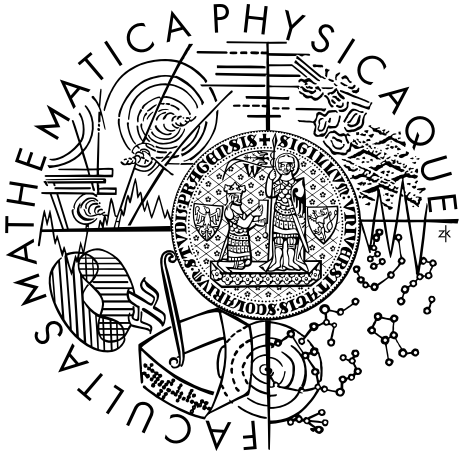
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Petr Mánek

Genetic programming in Swift for human-competitive evolution

Department of Software and Computer Science Education

Supervisor of the bachelor thesis: RNDr. František Mráz, CSc.
Study programme: Computer Science
Study branch: General Computer Science

Prague 2016

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

Title: Genetic programming in Swift for human-competitive evolution

Author: Petr Mánek

Department: Department of Software and Computer Science Education

Supervisor: RNDr. František Mráz, CSc., Department of Software and Computer Science Education

Abstract: Imitating the process of natural selection, evolutionary algorithms have shown to be efficient search techniques for optimization and machine learning in poorly understood and irregular spaces. In this thesis, we implement a library containing essential implementation of such algorithms in recently unveiled programming language Swift. The result is a lightweight framework compatible with Linux-based computing clusters as well as mobile devices. Such wide range of supported platforms allows for successful application even in situations, where signals from various sensors have to be acquired and processed independently of other devices. In addition, thanks to Swift's minimalistic and functional syntax, the implementation of bundled algorithms and their sample usage clearly demonstrates fundamentals of genetic programming, making the work usable in teaching and quick prototyping of evolutionary algorithms.

Keywords: genetic programming artificial evolution

Dedication.

Contents

1. Introduction

1.1 Evolutionary Algorithms

TODO

1.2 Genetic Programming

TODO

1.3 The Swift Language

TODO

1.4 Practical Application

TODO

1.5 Structure of This Document

TODO

2. Background

This chapter is dedicated to establishing and describing terms needed to understand the rest of the work.

2.1 Genetic Algorithms

Genetic algorithms (GA) are iterative randomized optimization methods[1], which are inspired by the biological process of natural selection. In the context of GA, points in the domain space are likened to *individuals* of animal species. Every individual carries a *chromosome*, which describes its location in the domain space, thus defining its properties similarly to the way DNA defines skills and capabilities of its carriers.

To initialize the GA, a *population* of individuals with random chromosomes is generated. During single iteration of the GA, individuals in the population compete for their right to reproduce, favoring those who maximize the value of a *fitness function* which customarily maps every individual to a single decimal value from the $[0; 1]$ interval. At the end of the iteration, fit individuals are selected and allowed to produce an *offspring population*, on which the next iