution structure.
- **Population Size:** A larger population size of 1000 individuals is set to provide a diverse genetic pool.
- **Alterers:** Both mutation and crossover strategies are employed to introduce genetic variations.
- **Termination Criteria:** The algorithm terminates upon achieving a target fitness or reaching the maximum number of generations.
- **Listeners:** Event listeners are integrated for tracking the algorithm's progress.

The complete implementation of the *Tracer* tool, encompassing these elements, can be found in the accompanying code listing referenced at listing C.5 on page 121.

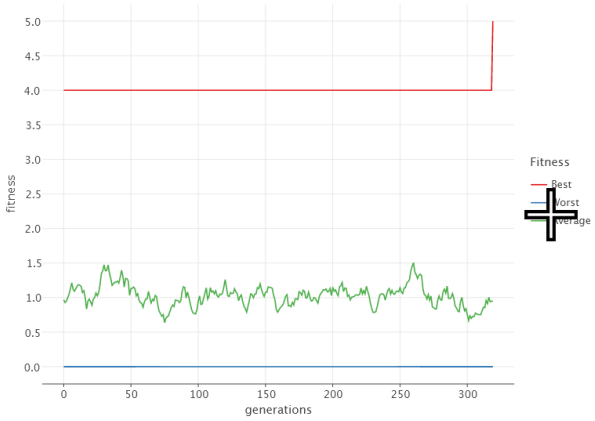
6.4 Results and Discussion

This section presents an evaluation of our crash reproduction approach, focusing on its performance across various genetic operators. We conducted experiments using three mutation operators—inversion, random, and swap—and two crossover operators, namely single-point and uniform. To illustrate the fitness evolution achieved by each combination of these genetic

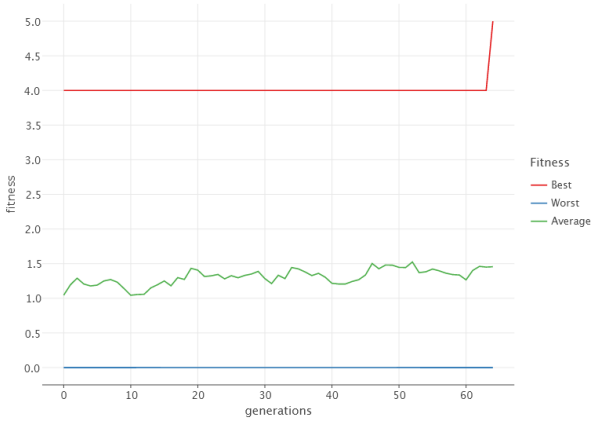
operators, we refer to Figures 6.1 and 6.2. Our analysis covers two test cases: `TC1` and `TC2`. The performance results for `TC1` are detailed in fig. 6.1, while those for `TC2` are depicted in fig. 6.2 on the next page. These results provide insights into the efficacy of different genetic operator combinations in the context of crash reproduction.



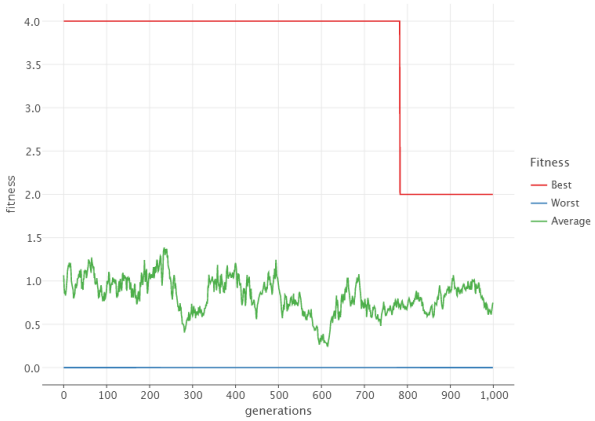
(a) Inversion Mutation with a Single Point Crossover



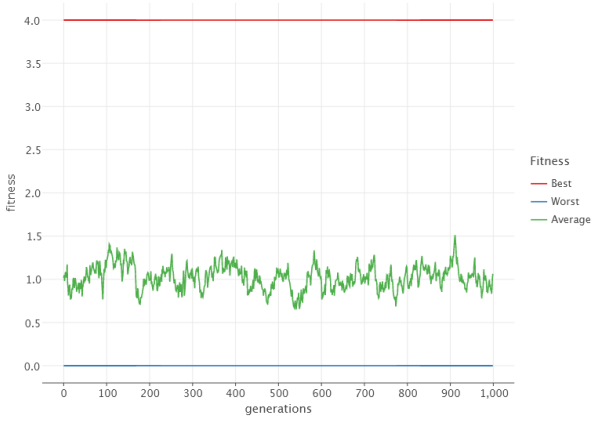
(b) Random Mutation with a Single Point Crossover



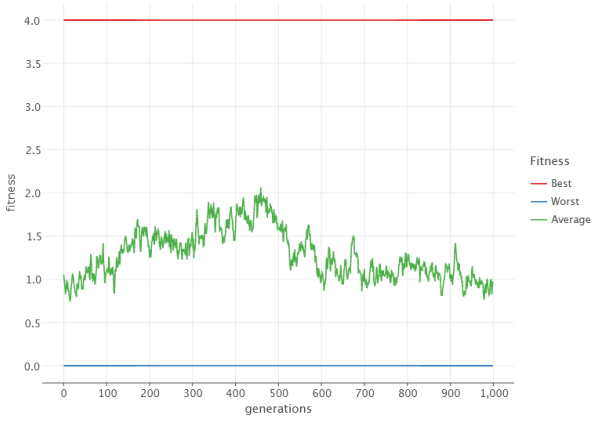
(c) Swap Mutation with a Single Point Crossover



(d) Inversion Mutation with a Uniform Crossover



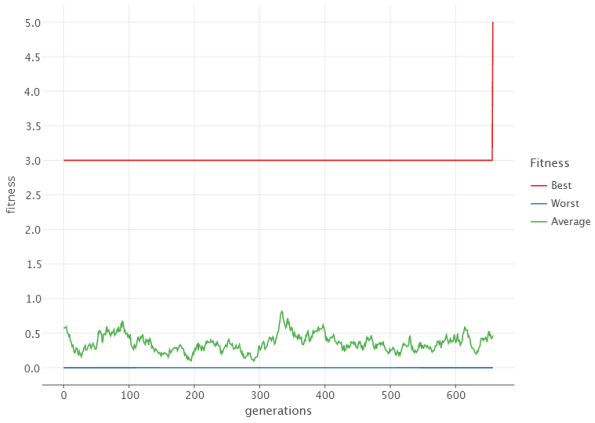
(e) Random Mutation with a Uniform Crossover



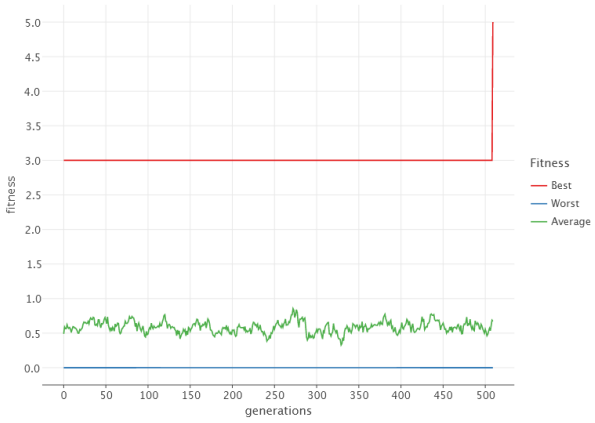
(f) Swap Mutation with a Uniform Crossover

Figure 6.1: Comparison of the three genetic operators on the `TC1` test case.

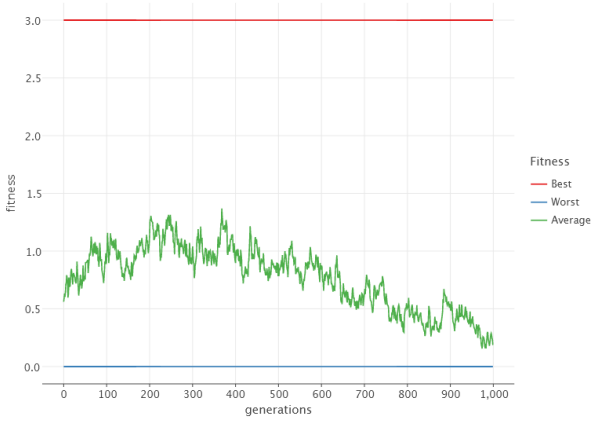
The experimental results demonstrate varied performances among different genetic operators in the context of crash reproduction. Specifically, for test case `TC1`, the inversion and swap mutation operators paired with single-point crossover achieved the target fitness value of 5.0, whereas the random mutation operator did not. In contrast, when utilizing uniform crossover, none of



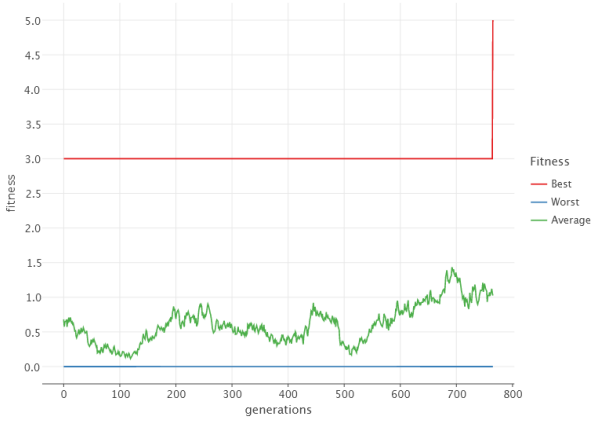
(a) Inversion Mutation with a Single Point Crossover



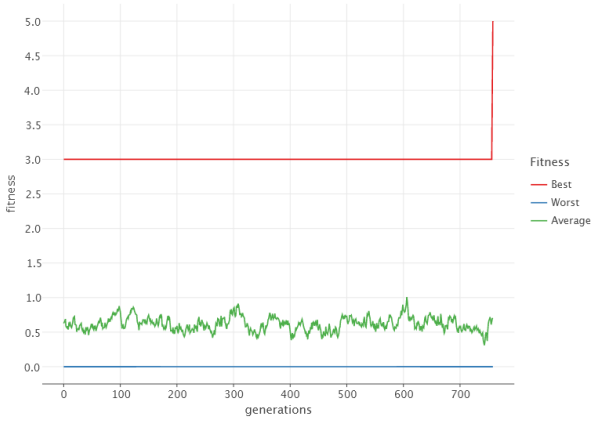
(b) Random Mutation with a Single Point Crossover



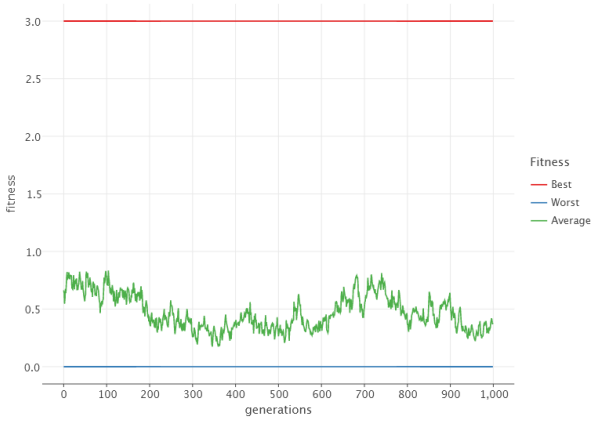
(c) Swap Mutation with a Single Point Crossover



(d) Inversion Mutation with a Uniform Crossover



(e) Random Mutation with a Uniform Crossover



(f) Swap Mutation with a Uniform Crossover

Figure 6.2: Comparison of the three genetic operators on the TC2 test case.

the operator combinations attained the fitness goal of 5.0. For TC2, both inversion and random mutation operators with single-point crossover reached the target fitness, while the swap mutation operator fell short. However, using uniform crossover, the inversion and swap mutation operators were successful in achieving the fitness target, unlike the random mutation operator.

These findings imply that the effectiveness of genetic operators significantly depends on the specific test case. In TC1, inversion and swap mutation operators outperform the random mutation operator, while in TC2, inversion and random mutation operators show superior performance compared to the swap mutation operator. Additionally, the uniform crossover operator

generally underperforms relative to the single-point crossover in crash reproduction tasks.

There was no noticeable trend in average fitness improvement over time, suggesting that the fitness function might not be optimally aligned with the crash reproduction problem. This lack of convergence on a solution could be due to the fitness function's inability to effectively differentiate solutions nearing the target fitness value.

The outcomes might also indicate limitations in the simplified LGP (Linear Genetic Programming) implementation used for this project. A more advanced LGP implementation or a Multi-Objective Genetic Programming (MOGP) approach might yield better results. MOGP could optimize both the target fitness value and the solution's instruction count, potentially leading to solutions that are both close to the desired fitness value and have fewer instructions, thus more likely representing practical crash reproduction solutions.

Given these insights, we conclude that a statistical analysis of the results is not necessary, as the current findings do not provide a conclusive basis for such an analysis.

6.5 Conclusion

Our comprehensive evaluation of various genetic operators in the context of crash reproduction reveals key insights into their effectiveness. The experiments, conducted across different test cases (`TC1` and `TC2`), demonstrate that the performance of these genetic operators is highly contingent on the specific characteristics of the test case. Notably, for `TC1`, inversion and swap mutation operators combined with single-point crossover successfully achieved the target fitness value, whereas the random mutation operator did not. Conversely, with the uniform crossover, none of the operator combinations met the fitness goal. In the case of `TC2`, a different pattern emerged, indicating a distinct response of the genetic operators to varying test conditions.

The absence of a clear trend in average fitness improvement over time suggests potential misalignment of the fitness function with the nuances of crash reproduction. This lack of convergence towards a solution hints at the fitness function's limitations in distinguishing near-optimal solutions. Furthermore, the results may reflect the constraints of the simplified LGP implementation employed in this study. A more sophisticated LGP framework or a Multi-Objective Genetic Programming approach, which balances the optimization of the target fitness value and instruction count, could potentially enhance performance. Such a strategy might yield solutions that are not only proximal to the target fitness value but also characterized by fewer instructions, aligning more closely with practical crash reproduction scenarios.

Considering these findings, we deem a statistical analysis of the results unnecessary. The current data, while insightful, does not provide a definitive foundation for such an analysis. Moving forward, the exploration of more advanced genetic programming techniques and the refinement of fitness functions tailored to crash reproduction may unlock further advancements in this field.

Chapter 7

Conclusions

7.1 Summary and Contributions

This thesis explores the domain of Evolutionary Computation (EC), a subset of Artificial Intelligence (AI) that utilizes natural selection principles to address complex challenges. EC encompasses a diverse range of algorithms, each distinct in capabilities and applications. The primary focus of this work is the application of EC in the realms of software engineering and scientific computing.

A significant contribution of this thesis is the development of a Kotlin-based framework for EC algorithms. The framework stands out for its modular and scalable architecture, facilitating the incorporation of novel algorithms and integration with existing ones.

An extensive review of existing EC frameworks was undertaken to inform the development of this novel framework. This review involved a comprehensive analysis of leading frameworks, identifying their key characteristics and functionalities. These insights were instrumental in shaping the design and development of the Kotlin-based EC framework.

The efficacy of the framework is demonstrated through two distinct case studies. The first involves applying a Genetic Algorithm (GA) to solve classic numerical optimization challenges, showcasing the framework's ability to tackle complex problems. The second case study implements a simplified Linear Genetic Programming (LGP) algorithm to replicate software crashes. While the results were mixed, they underscore the framework's potential in facilitating more intricate EC algorithms.

The framework presented here is an evolving entity, with ongoing enhancements anticipated. As an open-source project, it encourages collaboration and continuous improvement within the community. The framework is accessible at `www.github.com/r8vnhill/keen`, inviting contributions and further exploration in the field of EC.

A cornerstone of this thesis is the in-depth exploration of the theoretical aspects of Evolutionary Computation (EC). It delves into the foundational concepts, methodologies, and principles of EC, providing a rich, academic context that informs the practical applications developed. This comprehensive theoretical base not only enhances the understanding of EC but also serves as a reference for future research in the field.

An innovative outcome of this research is the creation of *StraitJakt*, a data validation tool developed using Kotlin. This tool emerged as a natural extension of the need for robust data validation within the EC framework. *StraitJakt* excels in validating the integrity of data inputs and outputs, offering a highly expressive and user-friendly interface. Its development signifies a contribution to the realm of data validation, particularly in applications involving complex computational algorithms. Available as an open-source resource, *StraitJakt* can be accessed at `www.github.com/r8vnhill/strait-jakt`, inviting developers and researchers to utilize and further enhance its capabilities.

7.2 Future Work

The *Keen* framework is still growing and has much potential for further development. We plan to expand its features and keep it up-to-date with the constantly changing field of Evolutionary Computation (EC). The upcoming improvements include:

- **Incorporation of Additional EC Algorithms:** A key area of focus is the integration of a wider array of EC algorithms into the *Keen* framework. This expansion is geared towards diversifying the framework's applicability across various complex problems and enhancing its robustness in solving diverse computational challenges.
- **Integration with Other Kotlin-based Libraries:** Another pivotal aspect of future work involves synergizing *Keen* with existing libraries in the Kotlin ecosystem. This integration aims to exploit the unique features and efficiencies offered by these libraries, thereby enriching the *Keen* framework's functionality and user experience.
- **Community Engagement and Open-Source Development:** Emphasis will also be placed on fostering a collaborative community around *Keen*. Encouraging contributions from a wide range of developers and researchers will not only enrich the framework with diverse insights but also drive innovation and continuous improvement.

7.2.1 Enhancements in Genetic Algorithms within the *Keen* Framework

The *Keen* framework is focusing on targeted enhancements in its Genetic Algorithm (GA) module. These upgrades are intended to boost the framework's capacity to handle diverse computational tasks more efficiently and effectively.

Expanding Individual Representation Capabilities Currently, the *Keen* framework's support for individual representation in GAs is limited to a select few data types. This limitation is a strategic choice, stemming from the framework's early development stage, focusing on ensuring foundational robustness and reliability. Future developments will be directed towards broadening the spectrum of supported data types. This enhancement is crucial as it will significantly widen the range of problems the framework can adeptly handle, making it more flexible and powerful in various computational contexts.

Diversification of Genetic Operators The range of genetic operators currently available in the *Keen* framework is somewhat limited. This initial scope was deliberately chosen to maintain stability and effectiveness during the framework's nascent phase. However, looking forward, there is a planned expansion in the variety of genetic operators. This expansion is not just a quantitative increase but also a qualitative enhancement. By incorporating a more diverse set of genetic operators, the framework will be able to offer more nuanced and tailored solutions to a wider array of problems, thus elevating its capacity to solve complex and varied computational tasks. For this, further research into other frameworks and libraries will be undertaken to identify the most effective and relevant genetic operators. This research will be followed by the integration of these operators into the *Keen* framework, ensuring seamless compatibility and optimal performance.

These forthcoming enhancements in the GA component of the *Keen* framework are expected to play a pivotal role in its evolution, making it a more adaptable and comprehensive tool for tackling the dynamic challenges in the field of Evolutionary Computation.

7.2.2 Advancing Genetic Programming in the *Keen* Framework

The *Keen* framework is set to advance in Genetic Programming (GP) by incorporating and improving GP techniques. This effort includes a thorough analysis of existing GP frameworks to identify their key components and effective strategies. The aim is to integrate the best aspects of these systems into *Keen*, enhancing its functionality and performance.

Expanding Primitive Set Support Presently, the *Keen* framework's capabilities in handling primitive sets in GP are restricted to a limited range of data types, a decision rooted in the need to establish a strong and reliable foundation during the early stages of development. Future enhancements are geared towards expanding this range, thereby offering a more versatile and powerful toolkit. This expansion is crucial for broadening the framework's applicability, enabling it to tackle a more diverse array of problems across various computational landscapes.

Enriching Genetic Operator Diversity The range of genetic operators in the current iteration of the *Keen* framework is intentionally limited, aimed at maintaining a balance between stability and efficacy. Moving forward, there is a concerted effort to not only increase the number of available genetic operators but also to enhance their sophistication. This will involve extensive research into existing GP frameworks and libraries to identify the most efficient and applicable operators. The subsequent integration of these operators into *Keen* aims at providing more nuanced and customized solutions for a wider range of computational challenges. Such an upgrade will not only increase the quantitative options within the framework but also significantly improve its qualitative output, ensuring seamless compatibility and optimal performance in diverse contexts.

These focused efforts to advance the Genetic Programming capabilities within the *Keen* framework signify a commitment to ongoing improvement and innovation.

7.2.3 Multi-Objective Evolution

The advancement of Multi-Objective Evolution (MOE) within the *Keen* framework constitutes a significant focus area in the roadmap for future development. MOE is a critical aspect of Evolutionary Computation, especially in scenarios where multiple, often conflicting objectives need to be simultaneously optimized. This capability is vital in many real-world applications, ranging from engineering design to financial modeling.

The first step in this development process will be to establish a robust theoretical foundation for MOE within the framework. This involves not only an understanding of the fundamental principles of multi-objective optimization but also an exploration of various strategies and approaches used in this domain. A comprehensive review of existing MOE methodologies will be conducted to identify best practices and effective techniques that can be adapted for the *Keen* framework.

Following the theoretical groundwork, the next phase will focus on the practical implementation of MOE strategies. This will include the development of algorithms capable of handling multiple objectives, such as Pareto optimization techniques. Additionally, the framework will be enhanced to support the efficient handling of multi-dimensional solution spaces, ensuring that the framework can effectively navigate and find optimal solutions in these complex environments.

Another crucial aspect will be the incorporation of performance metrics and evaluation methods specifically tailored for MOE. These metrics are essential for assessing the effectiveness of the optimization process and for comparing different solutions in a multi-objective context.

User experience and interface design will also be an important consideration. The framework will be designed to provide users with intuitive tools and visualizations to help them understand and analyze the trade-offs involved in multi-objective optimization. This user-centric approach will make the framework accessible to a broader range of users, including those who may not have a deep technical background in MOE.

In sum, the integration of Multi-Objective Evolution into the *Keen* framework will significantly enhance its capabilities, making it a more versatile and powerful tool for solving a wide range of complex problems. The goal is not only to add a new feature but to ensure that MOE is seamlessly integrated into the fabric of the framework, providing users with a sophisticated yet user-friendly tool for multi-objective optimization.

7.2.4 Neuroevolution

Expanding the *Keen* framework to encompass Neuroevolution is a significant step forward. Neuroevolution applies evolutionary algorithms to develop artificial neural networks and is especially effective for complex issues where conventional machine learning methods may not suffice. By incorporating Neuroevolution, *Keen* seeks to tap into this potential, offering a novel method for evolving intelligent systems.

The integration process will begin with a thorough investigation of current methodologies and best practices in Neuroevolution. This exploration will focus on understanding the nuances of evolving network architectures, weights, and learning rules. The framework will aim to support various forms of Neuroevolution, from simple direct encoding schemes to more sophisticated indirect encodings and topological evolution.

One of the key challenges in Neuroevolution is balancing the exploration of the search space with the exploitation of promising solutions. The *Keen* framework will incorporate adaptive mechanisms to manage this balance, ensuring efficient and effective evolution of neural networks. Special attention will be given to developing strategies that can evolve networks for a wide range of tasks, from simple function approximations to complex decision-making processes.

In addition to algorithmic development, a significant focus will be on creating user-friendly interfaces and tools for setting up, running, and analyzing Neuroevolution experiments. Visualization tools will be developed to help users intuitively understand the evolution process and the structure of the neural networks being evolved. These tools will be essential for both research and practical applications, allowing users to easily experiment with different configurations and understand the outcomes.

By incorporating Neuroevolution into the *Keen* framework, the aim is not only to expand the framework's capabilities but also to open up new possibilities for research and application in the field of evolutionary computation and artificial intelligence.

Integration with KotlinDL A strategic aspect of enhancing the Neuroevolution capabilities within the *Keen* framework involves its integration with KotlinDL, a deep learning library in Kotlin. This integration is pivotal for leveraging the strengths of KotlinDL, such as its intuitive syntax and efficient neural network management, to augment the Neuroevolution process. KotlinDL's robust set of tools and functionalities will provide a solid foundation for developing and evolving complex neural network models within the *Keen* framework.

7.2.5 Coevolution

The *Keen* framework is set to extend its capabilities by incorporating Coevolution, an advanced concept in evolutionary computation where multiple populations evolve in tandem, typically influencing each other's development. This addition is expected to significantly enhance the framework's ability to model and solve more complex, dynamic problems that are closer to real-world scenarios.

The first step in this integration is the development of a robust theoretical model for Coevolution within the *Keen* framework. This model will be based on a deep understanding of the dynamics of interacting populations, exploring various strategies such as competitive, cooperative, and parasitic coevolution. The aim is to create a flexible model that can be adapted to different types of problems and scenarios.

One of the key aspects of Coevolution is the design of effective interaction mechanisms between populations. This requires careful consideration of how individual populations impact each other's fitness landscapes and evolutionary trajectories. The *Keen* framework will focus on developing sophisticated interaction models that can capture the complex dynamics inherent in coevolving systems.

Another important aspect is the implementation of scalable algorithms capable of managing the increased computational complexity introduced by multiple interacting populations. The *Keen* framework will leverage modern computational techniques and parallel processing architectures to ensure efficient handling of coevolutionary processes.

Alongside these technical developments, attention will also be given to the user experience. Tools and visualizations will be developed to help users set up, monitor, and analyze coevolutionary experiments. These tools will be critical for understanding the intricate dynamics of coevolving populations and for drawing meaningful insights from their interactions.

7.2.6 Evolution Strategy

The future development of the *Keen* framework includes an important focus on the integration and enhancement of Evolution Strategies (ES), a class of evolutionary algorithms that emphasize the role of mutation over recombination. This expansion aims to broaden the framework's algorithmic repertoire, providing users with more nuanced tools for solving complex optimization problems.

The initial phase of incorporating ES into *Keen* involves a comprehensive analysis of existing strategies and their applications. This study will focus on understanding the unique characteristics of ES, such as self-adaptation, global search capabilities, and mutation-based exploration, and how they can be effectively implemented within the framework. The goal is to develop a robust and flexible ES module that can cater to a wide range of problem domains.

A critical aspect of this development will be the design and implementation of adaptive mutation operators. These operators are central to the success of ES, as they determine how the solution candidates explore the search space. The *Keen* framework will aim to incorporate advanced mutation strategies that can dynamically adjust to the problem landscape, thereby enhancing the efficiency and effectiveness of the search process.

Another key area of focus will be the integration of effective selection mechanisms. In ES, the selection process plays a crucial role in guiding the evolutionary search towards optimal solutions. The *Keen* framework will explore various selection techniques, from traditional deterministic approaches to more recent stochastic and rank-based methods, to provide a comprehensive set of options for users.

In addition to these algorithmic enhancements, user accessibility and interface design will be prioritized. The framework will be equipped with intuitive tools and visualizations that allow users to easily configure, run, and analyze ES experiments.

By integrating Evolution Strategies, the *Keen* framework will significantly expand its capabilities in addressing complex optimization challenges.

7.2.7 Other Evolutionary Algorithms

The ongoing development of the *Keen* framework involves the integration of a diverse array of evolutionary algorithms, each offering unique approaches to problem-solving. These algorithms, known for their effectiveness in various domains, will significantly enhance the framework's versatility and capability.

7.2.7.1 Ant Colony Optimization

Ant Colony Optimization (ACO) mimics the behavior of ants in finding optimal paths to food sources. In *Keen*, ACO will be utilized for solving complex optimization problems, especially those involving network structures like routing and graph-based problems. The integration of ACO aims to leverage its efficiency in finding high-quality solutions within large search spaces.

7.2.7.2 Artificial Immune Systems

Artificial Immune Systems (AIS) are inspired by the human immune system's ability to learn and adapt. In the *Keen* framework, AIS will be explored for its potential in pattern recognition and anomaly detection. The framework will focus on harnessing the adaptive learning and memory capabilities of AIS, making it suitable for dynamic and evolving problem environments.

7.2.7.3 Differential Evolution

Differential Evolution (DE) is known for its simplicity and effectiveness in continuous optimization problems. The integration of DE into *Keen* will focus on enhancing its capacity to handle high-dimensional and multimodal optimization tasks, offering a robust tool for engineering and scientific applications.

7.2.7.4 Estimation of Distribution Algorithms

Estimation of Distribution Algorithms (EDAs) replace traditional genetic operators with learning and sampling from probabilistic models. By incorporating EDAs, the *Keen* framework aims to offer a more nuanced approach to optimization, especially in scenarios where the relationship between variables is crucial.

7.2.7.5 Evolutionary Programming

Evolutionary Programming (EP), primarily focused on evolving finite state machines and neural network weights, will be integrated to strengthen the framework's capabilities in evolving complex, adaptive systems. EP's focus on mutation over recombination will provide a complementary approach to other evolutionary strategies in *Keen*.

7.2.7.6 Grammatical Evolution

Grammatical Evolution (GE) combines principles of genetic algorithms with grammar-based systems. In *Keen*, GE will be used for evolving programs and expressions, making it particularly useful for symbolic regression and automatic programming tasks.

7.2.7.7 Particle Swarm Optimization

Particle Swarm Optimization (PSO) is inspired by the social behavior of birds and fish. Its integration into *Keen* will focus on optimization problems where the search space can be effectively explored through the collective behavior of simple agents. PSO's applicability in multi-dimensional and continuous spaces makes it a valuable addition to the framework.

Each of these algorithms brings a unique set of strengths to the *Keen* framework, broadening its applicability and enhancing its problem-solving capabilities. The integration of these diverse evolutionary algorithms will position *Keen* as a comprehensive toolkit for researchers and practitioners in various domains seeking efficient and effective optimization solutions.

Appendix A

Glossary

In this appendix we present a glossary of terms used throughout this document that may be unfamiliar to the reader and are not defined in the main text.

A

Definition A.1 (Alteration)**.** *See section 2.2.4 on page 10*

Definition A.2 (Alterer)**.** *See definition 2.6 on page 10.*

Definition A.3 (Arity)**.** *The number of arguments a function takes, or the number of children a node has in a tree.*

E

Definition A.4 (Elitism)**.** *In evolutionary computation, elitism refers to a strategy wherein the top-performing individuals from a population are given a higher probability of being selected for the next generation. While this doesn't guarantee their direct transfer, it increases the likelihood of retaining the best solutions in subsequent generations, ensuring that the quality of the population does not degrade.*

Definition A.5 (Ephemeral Constant)**.** *A constant that is randomly generated at the start of the program and remains constant throughout the execution of the program.*
This is used to represent constant values in the program.

Definition A.6 (Evolutionary computation)**.** *Family of algorithms for global optimization inspired by the process of natural selection.*
This typically involves processes mimicking natural selection, mutation, recombination, and survival of the fittest.
The solutions to a problem are encoded as a set of "individuals" in a "population".
Over multiple generations, these individuals are selected and modified (via genetic operators like crossover and mutation) in order to find better solutions.

G

Definition A.7 (Generation)**.** *Number of iterations performed by an evolutionary algorithm.*

Definition A.8 (Genetic Diversity)**.** *In evolutionary computation, diversity refers to the degree of variation or difference in the genetic representation of individuals within a population. Maintaining diversity is essential for several reasons:*

1. ***Exploration vs. Exploitation:*** *A diverse population can explore various regions of the solution space, ensuring that the algorithm doesn't focus solely on one area. This balance between exploration (searching new areas) and exploitation (optimizing known good areas) is fundamental in evolutionary computation.*
2. ***Preventing Premature Convergence:*** *Without adequate diversity, the population may converge too quickly to a suboptimal solution, known as a local optimum. By maintaining diversity, the population has a better chance of discovering the global optimum.*
3. ***Adaptability:*** *A diverse population can better adapt to changing environments or requirements, making it more resilient against dynamic problems.*

Diversity can be measured in several ways, including genetic distance metrics, fitness-based measures, or phenotypic variance. To maintain or introduce diversity, techniques like mutation, crossover variations, immigration, and diversity preservation strategies (like fitness sharing or crowding) are employed.

Definition A.9 (Genotype)**.**

K

Definition A.10 (Kronecker delta function)**.** *Kronecker delta function is a function of two variables, usually integers, which is 1 if they are equal, and 0 otherwise. Formally:*

$$\delta_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \tag{A.1}$$

M

Definition A.11 (Metaheuristics)**.** *Problem-independent algorithmic method that yields a sufficiently good solution within reasonable time for an optimization problem, especially for complex problems where an exact solution is not crucial.*

Definition A.12 (Multimodal function)**.** *A multimodal function is a function that has multiple local maxima and/or multiple local minima within a given domain or interval. In other words, it has several peaks and troughs. The term "multi" in "multimodal" signifies the presence of multiple modes or peaks in the function.*
Mathematically, if f is a multimodal function in an interval $[a,\ b]$, then there exist multiple points $c_1,\ c_2,\ \ldots,\ c_n$ in $[a,\ b]$ such that f has local maxima and/or minima at these points.
Multimodal functions can be more challenging for optimization algorithms because the presence of multiple local optima can trap optimization techniques in suboptimal solutions, preventing them from finding the global optimum. This complexity often necessitates more sophisticated or stochastic optimization techniques, like simulated annealing or genetic algorithms, to effectively explore the search space.

Definition A.13 (Mutator)**.** *See definition 2.8 on page 12.*

P

Definition A.14 (Parameter optimization)**.** *Optimization problem where the solution is a set of parameters that optimize a given function.*

Definition A.15 (Phenotype)**.** *Same as* **??** *on page ??.*

Definition A.16 (Population)**.** *Set of candidate solutions to a given optimization problem.*

Definition A.17 (Program induction)**.** *Inference of an algorithm or program featuring recursive calls or repetition control structures, starting from information that is known to be incomplete, called the evidence, such as positive and negative I/O examples or clausal constraints.*

S

Definition A.18 (Sealed class)**.** *A sealed class or interface is a type of class or interface that restricts its subclassing or implementation to a limited set of classes, usually defined within the same module or file. This ensures tighter control over inheritance, allowing developers to dictate where and how a class or interface can be extended or implemented.*

Definition A.19 (Search space)**.** *Set of all candidate solutions to a given optimization problem.*

Definition A.20 (Selector)**.** *See definition 2.5 on page 9.*

U

Definition A.21 (Unimodal function)**.** *A unimodal function is a function that, within a given domain or interval, has only one local maximum and one local minimum, which can also be the global maximum and minimum respectively. In other words, the function increases to a certain point and then decreases, or vice versa. This single peak or trough is the "uni" in "unimodal".*
Mathematically speaking, if f is a unimodal function in the interval $[a,\ b]$, then there exists some point c in $[a,\ b]$ such that:

1. *$f(x)$ is increasing on $[a,\ c]$ and decreasing on $[c,\ b]$, or*
2. *$f(x)$ is decreasing on $[a,\ c]$ and increasing on $[c,\ b]$.*

V

Definition A.22 (Variadic function)**.** *Function that accepts a variable number of arguments.*

Appendix B

Test Functions for Optimization

This appendix contains the test functions used in the numerical experiments in chapter 5 on page 71.

B.1 Ackley Function

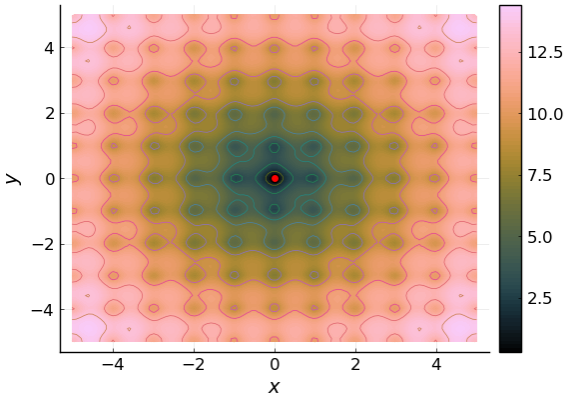
In the field of optimization, especially within evolutionary algorithms and swarm intelligence, the ***Ackley function*** serves as a prevalent benchmark function. Named after *David H. Ackley*, who introduced it during his research,[1] this function is particularly challenging for optimization algorithms due to its property of possessing a large number of local minima, despite having a single global minimum. This trait can often cause such algorithms to become entrapped in local minima.

Definition B.1 (Ackley Function)**.** *The **Ackley Function**, denoted as $f : \mathbb{R}^2 \to \mathbb{R}$, is mathematically expressed as:*

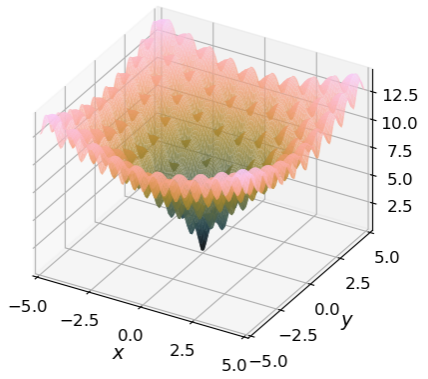
$$f(x,\, y) = -20\, e^{-0.2\sqrt{0.5(x^2+y^2)}} - e^{0.5\,[\cos(2\pi x)+\cos(2\pi y)]} + e + 20 \tag{B.1}$$

This function is evaluated within the range $x,\, y \in [-5,\, 5]$.

The global minimum of the Ackley function is located at $f(0,\, 0) = 0$. Visualizations of the Ackley function are depicted as a contour plot and a surface plot in fig. B.1.



(a) Contour plot of the Ackley Function



(b) Surface plot of the Ackley Function

Figure B.1: Illustrations of the Ackley Function with the global minimum indicated by a red dot.

[1] Ackley, D. H. (1987) "A connectionist machine for genetic hillclimbing", Kluwer Academic Publishers, Boston MA.

B.2 Beale Function

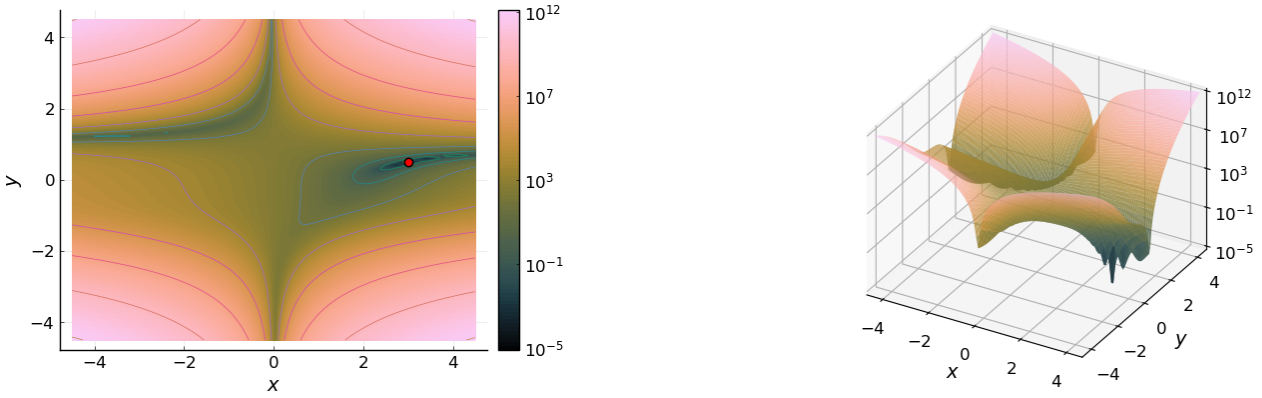
Unveiled by Beale in 1958[2], the Beale function is recognized for its multimodal characteristics and sharp peaks that define the domain's corners.

Definition B.2 (Beale Function). *The **Beale Function**, symbolized as $f : \mathbb{R}^2 \to \mathbb{R}$, is expressed by the equation:*

$$f(x, y) = (1.5 - x + xy)^2 + (2.25 - x + xy^2)^2 + (2.625 - x + xy^3)^2 \tag{B.2}$$

It's evaluated within the domain $x, y \in [-4.5, 4.5]$.

The global minimum of the Beale function is found at $f(3, 0.5) = 0$. Both contour and surface plots of the Beale function are depicted in fig. B.2.



(a) Contour plot of the Beale Function

(b) Surface plot of the Beale Function

Figure B.2: Visual representation of the Beale Function

B.3 Booth Function

The Booth function is a quadratic test problem used in the optimization field, specifically tailored for algorithms that handle two-dimensional search spaces.

Definition B.3 (Booth Function). *The **Booth function**, $f : \mathbb{R}^2 \to \mathbb{R}$, is defined as:*

$$f(x, y) = (x + 2y - 7)^2 + (2x + y - 5)^2 \tag{B.3}$$

where:

- *$x, y \in \mathbb{R}$ represent the decision variables.*

The global minimum of the Booth function is $f(1, 3) = 0$. The function is usually constrained to the square $[-10, 10]^2$. A contour plot and a surface plot of the Booth function are illustrated in fig. B.3 on the facing page.

B.4 Bukin Function N.6

The ***Bukin function N.6***, a two-dimensional benchmark problem, is renowned for its inherent complexity and frequent utilization in assessing optimization algorithms. Notable for a sharply defined, deep valley, the Bukin function N.6 presents distinct hurdles for optimization techniques owing to its abrupt discontinuity and non-differentiability at $x = 0$.

[2] Beale, E. M. (1958). "On an Iterative Method for Finding a Local Minimum of a Function of More than One Variable".

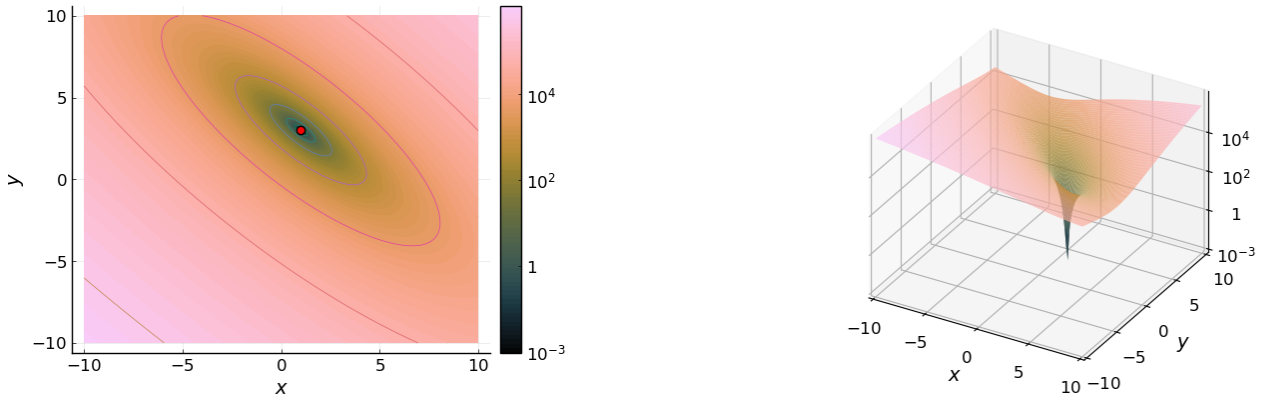


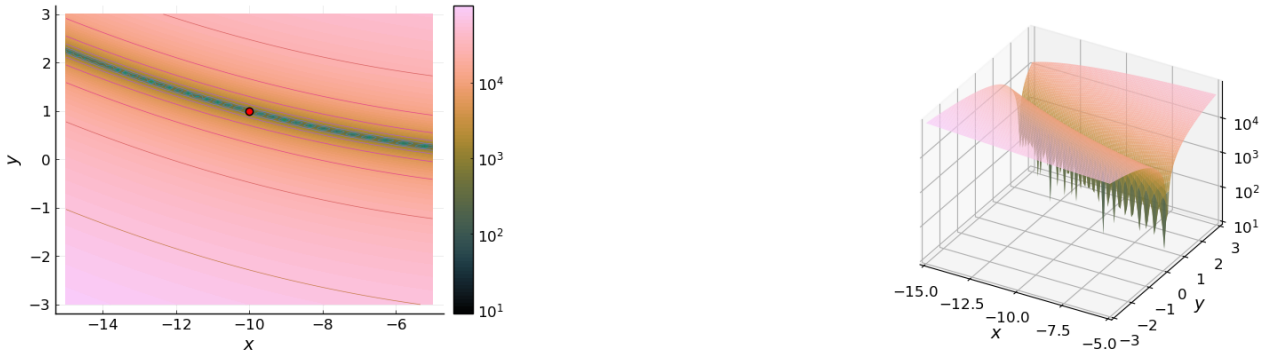
Figure B.3: Booth Function

Definition B.4 (Bukin Function N.6). *The **Bukin function N.6**, given by $f : \mathbb{R}^2 \to \mathbb{R}$, is mathematically represented as:*

$$f(x, y) = 100\sqrt{|y - 0.01x^2|} + 0.01|x + 10| \tag{B.4}$$

The decision variables, $x, y \in \mathbb{R}$, typically have the prescribed domains: $-15 \leq x \leq -5$ and $-3 \leq y \leq 3$ respectively.

The function attains its global minimum at $(x, y) = (-10, 1)$, yielding a value of zero. The Bukin function N.6, due to its unique characteristics, offers a striking visualization. A profound ridge, extending diagonally across the domain, forms a distinguishing feature. The contour and surface plots illustrating the Bukin function N.6 are presented in fig. B.4.



(a) Contour plot of Bukin function N.6

(b) Surface plot of Bukin function N.6

Figure B.4: Contour and Surface Plots of Bukin Function N.6

B.5 The Cross-in-Tray Function

The Cross-in-Tray function, known for its utility in testing optimization algorithms, poses a challenge due to its numerous local minima and four identical global minima, thereby making it an effective benchmark for evaluating an algorithm's capability to escape local optima and locate global optima.

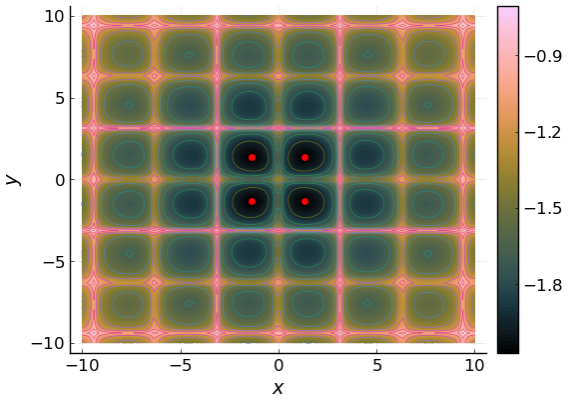
Definition B.5 (Cross-in-Tray Function). *The **Cross-in-Tray Function**, denoted as a mapping from $\mathbb{R}^2 \to \mathbb{R}$, is formally defined as:*

$$f(x,\, y) = -0.0001 \left[\left| \sin(x) \sin(y) \exp\left(\left| 100 - \frac{\sqrt{x^2 + y^2}}{\pi} \right| \right) \right| + 1 \right]^{0.1} \tag{B.5}$$

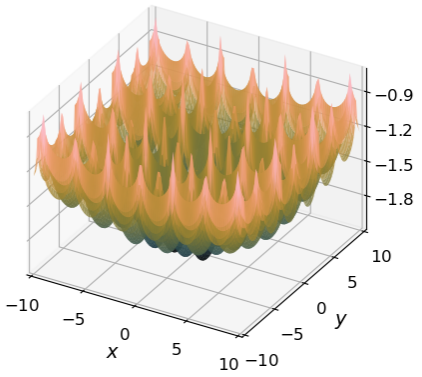
This function is usually evaluated over the domain $-10 \leq x, y \leq 10$.

This function exhibits four identical global minima located at $(\pm 1.349\,41,\, \pm 1.349\,41)$, with each of these points having the function value of $f(x,\, y) = -2.062\,61$.

In fig. B.5, contour and surface plots are presented to provide a visual understanding of the function's intricate topology.



(a) Contour plot showcasing the complex landscape of the Cross-in-Tray function. The red dot denotes the location of the global minimum.



(b) Surface plot providing a 3D representation of the Cross-in-Tray function, enhancing the visualization of its global and local minima.

Figure B.5: Contour and surface plots illustrating the topological complexity of the Cross-in-Tray function

B.6 Easom Function

The ***Easom function*** is a well-known unimodal benchmark function employed in the evaluation of optimization algorithms. It gains its name from Charles Easom and is distinctively recognized by its 'needle'-like global minimum. This feature presents a demanding task for optimization algorithms due to its confined optimal search space.

Definition B.6 (Easom Function). *The **Easom function**, defined for $f : [-100,\, 100]^2 \to \mathbb{R}$, is formally expressed as:*

$$f(x,\, y) = -\cos(x) \cos(y) \exp\left(-((x - \pi)^2 + (y - \pi)^2)\right) \tag{B.6}$$

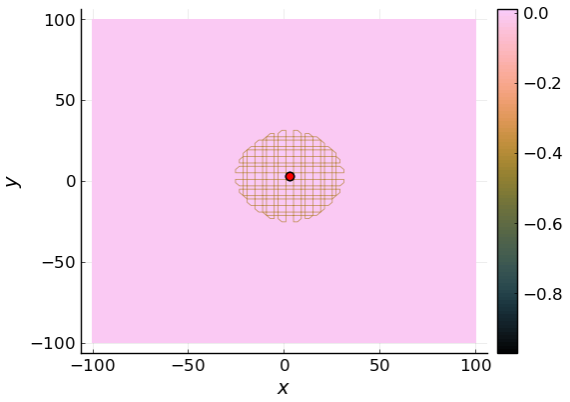
Here, x and y constitute the decision variables.

The global minimum of the Easom function is situated at the coordinates $f(\pi,\, \pi) = -1$.

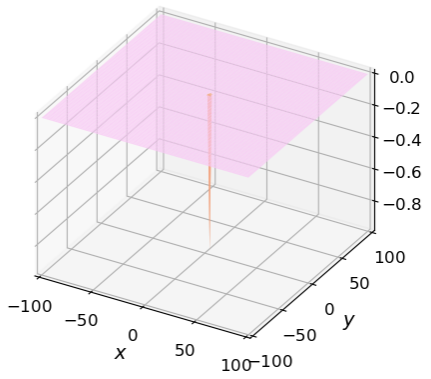
The intricate structure and narrow optimal space of the Easom function are clearly revealed in its contour and surface plots.

B.7 Eggholder Function

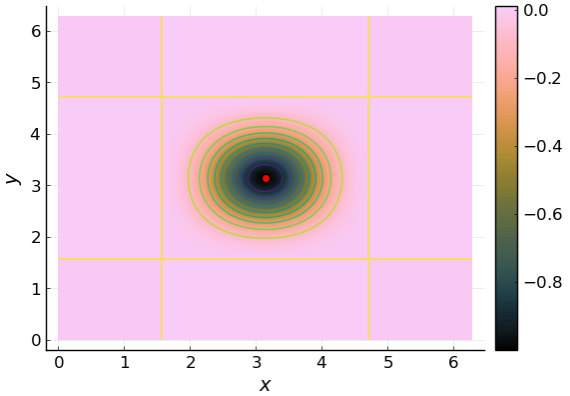
The ***Eggholder function*** is a widely-used non-convex function in the field of optimization, particularly for benchmarking optimization algorithms. The function is notorious for its multitude of local minima, presenting a complex search space that challenges the robustness and capability of an optimization algorithm to locate the global minimum.



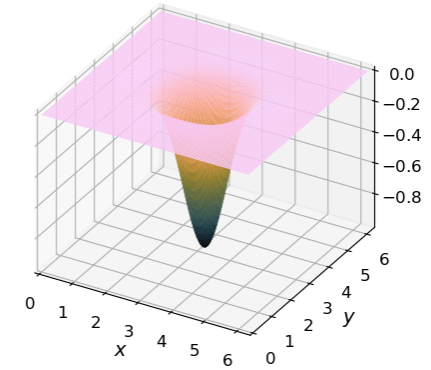
(a) Contour plot of the Easom function with the global minimum represented by the red dot



(b) Surface plot of the Easom function



(c) Contour plot of the Easom function in the vicinity of the global minimum, $[0,\ 2\pi]$.



(d) Surface plot of the Easom function in the vicinity of the global minimum, $[0,\ 2\pi]$.

Figure B.6: Contour and surface visualizations of the Easom function

Definition B.7 (Eggholder Function)**.** *Formally, the* ***Eggholder Function*** *is represented as a mapping* $\mathbb{R}^2 \to \mathbb{R}$*, and is mathematically defined as:*

$$f(x,\ y) = -(y+47)\sin\left(\sqrt{\left|\frac{x}{2}+(y+47)\right|}\right) - x\sin\left(\sqrt{|x-(y+47)|}\right) \tag{B.7}$$

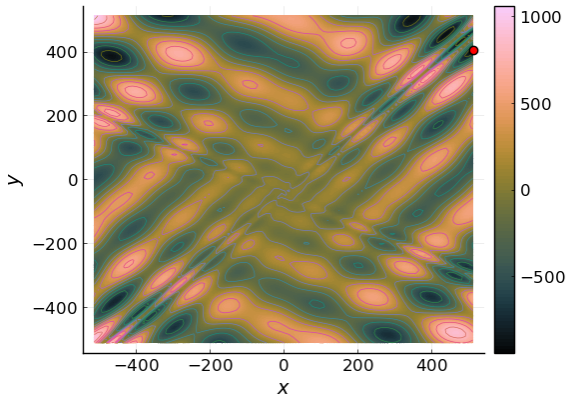
where the typical domain for evaluation spans $-512 \leq x, y \leq 512$*.*

The Eggholder function's global minimum resides at $(512,\ 404.2319)$, delivering a function value of $f(x,\ y) = -959.6407$.

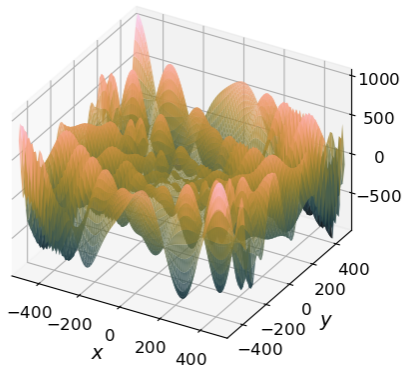
Visual representations of the function can enhance understanding of its complexity. Figure fig. B.7 on the next page offers both contour and surface plots of the Eggholder function.

B.8 Goldstein-Price Function

The ***Goldstein-Price function***, believed to be proposed by individuals named Goldstein and Price, is a challenging multimodal function recognized for its landscape densely populated with local minima. This function serves as a standard benchmark in the field of optimization, testing the efficacy of various algorithms. The precise origins of this function, however, remain elusive in academic literature.



(a) A contour plot of the Eggholder function, displaying its intricate landscape. The red dot signifies the location of the global minimum.



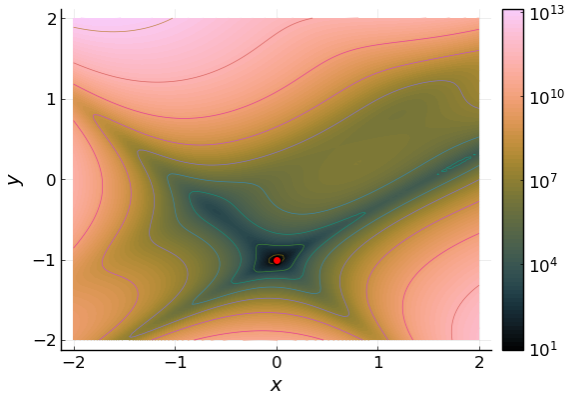
(b) A 3D surface plot of the Eggholder function, allowing a more comprehensive visualization of its local and global minima.

Figure B.7: Contour and surface visualizations of the Eggholder function, showcasing its intricate topology

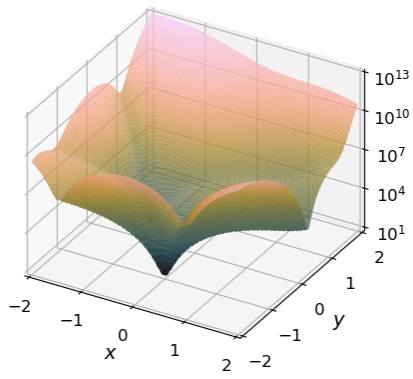
Definition B.8 (Goldstein-Price Function)**.** *The* ***Goldstein-Price function****, denoted as* $f : \mathbb{R}^2 \to \mathbb{R}$*, is formally articulated as follows:*

$$f(x, y) = \left[1 + (x + y + 1)^2 \cdot (19 - 14x + 3x^2 - 14y + 6xy + 3y^2)\right] \cdot \left[30 + (2x - 3y)^2 \cdot (18 - 32x + 12x^2 + 48y - 36xy + 27y^2)\right] \tag{B.8}$$

The function's global minimum is located at $f(x^*, y^*) = 3$ with $(x^*, y^*) = (0, -1)$. Evaluations of the Goldstein-Price function are typically performed within the range $x, y \in [-2, 2]$.



(a) Contour plot.



(b) Surface plot.

Figure B.8: Visual representations of the Goldstein-Price function

B.9 Himmelblau's Function

The ***Himmelblau's function***, attributed to David Mautner Himmelblau,[3] is a significant benchmark function in the realm of optimization, renowned for its complex multi-modal nature. Himmelblau, an American engineer, made substantial contributions to systems engineering and optimization theory.

[3] Himmelblau, D. (1972). "Applied Nonlinear Programming". McGraw-Hill. ISBN 0-07-028921-2.

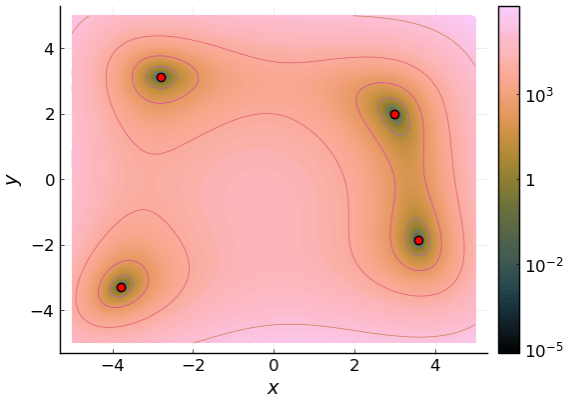
Definition B.9 (Himmelblau's Function). *The **Himmelblau's function**, symbolized as $f : \mathbb{R}^2 \to \mathbb{R}$, is formally described as:*

$$f(x,\, y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2 \tag{B.9}$$

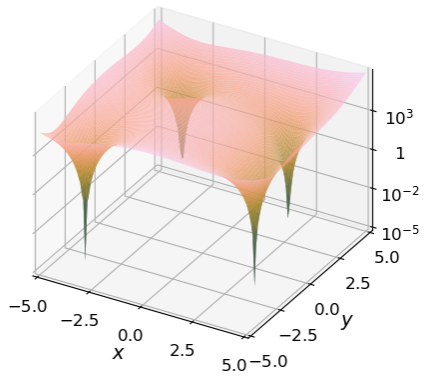
Here, $x,\, y \in \mathbb{R}$ are the decision variables, with the domains $\{x \mid -10 \leq x \leq 10\}$ and $\{y \mid -10 \leq y \leq 10\}$.

Distinguished by its four minima, located at $(3,\, 2)$, $(-2.805118,\, 3.131312)$, $(-3.779310,\, -3.283186)$, and $(3.584428,\, -1.848126)$, all roughly equating to zero, this function showcases its complexity.

Figure B.9 portrays the multi-modal landscape of Himmelblau's function through contour and surface plots, underlining its inherent intricacy, and as such, its utility in the evaluation of optimization techniques.



(a) Contour plot of Himmelblau's function, the minima are signified by the red dots



(b) Surface plot of Himmelblau's function

Figure B.9: The detailed multi-modal structure of Himmelblau's function illustrated through contour and surface plots

B.10 Hölder Table Function

The ***Hölder Table function*** is a two-dimensional real-valued function employed in various branches of mathematical analysis. Due to its intriguing properties and complex topography, it serves as a valuable tool for function approximation and numerical analysis studies.

Definition B.10 (Hölder Table function). *The **Hölder Table function**, designated as $f : \mathbb{R}^2 \to \mathbb{R}$, is described by the equation*

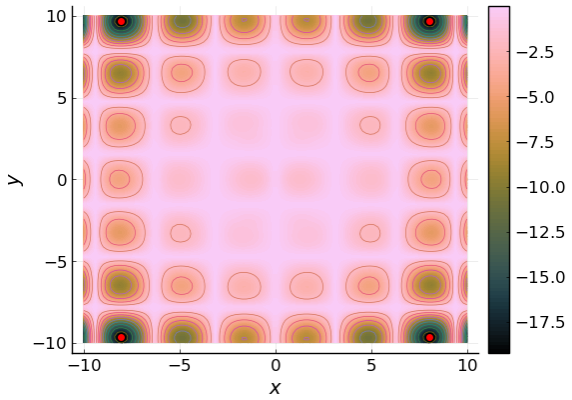
$$f(x,\, y) = -\left|\sin(x)\cos(y)\exp\left(\left|1 - \frac{\sqrt{x^2 + y^2}}{\pi}\right|\right)\right| \tag{B.10}$$

This function is defined for $-10 \leq x,\, y \leq 10$.

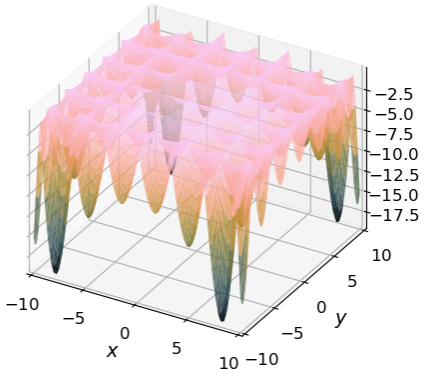
The Hölder Table function peaks globally at $f(x^*,\, y^*) = 19.208\,5$ for $x^* = \pm 8.05502$ and $y^* = \pm 9.66459$. Its intricately structured landscape can be vividly illustrated through contour and surface plots, as depicted in fig. B.10 on the following page.

B.11 Lévi Function N.13

The ***Lévi function N.13*** is a noteworthy two-dimensional function frequently employed in the field of optimization algorithms for performance testing. Its complex, sinuous landscape, teeming with numerous local minima, presents a significant challenge to optimization procedures.



(a) Contour visualization of the Hölder Table function. The global minimum points are denoted by red dots.



(b) Three-dimensional surface representation of the Hölder Table function.

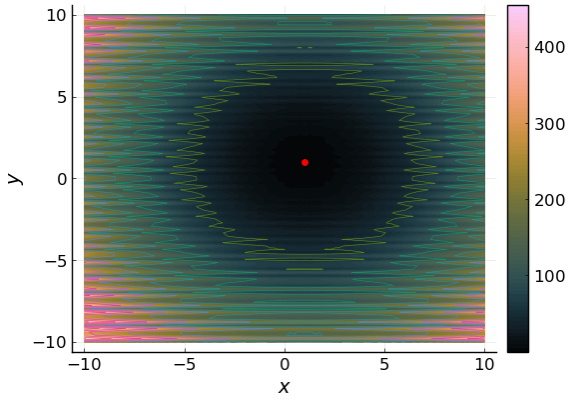
Figure B.10: The Complex Topography of the Hölder Table Function: Contour and Surface Representations

Definition B.11 (Lévi Function N.13). *The **Lévi function N.13**, denoted as $f : \mathbb{R}^2 \to \mathbb{R}$, is formally defined as follows:*

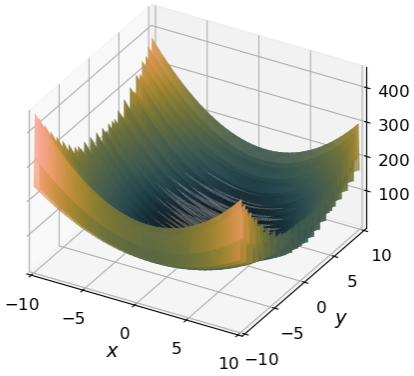
$$f(x, y) = \sin^2(3\pi x) + (x-1)^2 \cdot (1 + \sin^2(3\pi y)) + (y-1)^2 \cdot (1 + \sin^2(2\pi y)) \tag{B.11}$$

where $x, y \in \mathbb{R}$ are the decision variables.

The Lévi function N.13 finds its global minimum at $f(1, 1) = 0$. The complex topology of this function is visually captured in the contour and surface plots shown in Figure B.11.



(a) Contour plot of the Lévi function N.13



(b) Surface plot of the Lévi function N.13

Figure B.11: Contour and Surface Representations of the Lévi Function N.13

B.12 Matyas Function

The ***Matyas function***, known for its simplicity and convex nature, is a standard test problem in the field of optimization algorithms. Despite its apparent simplicity, it provides invaluable insights into an algorithm's performance and behavior.

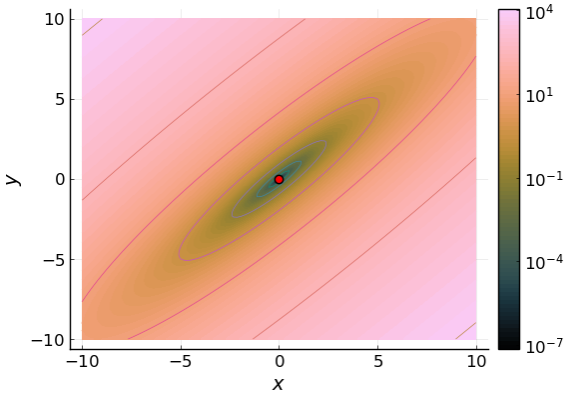
Definition B.12 (Matyas Function). *The **Matyas function**, denoted as $f : \mathbb{R}^2 \to \mathbb{R}$, is formulated as:*

$$f(x, y) = 0.26(x^2 + y^2) - 0.48xy \tag{B.12}$$

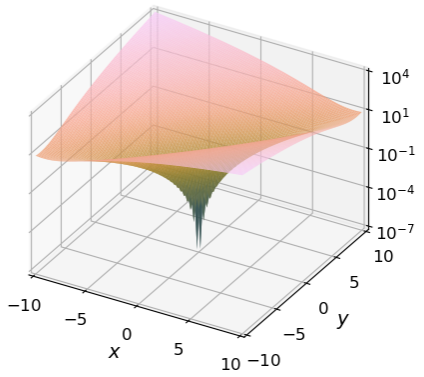
where:

- $x, y \in \mathbb{R}$ *denote the decision variables.*

The Matyas function reaches its global minimum at the origin, with $f(0, 0) = 0$. The contour and surface visualizations of the Matyas function, offering perspectives on its topographical attributes, are presented in fig. B.12.



(a) Contour plot of the Matyas function with a red dot indicating the global minimum



(b) Surface plot of the Matyas function

Figure B.12: Contour and Surface Visualizations of the Matyas Function

B.13 McCormick Function

The field of optimization research frequently employs the ***McCormick function*** as an insightful benchmark function. This function owes its name to the researcher, Garth P. McCormick, who first employed it in his seminal study on factorable nonconvex programs.[4] The McCormick function is renowned for its oscillatory properties and its complex landscape featuring a single global minimum amidst numerous local minima. This complex landscape can pose a significant challenge to optimization algorithms, as they risk being ensnared in the local minima.

Definition B.13 (McCormick Function). *The **McCormick Function**, symbolized as $f : \mathbb{R}^2 \to \mathbb{R}$, is expressed mathematically as follows:*

$$f(x, y) = \sin(x + y) + (x - y)^2 - 1.5x + 2.5y + 1 \tag{B.13}$$

The McCormick function is typically evaluated within the range $x, y \in [-1.5, 4]$ and $[-3, 4]$ respectively.

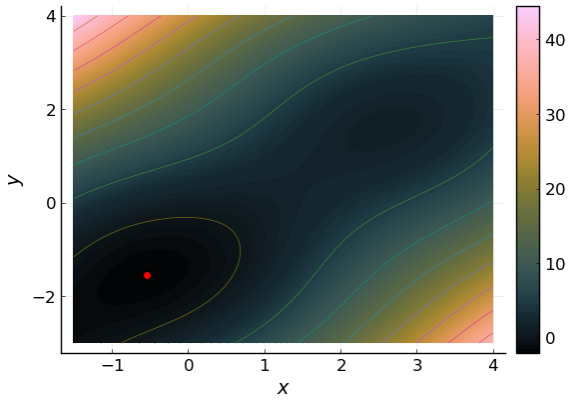
The global minimum of the McCormick function is found at $f(-0.54719, -1.54719) \approx -1.9133$. Visualizations of the McCormick function in the form of a contour plot and a surface plot are provided in fig. B.13 on the following page.

B.14 Rastrigin Function

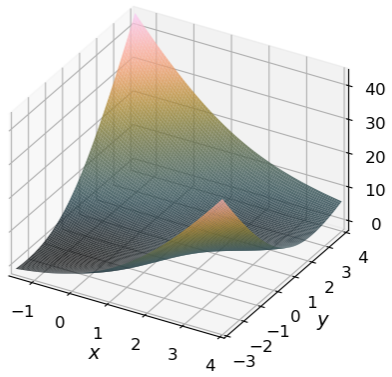
The Rastrigin function, first proposed by Rastrigin in 1974[5], is a prominent non-convex function utilized as a benchmark for optimization algorithms. This function is a classic example of non-linear multimodal optimization problems, known for their

[4] McCormick, Garth P. "Computability of Global Solutions to Factorable Nonconvex Programs: Part I — Convex Underestimating Problems." Mathematical Programming 10, no. 1 (December 1976): 147–75. https://doi.org/10.1007/BF01580665.

[5] Rastrigin, L. A. (1974). "Systems of extremal control." Mir, Moscow.



(a) Contour plot of the McCormick Function. The location of the global minimum is designated by a red dot.



(b) Surface plot of the McCormick Function

Figure B.13: Visual representations of the McCormick Function.

complexity due to the abundance of local minima.

Definition B.14 (Rastrigin Function)**.** *The **Rastrigin function**, denoted as $f : \mathbb{R}^n \to \mathbb{R}$, is formulated as:*

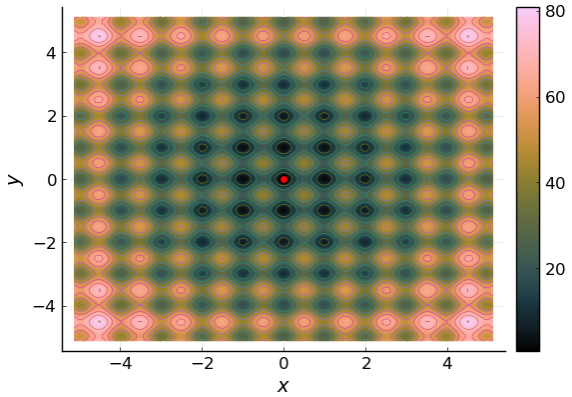
$$f(\mathbf{x}) = An + \sum_{i=1}^{n} \left[\mathbf{x}_i^2 - A\cos(2\pi\mathbf{x}_i)\right] \tag{B.14}$$

Here,

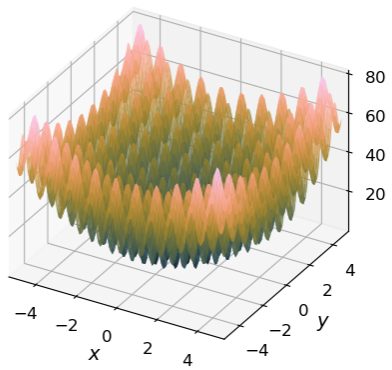
- *n is the number of dimensions.*
- *$\mathbf{x}$ is a vector composed of n real numbers.*
- *A is a constant, typically set to 10.*

The global minimum of the Rastrigin function is situated at $\mathbf{x}^* = (0, \ldots, 0)$, with $f(\mathbf{x}^*) = 0$. It's particularly challenging due to its numerous local minima, distributed regularly throughout the search space. The function is predominantly evaluated within the hypercube $\mathbf{x} \in [-5.12,\ 5.12]^n$.

Visualizations of the Rastrigin function for $n = 2$ are presented in fig. B.14, showcasing both the contour plot and the 3D surface plot.



(a) Contour plot of the Rastrigin Function.



(b) 3D surface plot of the Rastrigin Function.

Figure B.14: Visualizations of the Rastrigin Function for $n = 2$, with the global minimum marked by a red dot.

B.15 The Rosenbrock Function

Introduced by Howard H. Rosenbrock in 1960[6], the Rosenbrock function is a non-convex function that serves as a benchmark for optimization algorithms. Despite its simplicity, it poses a challenge due to its characteristic landscape, a long, narrow, parabolic-shaped flat valley. While finding the valley is relatively simple, converging to the global minimum within it is notably difficult.

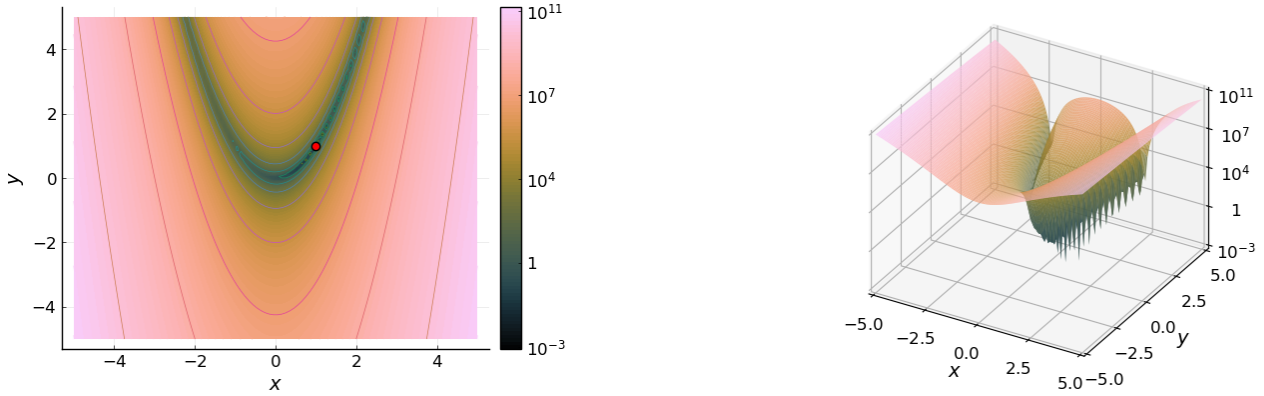
Definition B.15 (Rosenbrock function)**.** *The **Rosenbrock function**, denoted as $f : \mathbb{R}^n \to \mathbb{R}$, is defined by the following equation:*

$$f(\mathbf{x}) = \sum_{i=1}^{n-1} \left[100(\mathbf{x}_{i+1} - \mathbf{x}_i^2)^2 + (1 - \mathbf{x}_i)^2\right] \tag{B.15}$$

In this equation:

- *n is the number of dimensions.*
- *$\mathbf{x}$ is a vector composed of n real numbers.*

The Rosenbrock function's global minimum is at $f(\mathbf{x}^*) = 0$, corresponding to the point $\mathbf{x}^* = (1, \ldots, 1)$. To illustrate the function's behavior for $n = 2$, fig. B.15 presents a contour plot and a surface plot.



(a) Contour plot of the Rosenbrock function.

(b) Surface plot of the Rosenbrock function.

Figure B.15: Rosenbrock Function for $n = 2$, the red dot indicates the global minimum.

B.16 Schaffer Function N.2

The ***Schaffer Function N.2*** is a well-established mathematical function frequently used in the realm of algorithm testing. It serves as a performance benchmark for a wide array of optimization algorithms, particularly those grounded in evolutionary computation and swarm intelligence principles.

Definition B.16 (Schaffer Function N.2)**.** *The Schaffer Function N.2 is defined over a two-dimensional domain, mapping $\mathbb{R}^2$ to $\mathbb{R}$. The function, denoted as $f(\mathbf{x})$, is expressed as follows:*

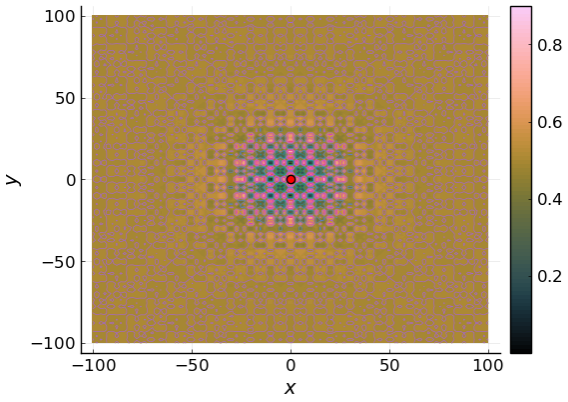
$$f(\mathbf{x}) = 0.5 + \frac{\sin^2(\sqrt{x_1^2 + x_2^2}) - 0.5}{(1.0 + 0.001 \cdot (x_1^2 + x_2^2))^2}$$

Here, $\mathbf{x} = (x_1, x_2)$ represents a point in the two-dimensional domain, with x_1, x_2 each ranging within the interval [-100, 100].

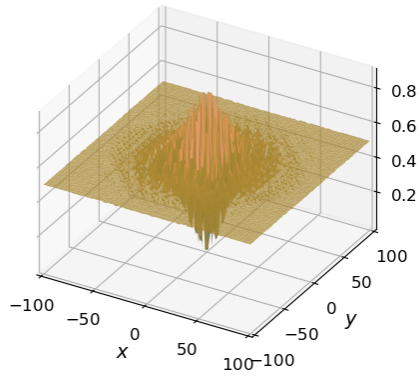
[6]Rosenbrock, H.H. (1960). "An automatic method for finding the greatest or least value of a function". The Computer Journal. 3 (3): 175-184. doi:10.1093/comjnl/3.3.175. ISSN 0010-4620.

The Schaffer Function N.2 achieves its global minimum at $f(0,\,0) = 0$. A visual exploration of this function can be enhanced by both contour and surface plots, providing a clearer understanding of its global minimum and topological characteristics.

fig. B.17 illustrate the *Schaffer Function N.2*.

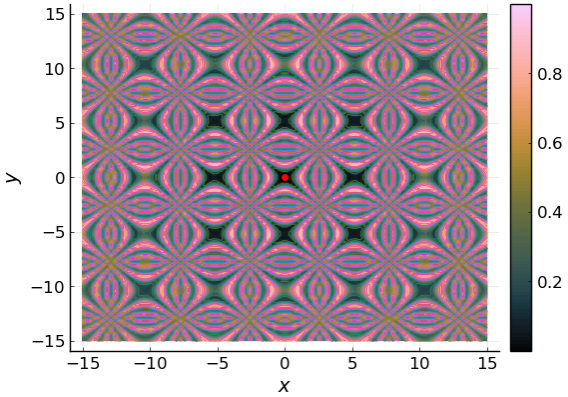


(a) Contour plot of the Schaffer Function N.2

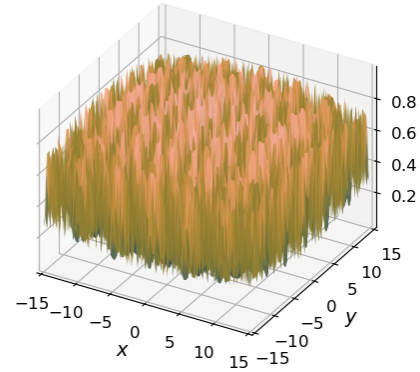


(b) Surface plot of the Schaffer Function N.2

Figure B.16: a



(a) Close-up contour plot of the Schaffer Function N.2 in the vicinity of the global minimum.



(b) Close-up surface plot of the Schaffer Function N.2 in the vicinity of the global minimum.

Figure B.17: Visualization of the Schaffer Function N.2. The contour and surface plots illustrate the function's topology, with the global minimum denoted by a red dot. The close-up contour and surface plots provide a more precise view of the global minimum and the immediate surroundings.

B.17 Schaffer Function N.4

The ***Schaffer Function N.4*** is a mathematical function used as a performance test problem for optimization algorithms. It is particularly suited for testing algorithms that need to optimize complex, non-linear problems with many local minima.

Definition B.17 (Schaffer Function N.4)**.** *Schaffer Function N.4 is defined as follows:*
For $x \in \mathbb{R}^2$, the function is given by:

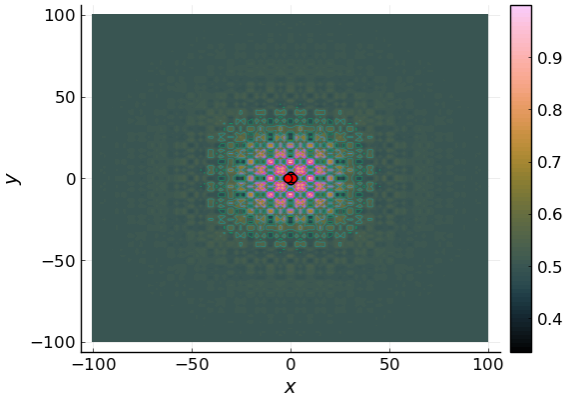
$$f(x) = 0.5 + \frac{\cos^2\left(\sin\left[\left|x^2 - y^2\right|\right]\right) - 0.5}{\left[1 + 0.001\left(x^2 + y^2\right)\right]^2}$$

Where:

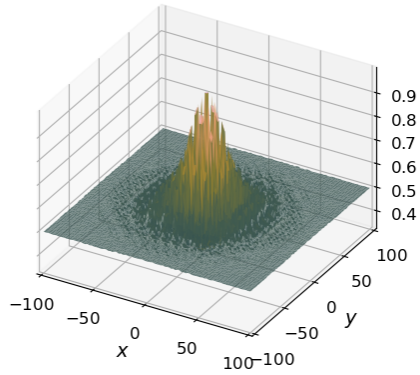
- *x and y are decision variables.*
- *The function has four global minima of 0.292579, located at $0,\ \pm 1.25313$ and $\pm 1.25313,\ 0$.*

The characteristic feature of Schaffer N.4 is its landscape with several local minima and four global minima. This structure poses a significant challenge to optimization algorithms, especially those prone to being trapped in local minima.

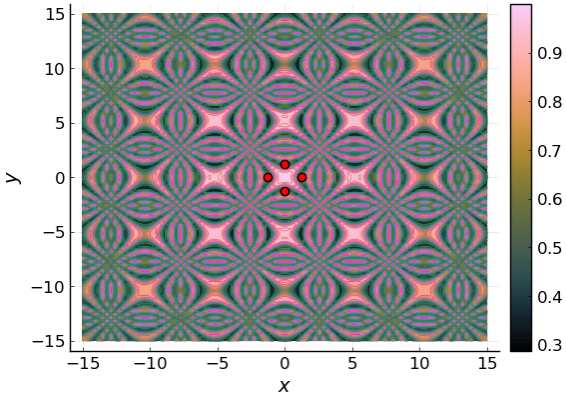
Figure B.18 provides a visualization of the Schaffer Function N.4. The contour and surface plots show the function's topology, with the red dots indicating the locations of the global minima. The close-up contour and surface plots offer a detailed view of the global minima and their immediate surroundings, highlighting the intricacy of the function's landscape.



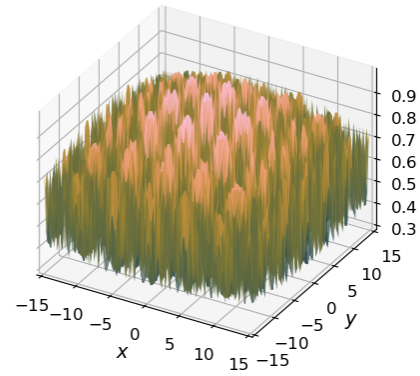
(a) Contour plot of the Schaffer Function N.4



(b) Surface plot of the Schaffer Function N.4



(c) Close-up contour plot of the Schaffer Function N.4 in the vicinity of the global minimum.



(d) Close-up surface plot of the Schaffer Function N.4 in the vicinity of the global minimum.

Figure B.18: Visualization of the Schaffer Function N.4. The contour and surface plots illustrate the function's topology, with the global minimum denoted by a red dot. The close-up contour and surface plots provide a more precise view of the global minimum and the immediate surroundings.

B.18 The Sphere Function

The sphere function serves as a prominent benchmark problem in the realm of optimization algorithms. This convex function's simplicity and well-defined nature make it ideal for gauging the performance of such algorithms.

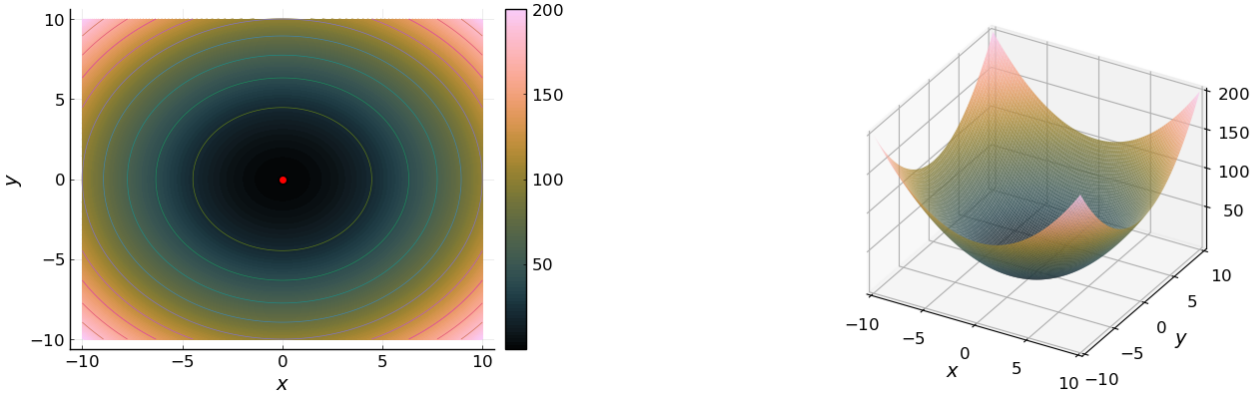
Definition B.18 (Sphere Function)**.** *The **sphere function**, denoted as $f : \mathbb{R}^n \to \mathbb{R}$, is formally described as:*

$$f(\mathbf{x}) = \sum_{i=1}^{n} \mathbf{x}_i^2 \tag{B.16}$$

Herein:

- $n \in \mathbb{N}$ *signifies the number of dimensions.*
- $\mathbf{x}_i \in \mathbb{R}$ *represents the i-th element of the vector* $\mathbf{x}$*.*

The global minimum of the sphere function occurs at $f(\mathbf{x}^*) = 0$ with the input vector $\mathbf{x}^* = (0, \ldots, 0)$. The contour and surface plots showcasing the behavior of the sphere function for a two-dimensional input ($n = 2$) are illustrated in fig. B.19.



(a) Contour plot of the Sphere Function.

(b) Surface plot of the Sphere Function.

Figure B.19: Visual Representation of the Sphere Function with $n = 2$

B.19 Styblinski-Tang function

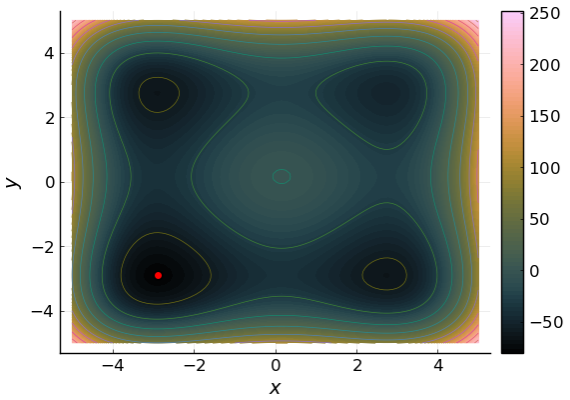
The ***Styblinski-Tang function*** is a non-convex function used as a performance test problem for optimization algorithms. This function is known for its complex and intricate landscape, presenting numerous local minima that pose challenges to optimization algorithms.

Definition B.19 (Styblinski-Tang function)**.** *The Styblinski-Tang function is defined in two dimensions, with the variables x and y.*

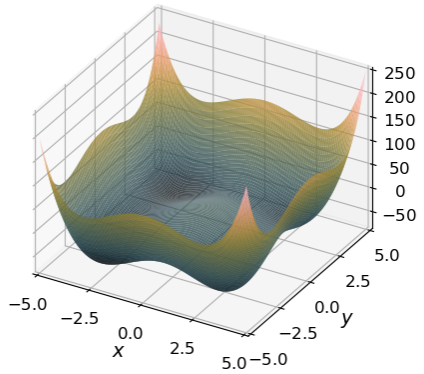
$$f(\mathbf{x}) = \frac{1}{2} \sum_{i=1}^{n} \left(\mathbf{x}_i^4 - 16\mathbf{x}_i^2 + 5\mathbf{x}_i \right) \tag{B.17}$$

where x and y are any real numbers.

The intriguing properties of this function have made it a common benchmark in the study of algorithm performance, particularly in the field of evolutionary computation and swarm intelligence.



(a) Contour plot of the Styblinski-Tang function. The intricate patterns show the complex landscape of the function. The red dot signifies the global minimum.



(b) Three-dimensional surface plot of the Styblinski-Tang function. The plot illustrates the numerous local minima and the complex topography of the function.

Figure B.20: Visualizations of the Styblinski-Tang function.

B.20 Three-Hump Camel Function

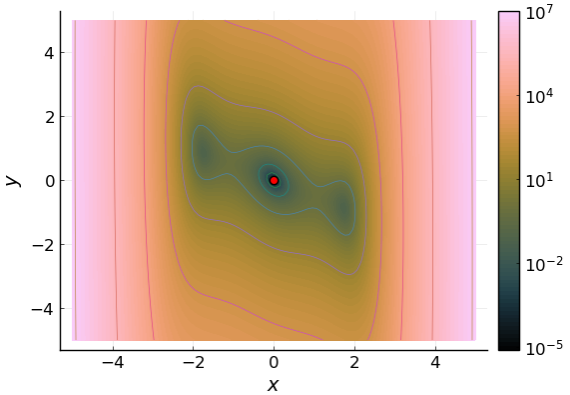
The ***Three-Hump Camel function***, colloquially termed as the camel-back function, serves as a standard benchmark in the optimization algorithms testing landscape. This two-dimensional function earns its name from the characteristic tri-modal hump visual pattern it presents in a three-dimensional space, bearing resemblance to a camel's back.

Definition B.20 (Three-Hump Camel Function)**.** *The* ***Three-Hump Camel function****, depicted as* $f : \mathbb{R}^2 \to \mathbb{R}$*, is formally expressed as:*

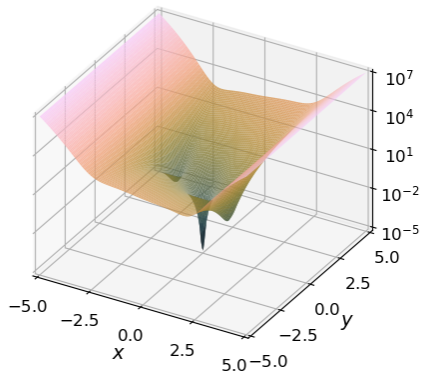
$$f(x, y) = 2x^2 - 1.05x^4 + \frac{x^6}{6} + xy + y^2 \tag{B.18}$$

where $-5 \leq x \leq 5$ *and* $-5 \leq y \leq 5$*.*

The Three-Hump Camel function exhibits its global minimum at the origin, $f(0, 0) = 0$. Although its form appears straightforward, the function's multiple local optima pose considerable challenges for optimization algorithms, making it an excellent test case.



(a) Contour plot of the Three-Hump Camel function. The red dot signifies the global minimum.



(b) Surface plot showcasing the characteristic tri-modal humps of the Three-Hump Camel function.

Figure B.21: Contour and surface visualizations of the Three-Hump Camel function

Appendix C

Additional Listings

This appendix contains additional listings of the source code used in this thesis that are not essential to the understanding of the thesis. The listings are included here for completeness.

C.1 Keen

Listing C.1– Minimal implementation of the *One Max* problem using the *Keen* framework.

```
Domain.random = Random(11)
val engine =
    evolutionEngine(
        { genotype: Genotype<Boolean, BooleanGene> -> genotype.flatten().count { it }.
        ↪  toDouble() },
        genotypeOf {
            chromosomeOf {
                booleans {
                    size = 20
                    trueRate = 0.15
                }
            }
        }) {
        populationSize = 20
        alterers += listOf(
            BitFlipMutator(individualRate = 0.5), SinglePointCrossover(chromosomeRate = 0.6
            ↪  )
        )
        limits += TargetFitness(TARGET_FITNESS)
    }
engine.evolve()
engine.listeners.forEach { it.display() }
```

Listing C.2– *Room Scheduling* problem using the *Keen* framework.

```
private data class Meeting(val start: Int, val end: Int)

private val meetings =
    listOf(
```

```
        Meeting(start = 1, end = 3),
        Meeting(start = 2, end = 3),
        Meeting(start = 5, end = 6),
        Meeting(start = 7, end = 9),
        Meeting(start = 4, end = 7),
        Meeting(start = 8, end = 10),
        Meeting(start = 2, end = 7),
        Meeting(start = 3, end = 4),
        Meeting(start = 1, end = 5),
        Meeting(start = 3, end = 6),
        Meeting(start = 4, end = 5)
    )

private fun fitnessFunction(genotype: Genotype<Int, IntGene>): Double {
    // We can access the genotype components by index as it is a matrix.
    val rooms = meetings.groupBy { genotype[meetings.indexOf(it)][0].value }
    val conflicts = rooms.values.sumOf { meetingList ->
        val table = IntArray(size = 10)
        meetingList.forEach { meeting ->
            // The ..< operator is equivalent to the range: [start, end)
            for (i in meeting.start..<meeting.end) {
                table[i]++
            }
        }
        table.count { it > 1 }
    }
    // Fitness is penalized by the number of conflicts.
    return rooms.size.toDouble() + conflicts
}

private const val POPULATION_SIZE = 100

fun main() {
    Domain.random = Random(420)
    val summary = EvolutionSummary<Int, IntGene>()
    val plotter = EvolutionPlotter<Int, IntGene>()
    val engine = evolutionEngine(
        ::fitnessFunction,
        genotypeOf {
            repeat(meetings.size) {
                chromosomeOf {
                    ints {
                        size = 1
                        ranges  += meetings.indices.first..meetings.indices.last
                    }
                }
            }
        }) {
        populationSize = POPULATION_SIZE
        ranker = FitnessMinRanker()
        alterers += listOf(
          RandomMutator(individualRate = 0.06), SinglePointCrossover(chromosomeRate = 0.2)
```

```
        )
        limits += listOf(
          SteadyGenerations(generations = 20), GenerationLimit(generations = 100)
        )
        listeners += plotter + summary  // Add both listeners to the engine
    }
    engine.evolve()
    summary.display()
    val schedule = MutableList(meetings.size) { mutableListOf<Meeting>() }
    meetings.forEachIndexed { index, meeting ->
        val room = summary.fittest.genotype[index][0].value
        schedule[room] += meeting
    }
    schedule.forEachIndexed { index, meetings ->
        println("Room $index: $meetings")
    }
    plotter.display()
}
```

The above code will produce the following output:

```
------------ Evolution Summary ---------------
|--> Initialization time: 43 ms
------------- Evaluation Times ----------------
|--> Average: 0.5 ms
|--> Max: 10 ms
|--> Min: 0 ms
-------------- Selection Times ----------------
|   |--> Offspring Selection
|   |   |--> Average: 0.15625 ms
|   |   |--> Max: 5 ms
|   |   |--> Min: 0 ms
|   |--> Survivor Selection
|   |   |--> Average: 0.0 ms
|   |   |--> Max: 0 ms
|   |   |--> Min: 0 ms
--------------- Alteration Times --------------
|--> Average: 1.4375 ms
|--> Max: 17 ms
|--> Min: 0 ms
-------------- Evolution Results --------------
|--> Total time: 222 ms
|--> Average generation time: 5.53125 ms
|--> Max generation time: 99 ms
|--> Min generation time: 1 ms
|--> Generation: 32
|--> Steady generations: 21
|--> Fittest: Genotype(chromosomes=[[7], [5], [10], [7], [7], [5], [4], [7], [10], [0], [5]])
|--> Best fitness: 5.0
Room 0: [Meeting(start=3, end=6)]
Room 1: []
Room 2: []
Room 3: []
Room 4: [Meeting(start=2, end=7)]
```

```
Room 5: [Meeting(start=2, end=3), Meeting(start=8, end=10), Meeting(start=4, end=5)]
Room 6: []
Room 7: [Meeting(start=1, end=3), Meeting(start=7, end=9), Meeting(start=4, end=7),
↪  Meeting(start=3, end=4)]
Room 8: []
Room 9: []
Room 10: [Meeting(start=5, end=6), Meeting(start=1, end=5)]
```

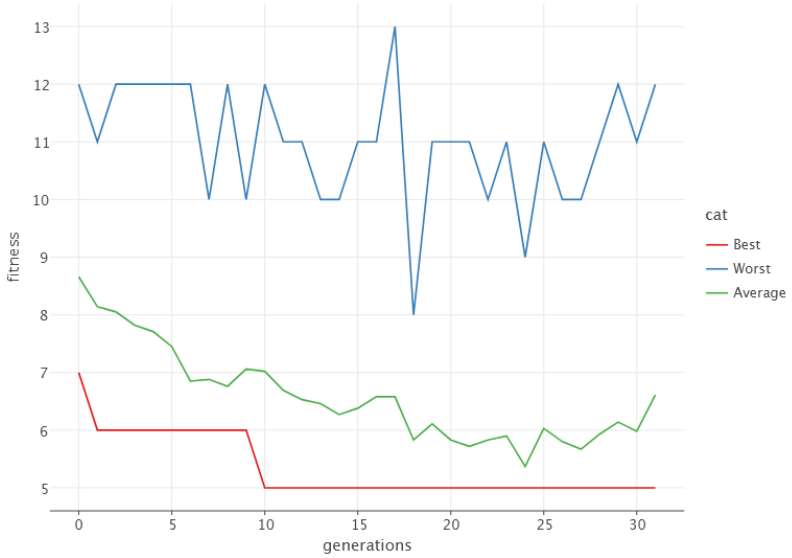


Figure C.1: Evolution plot for the *Room Scheduling* problem.

Listing C.3– Example of using the Fun class to implement the addition operation

```
// Using the Fun class
var add = Fun<Double>("+", 2) { it[0] + it[1] }
// Extending the Fun class
class Add : Fun<Double>("+", 2, { it[0] + it[1] })
var add = Add()
// As an anonymous object
var add = object : Fun<Double>("+", 2, { it[0] + it[1] }) {}
```

C.2 Crash Reproduction

Listing C.4– Functions utilized as the search space for the *Tracer* tool.

```
fun one(): Int = 1
fun two(): Int = 2
fun zero(): Int = 0
fun negativeInt(): Int = -1
fun throwException0(): Nothing =
    throw IllegalArgumentException("This function always throws an exception!")
fun compareWithOneHundred(x: Int): Nothing = if (x < 100) {
    if (x > 50) throwIAE("The number is greater than 50")
    else throw AssertionError("The number is less than 50")
```

```
} else {
    throw IllegalArgumentException("The number is greater than 100")
}
private fun throwIAE(s: String): Nothing {
    throw IllegalArgumentException(s)
}
fun addNumbers(x: Int, y: Int) = x + y
fun multiplyNumbers(x: Int, y: Int) = x * y
```

The code snippet presents a collection of functions utilized in the *Tracer* tool's genetic algorithm as the search space.

Listing C.5– The `Tracer` tool implementation.

```
class Tracer<T : Throwable>(
    private val functions: List<KFunction<*>>,
    private val targetException: KClass<T>,
    private val targetMessage: String,
    private val targetFunction: String,
    private val listeners: List<EvolutionListener<KFunction<*>, KFunctionGene>>,
) {
    val engine = evolutionEngine(
        ::fitnessFunction,
        genotype {
            chromosomeOf {
                KFunctionChromosome.Factory(10) {
                    KFunctionGene(functions.random(Domain.random), functions)
                }
            }
        }
    ) {
        this.populationSize = 1000
        this.alterers += listOf(RandomMutator(0.2), SinglePointCrossover(0.3))
        this.limits += listOf(TargetFitness(5.0), GenerationLimit(1000))
        this.listeners += this@Tracer.listeners
    }

    fun run(): MinimalCrashReproduction {
        val result = engine.evolve()
        val program = minimize(result).population
            .groupBy { it.fitness }
            .maxByOrNull { it.key }?.value
            ?.minByOrNull { it.flatMap().size }
        return MinimalCrashReproduction(program!!)
    }

    private fun minimize(result: EvolutionState<KFunction<*>, KFunctionGene>) =
        result.trimStart().trimEnd()

    private fun EvolutionState<KFunction<*>, KFunctionGene>.trimEnd() = map {
        // Iteratively remove the last instruction from the genotype
        for (i in 1..it.genotype.first().size) {
            val candidate = Genotype(KFunctionChromosome(it.genotype.first().take(i)))
            val candidateFitness = fitnessFunction(candidate)
```

```
            // Replace genotype with the minimized one if fitness is maintained or improved
            if (candidateFitness >= it.fitness) {
                return@map it.copy(genotype = candidate, fitness = candidateFitness)
            }
        }
        it
    }


    private fun EvolutionState<KFunction<*>, KFunctionGene>.trimStart() = map {
        // Iteratively remove the first instruction from the genotype
        for (i in it.genotype.first().size downTo 1) {
            val candidate = Genotype(KFunctionChromosome(it.genotype.first().drop(i)))
            val candidateFitness = fitnessFunction(candidate)
            // Replace genotype with the minimized one if fitness is maintained or improved
            if (candidateFitness >= it.fitness) {
                return@map it.copy(genotype = candidate, fitness = candidateFitness)
            }
        }
        it
    }

    private fun fitnessFunction(genotype: Genotype<KFunction<*>, KFunctionGene>) =
        genotype.first().let { functions ->
            try {
                val variables = mutableListOf<Any?>()
                functions.forEach { f ->
                    variables += f(variables.takeLast(f.arity))
                }
                0.0
            } catch (invocationTargetException: InvocationTargetException) {
                var fitness = 0.0
                val ex = invocationTargetException.targetException
                val stack = ex.stackTrace
                if (targetException == ex::class) {
                    fitness += 2.0
                    if (targetMessage.isEmpty() || targetMessage == ex.message) {
                        fitness++
                    }
                }
                if (targetFunction.isEmpty()
                    || stack.any { it.methodName == targetFunction }) {
                    fitness += 2.0
                }
                fitness
            } catch (e: Exception) {
                0.0
            }
        }

    companion object {
        inline fun <reified E : Throwable> create(
```

```
            functions: List<KFunction<*>>,
            targetMessage: String = "",
            targetFunction: String = "",
            listeners: List<EvolutionListener<KFunction<*>, KFunctionGene>> = emptyList()
        ) = Tracer(functions, E::class, targetMessage, targetFunction, listeners)
    }
}
```

C.3 Others

Listing C.6– Calculation of $|\mathbb{T}_{\leq 5}(\mathcal{T}, \mathcal{F})|$ for $\mathcal{T} = \{x, c\}$ and the set $\mathcal{F} = \{+, -, \times, /, \sin, \cos, \text{pow}\}$ using the *Julia* programming language.

```
arities = [2, 2, 2, 2, 1, 1, 2] # A({+, −, ×, /, sin, cos, pow}) = {2, 2, 2, 2, 1, 1, 2}
terminals_size = 8  # |T| = |{x, 1, 2, 3, 4, 5, 6, 7}| = 8
# |T_{≤h}|
t_leq(h::Int)::Int128 = if h == 0 # |T| if h = 0
terminals_size
else  # (Σ_{h=0}^{h-1} Σ_{f∈F} |T_h|^{A(f)}) + |T| if h > 0
c_sum = terminals_size
for i = 0:h - 1
    c_sum = c_sum + sum(t(i) .^ arities)
end
c_sum
end
# |T_h|
t(h::Int)::Int128 = if h == 0 # |T| if h = 0
terminals_size
else  # Σ_{f∈F} |T_{h-1}|^{A(f)} if h > 0
sum(t(h - 1) .^ arities)
end
res = t_leq(5)  # |T_{≤5}(T, F)|
```

Listing C.7– "Desugared" version of the code presented in listing 4.26 on page 67

```
constraints(() -> {
    invoke("The value of x must be greater than or equal to 10", () -> {
        must(x, BeAtLeast(10))
    })
    invoke("The value of y must not be negative",
        () -> CustomConstraintException("The value of y must not be negative")
    () -> {
        mustNot(y, BeNegative)
    })
})
```

Listing C.8– Demonstration of the use of *Strait-Jakt*'s `constraint` DSL to validate preconditions inside the *Subtree Crossover* operator.

```
private fun enforcePreconditions(chromosomes: List<Chromosome<T, G>>) = constraints {
    "The crossover operator requires two chromosomes." { chromosomes must HaveSize(2) }
    chromosomes.forEach { chromosome ->
        "The parents must have the same size" {
            chromosome must HaveSize(chromosomes[0].size)
        }
        chromosome.forEach { gene ->
            ("The gene's arity (${gene.value.arity}) must match "
                + "the gene's children (${gene.value.children.size})") {
                gene.value.children must HaveSize(gene.value.arity)
            }
        }
    }
}
```

In listing C.8 we can see an example of the use of the *StraitJakt* library to validate the preconditions of the *Subtree Crossover* operator. The `constraints` DSL allows us to define a set of constraints that must be satisfied for the operation to be executed. In this case, we are validating that the operator receives two chromosomes with the same size and that each gene's arity matches the number of children it has. If any of these constraints is not satisfied, the *StraitJakt* library will throw a *composite exception* containing all the errors found. This is done by accumulating the exceptions instead of throwing them immediately. This approach allows us to validate all the constraints at once, instead of throwing an exception for each error found.

Appendix D

Keen Open Source License: BSD 2-Clause License

Bibliography

[1] A. M. Turing. "I.—COMPUTING MACHINERY AND INTELLIGENCE". In: ***Mind*** LIX.236 (Oct. 1, 1950), pp. 433–460. ISSN: 1460-2113, 0026-4423. DOI: 10.1093/mind/LIX.236.433. URL: https://academic.oup.com/mind/article/LIX/236/433/986238 (visited on 06/10/2023).

[2] H. H. Rosenbrock. "An Automatic Method for Finding the Greatest or Least Value of a Function". In: ***The Computer Journal*** 3.3 (Mar. 1, 1960), pp. 175–184. ISSN: 0010-4620, 1460-2067. DOI: 10.1093/comjnl/3.3.175. URL: https://academic.oup.com/comjnl/article-lookup/doi/10.1093/comjnl/3.3.175 (visited on 07/04/2023).

[3] Nils Aall Barricelli. "Numerical Testing of Evolution Theories: Part I Theoretical Introduction and Basic Tests". In: ***Acta Biotheoretica*** 16.1-2 (Mar. 1962), pp. 69–98. ISSN: 0001-5342, 1572-8358. DOI: 10.1007/BF01556771. URL: http://link.springer.com/10.1007/BF01556771 (visited on 06/10/2023).

[4] Sartaj Sahni. "Approximate Algorithms for the 0/1 Knapsack Problem". In: ***Journal of the ACM (JACM)*** 22.1 (1975), pp. 115–124.

[5] David E Goldberg and Robert Lingle. "Alleles, Loci, and the Traveling Salesman Problem". In: ***Proceedings of an international conference on genetic algorithms and their applications*** (1985), pp. 154–159.

[6] John H. Holland. ***Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence***. 1st MIT Press ed. Complex Adaptive Systems. Cambridge, Mass: MIT Press, 1992. 211 pp. ISBN: 978-0-262-08213-6 978-0-262-58111-0.

[7] John R. Koza. ***Genetic Programming. 1: On the Programming of Computers by Means of Natural Selection***. Complex Adaptive Systems. Cambridge, Mass.: MIT Press, 1992. 819 pp. ISBN: 978-0-262-52791-0.

[8] Peter J Angeline and Jordan B Pollack. "The Evolutionary Induction of Subroutines". In: ***Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society***. Bloomington, Indiana, 1992, pp. 236–241.

[9] John R. Koza. ***Genetic Programming II: Automatic Discovery of Reusable Programs***. Complex Adaptive Systems. Cambridge, Mass: MIT Press, 1994. 746 pp. ISBN: 978-0-262-11189-8.

[10] Peter J. Angeline. "Genetic Programming and Emergent Intelligence". In: ***Advances in Genetic Programming***. Advances in Genetic Programming 1. Cambridge (Mass.) London: MIT press, 1994, pp. 75–97. ISBN: 978-0-262-11188-1.

[11] J.P. Rosca and D.H. Ballard. "Learning by Adapting Representations in Genetic Programming". In: ***Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence***. First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence. Orlando, FL, USA: IEEE, 1994, pp. 407–412. ISBN: 978-0-7803-1899-1. DOI: 10.1109/ICEC.1994.349916. URL: http://ieeexplore.ieee.org/document/349916/ (visited on 06/23/2023).

[12] Tobias Blickle and Lothar Thiele. "A Mathematical Analysis of Tournament Selection." In: ***ICGA*** 95 (1995), pp. 9–15.

[13] Shigeyoshi Tsutsui, Masayuki Yamamura, and T. Higuchi. "Multi-Parent Recombination with Simplex Crossover in Real-Coded Genetic Algorithms". In: ***Proceedings of Genetic and Evolutionary Computation Conference*** (Jan. 1, 1999).

[14] R. Andonov, V. Poirriez, and S. Rajopadhye. "Unbounded Knapsack Problem: Dynamic Programming Revisited". In: ***European Journal of Operational Research*** 123.2 (June 2000), pp. 394–407. ISSN: 03772217. DOI: 10.1016/S0377-2217(99)00265-9. URL: https://linkinghub.elsevier.com/retrieve/pii/S0377221799002659 (visited on 08/09/2023).

[15] I. Kushchu. "Genetic Programming and Evolutionary Generalization". In: ***IEEE Transactions on Evolutionary Computation*** 6.5 (Oct. 2002), pp. 431–442. ISSN: 1941-0026. DOI: 10.1109/TEVC.2002.805038.

[16] W. B. Langdon and Riccardo Poli. ***Foundations of Genetic Programming***. Berlin ; New York: Springer, 2002. 260 pp. ISBN: 978-3-540-42451-2.

[17] Kalyanmoy Deb, Pawan Zope, and Abhishek Jain. "Distributed Computing of Pareto-Optimal Solutions with Evolutionary Algorithms". In: ***Evolutionary Multi-Criterion Optimization***. Ed. by Carlos M. Fonseca et al. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2003, pp. 534–549. ISBN: 978-3-540-36970-7. DOI: 10.1007/3-540-36970-8_38.

[18] Kenneth A. De Jong. ***Evolutionary Computation: A Unified Approach***. 2006. ISBN: 978-0-262-25598-1. URL: https://ieeexplore.ieee.org/servlet/opac?bknumber=6267245.

[19] ***The MIT License***. Open Source Initiative. Oct. 31, 2006. URL: https://opensource.org/license/mit/ (visited on 07/26/2023).

[20] Carlos Pacheco and Michael D. Ernst. "Randoop: Feedback-Directed Random Testing for Java". In: ***Companion to the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion***. OOPSLA '07. New York, NY, USA: Association for Computing Machinery, Oct. 20, 2007, pp. 815–816. ISBN: 978-1-59593-865-7. DOI: 10.1145/1297846.1297902. URL: https://dl.acm.org/doi/10.1145/1297846.1297902 (visited on 12/07/2023).

[21] Dipankar Dasgupta et al. "A Comparison of Multiobjective Evolutionary Algorithms with Informed Initialization and Kuhn-Munkres Algorithm for the Sailor Assignment Problem". In: ***Proceedings of the 10th Annual Conference Companion on Genetic and Evolutionary Computation***. GECCO '08. New York, NY, USA: Association for Computing Machinery, July 12, 2008, pp. 2129–2134. ISBN: 978-1-60558-131-6. DOI: 10.1145/1388969.1389035. URL: https://doi.org/10.1145/1388969.1389035 (visited on 08/13/2023).

[22] Riccardo Poli et al. ***A Field Guide to Genetic Programming***. [Morrisville, NC: Lulu Press], 2008. 233 pp. ISBN: 978-1-4092-0073-4.

[23] Lucas de P. Veronese and Renato A. Krohling. "Differential Evolution Algorithm on the GPU with C-CUDA". In: ***IEEE Congress on Evolutionary Computation***. IEEE Congress on Evolutionary Computation. July 2010, pp. 1–7. DOI: 10.1109/CEC.2010.5586219.

[24] Travis Desell et al. "An Analysis of Massively Distributed Evolutionary Algorithms". In: ***IEEE Congress on Evolutionary Computation***. IEEE Congress on Evolutionary Computation. July 2010, pp. 1–8. DOI: 10.1109/CEC.2010.5586073.

[25] Thé Van Luong, Nouredine Melab, and El-Ghazali Talbi. "GPU-based Island Model for Evolutionary Algorithms". In: ***Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation***. GECCO '10. New York, NY, USA: Association for Computing Machinery, July 7, 2010, pp. 1089–1096. ISBN: 978-1-4503-0072-8. DOI: 10.1145/1830483.1830685. URL: https://doi.org/10.1145/1830483.1830685 (visited on 08/08/2023).

[26] Thé Van Luong, Nouredine Melab, and El-Ghazali Talbi. "Parallel Hybrid Evolutionary Algorithms on GPU". In: ***IEEE Congress on Evolutionary Computation***. IEEE Congress on Evolutionary Computation. July 2010, pp. 1–8. DOI: 10.1109/CEC.2010.5586403.

[27] Xinjie Yu and Mitsuo Gen. ***Introduction to Evolutionary Algorithms***. Decision Engineering. London ; New York: Springer, 2010. 418 pp. ISBN: 978-1-84996-128-8 978-1-84996-129-5.

[28] Saber M. Elsayed, Ruhul A. Sarker, and Daryl L. Essam. "GA with a New Multi-Parent Crossover for Solving IEEE-CEC2011 Competition Problems". In: ***2011 IEEE Congress of Evolutionary Computation (CEC)***. 2011 IEEE Congress of Evolutionary Computation (CEC). June 2011, pp. 1034–1040. DOI: 10.1109/CEC.2011.5949731.

[29] Gordon Fraser and Andrea Arcuri. "EvoSuite: Automatic Test Suite Generation for Object-Oriented Software". In: ***Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering***. ESEC/FSE '11. New York, NY, USA: Association for Computing Machinery, Sept. 9, 2011, pp. 416–419. ISBN: 978-1-4503-0443-6. DOI: 10.1145/2025113.2025179. URL: https://dl.acm.org/doi/10.1145/2025113.2025179 (visited on 12/07/2023).

[30] Otman Abdoun, Jaafar Abouchabaka, and Chakir Tajani. ***Analyzing the Performance of Mutation Operators to Solve the Travelling Salesman Problem***. Mar. 14, 2012. DOI: 10.48550/arXiv.1203.3099. arXiv: 1203.3099 [cs]. URL: http://arxiv.org/abs/1203.3099 (visited on 08/22/2023). preprint.

[31] Daniel Shiffman. ***The Nature of Code***. Version 1.0, generated December 6, 2012. s.l.: Selbstverl., 2012. 498 pp. ISBN: 978-0-9859308-0-6.

[32] Mateusz Guzek et al. "Multi-Objective Evolutionary Algorithms for Energy-Aware Scheduling on Distributed Computing Systems". In: ***Applied Soft Computing*** 24 (Nov. 1, 2014), pp. 432–446. ISSN: 1568-4946. DOI: 10.1016/j.asoc.2014.07.010. URL: https://www.sciencedirect.com/science/article/pii/S1568494614003408 (visited on 08/08/2023).

[33] Pawel Liskowski et al. "Comparison of Semantic-aware Selection Methods in Genetic Programming". In: ***Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation***. GECCO '15: Genetic and Evolutionary Computation Conference. Madrid Spain: ACM, July 11, 2015, pp. 1301–1307. ISBN: 978-1-4503-3488-4. DOI: 10.1145/2739482.2768505. URL: https://dl.acm.org/doi/10.1145/2739482.2768505 (visited on 06/21/2023).

[34] Clara Schmitt et al. "Half a Billion Simulations: Evolutionary Algorithms and Distributed Computing for Calibrating the Simpoplocal Geographical Model". In: ***Environment and Planning B: Planning and Design*** 42.2 (Apr. 1, 2015), pp. 300–315. ISSN: 0265-8135. DOI: 10.1068/b130064p. URL: https://doi.org/10.1068/b130064p (visited on 08/08/2023).

[35] Qi Chen, Mengjie Zhang, and Bing Xue. "Feature Selection to Improve Generalization of Genetic Programming for High-Dimensional Symbolic Regression". In: ***IEEE Transactions on Evolutionary Computation*** 21.5 (Oct. 2017), pp. 792–806. ISSN: 1941-0026. DOI: 10.1109/TEVC.2017.2683489.

[36] Mozhan Soltani, Annibale Panichella, and Arie Van Deursen. "A Guided Genetic Algorithm for Automated Crash Reproduction". In: ***2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)***. 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). Buenos Aires: IEEE, May 2017, pp. 209–220. ISBN: 978-1-5386-3868-2. DOI: 10.1109/ICSE.2017.27. URL: http://ieeexplore.ieee.org/document/7985663/ (visited on 12/08/2023).

[37] Anas Arram and Masri Ayob. "A Novel Multi-Parent Order Crossover in Genetic Algorithm for Combinatorial Optimization Problems". In: ***Computers & Industrial Engineering*** 133 (July 1, 2019), pp. 267–274. ISSN: 0360-8352. DOI: 10.1016/j.cie.2019.05.012. URL: https://www.sciencedirect.com/science/article/pii/S0360835219302773 (visited on 08/24/2023).

[38] Sergii Yaroshko and Svitlana Yaroshko. "Multithreaded Evolutionary Computing". In: ***2019 IEEE 2nd Ukraine Conference on Electrical and Computer Engineering (UKRCON)***. 2019 IEEE 2nd Ukraine Conference on Electrical and Computer Engineering (UKRCON). July 2019, pp. 1041–1045. DOI: 10.1109/UKRCON.2019.8879863.

[39] Alexandre Bergel. ***Agile Artificial Intelligence in Pharo: Implementing Neural Networks, Genetic Algorithms, and Neuroevolution***. For Professionals by Professionals. New York: Apress, 2020. 386 pp. ISBN: 978-1-4842-5384-7 978-1-4842-5383-0.

[40] Alexandre Bergel and Ignacio Slater Muñoz. "Beacon: Automated Test Generation for Stack-Trace Reproduction Using Genetic Algorithms". In: ***2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)***. 2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST). Madrid, Spain: IEEE, May 2021, pp. 1–7. ISBN: 978-1-66544-571-9. DOI: 10.1109/SBST52555.2021.00007. URL: https://ieeexplore.ieee.org/document/9476241/ (visited on 06/07/2023).

[41] Qi Chen, Bing Xue, and Mengjie Zhang. "Rademacher Complexity for Enhancing the Generalization of Genetic Programming for Symbolic Regression". In: ***IEEE Transactions on Cybernetics*** 52.4 (Apr. 2022), pp. 2382–2395. ISSN: 2168-2275. DOI: 10.1109/TCYB.2020.3004361.

[42] Zhengyuan Wang, Hui Zhang, and Yali Li. "Fast Polynomial Time Approximate Solution for 0-1 Knapsack Problem". In: ***Computational Intelligence and Neuroscience*** 2022 (Oct. 22, 2022), e1266529. ISSN: 1687-5265. DOI: 10.1155/2022/1266529. URL: https://www.hindawi.com/journals/cin/2022/1266529/ (visited on 08/09/2023).

[43] djakobovic. ***ECF - Evolutionary Computation Framework***. Mar. 13, 2023. URL: https://github.com/djakobovic/ECF (visited on 08/09/2023).

[44] A.Shaji George, A.S.Hovan George, and A.S.Gabrio Martin. "The Environmental Impact of AI: A Case Study of Water Consumption by Chat GPT". In: (Apr. 20, 2023). DOI: 10.5281/ZENODO.7855594. URL: https://zenodo.org/record/7855594 (visited on 12/07/2023).

[45] Diego Giacomelli. ***GeneticSharp***. June 7, 2023. URL: https://github.com/giacomelli/GeneticSharp (visited on 06/08/2023).

[46] ***Kotlin Telegram Bot***. Kotlin Telegram Bot, Aug. 3, 2023. URL: https://github.com/kotlin-telegram-bot/kotlin-telegram-bot (visited on 08/03/2023).

[47] ***Kotlinx.Html***. Kotlin, Aug. 3, 2023. URL: https://github.com/Kotlin/kotlinx.html (visited on 08/03/2023).

[48] ***Pagmo***. European Space Agency, Aug. 8, 2023. URL: https://github.com/esa/pagmo2 (visited on 08/09/2023).

[49] Christian S. Perone. ***Perone/Pyevolve***. May 23, 2023. URL: https://github.com/perone/Pyevolve (visited on 06/08/2023).

[50] ***Pygmo***. European Space Agency, June 4, 2023. URL: https://github.com/esa/pygmo2 (visited on 06/08/2023).

[51] ***The Latest Programming Languages Statistics & Trends For 2023 • GITNUX***. Feb. 1, 2023. URL: https://blog.gitnux.com/programming-languages-statistics/ (visited on 07/21/2023).

[52] Zhuokui Xie et al. "ChatUniTest: A ChatGPT-based Automated Unit Test Generation Tool". Version 1. In: (2023). DOI: 10.48550/ARXIV.2305.04764. URL: https://arxiv.org/abs/2305.04764 (visited on 12/07/2023).

[53] ***Android's Kotlin-first Approach***. Android Developers. URL: https://developer.android.com/kotlin/first (visited on 07/21/2023).

[54] ***Arrow***. URL: https://arrow-kt.io/ (visited on 01/29/2024).

[55] ***Calling Java from Kotlin | Kotlin***. Kotlin Help. URL: https://kotlinlang.org/docs/java-interop.html (visited on 07/21/2023).

[56] ***Calling Kotlin from Java | Kotlin***. Kotlin Help. URL: https://kotlinlang.org/docs/java-to-kotlin-interop.html (visited on 07/21/2023).

[57] Fort Collins. "A Comparison of Genetic Sequencing Operators". In: (), p. 8.

[58] ***DEAP Documentation — DEAP 1.3.3 Documentation***. URL: https://deap.readthedocs.io/en/master/ (visited on 06/08/2023).

[59] ***Easy_ga/Src at Main · RubenRubioM/Easy_ga***. GitHub. URL: https://github.com/RubenRubioM/easy_ga (visited on 06/13/2023).

[60] ***ECJ***. URL: https://cs.gmu.edu/~eclab/projects/ecj/ (visited on 06/08/2023).

[61] ***EvolvingObjects: Welcome to Evolving Objects***. URL: https://eodev.sourceforge.net/eo/doc/html/index.html (visited on 06/08/2023).

[62] ***Genevo - Rust***. URL: https://docs.rs/genevo/latest/genevo/index.html (visited on 06/13/2023).

[63] ***GNU Lesser General Public License v3.0 - GNU Project - Free Software Foundation***. URL: https://www.gnu.org/licenses/lgpl-3.0.en.html (visited on 07/26/2023).

[64] ***Inspyred: Bio-inspired Algorithms in Python — Inspyred 1.0.1 Documentation***. URL: https://aarongarrett.github.io/inspyred/ (visited on 06/08/2023).

[65] ***JetBrains/Lets-Plot-Skia: Skia Frontend for Lets-Plot Multiplatform Plotting Library.*** URL: https://github.com/JetBrains/lets-plot-skia/tree/main (visited on 12/10/2023).

[66] ***Kotest | Kotest***. URL: https://kotest.io/ (visited on 08/03/2023).

[67] ***Kotlin Programming Language***. Kotlin. URL: https://kotlinlang.org/ (visited on 07/21/2023).

[68] ***Kotlin-Java Interop Guide***. Android Developers. URL: https://developer.android.com/kotlin/interop (visited on 07/21/2023).

[69] ***One Max Problem — DEAP 1.3.3 Documentation***. URL: https://deap.readthedocs.io/en/master/examples/ga_onemax.html (visited on 06/11/2023).

[70] ***Q&A: UW Researcher Discusses Just How Much Energy ChatGPT Uses***. UW News. URL: https://www.washington.edu/news/2023/07/27/how-much-energy-does-chatgpt-use/ (visited on 12/07/2023).

[71] ***Schlatterbeck/Pgapack: Parallel Genetic Algorithm Library Originally by David Levine from Argonne National Laboratory***. URL: https://github.com/schlatterbeck/pgapack/tree/master (visited on 08/09/2023).

[72] ***Schlatterbeck/Pgapy: Python Wrapper for Pgapack, the Parallel Genetic Algorithm Library***. URL: https://github.com/schlatterbeck/pgapy (visited on 08/09/2023).

[73] Ignacio Slater Muñoz. ***Keen | Kotlin Genetic Algorithm Framework***. URL: https://github.com/r8vnhill/keen.

[74] ***Stream (Java Platform SE 8)***. URL: https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html (visited on 07/26/2023).

[75] ***The Top Programming Languages***. The State of the Octoverse. URL: https://octoverse.github.com/2022/top-programming-languages (visited on 07/21/2023).

[76] ***Type-Safe Builders | Kotlin***. Kotlin Help. URL: https://kotlinlang.org/docs/type-safe-builders.html (visited on 07/31/2023).

[77] Franz Wilhelmstötter. ***Jenetics: Java Genetic Algorithm Library***. jenetics.io. URL: https://jenetics.io/ (visited on 06/08/2023).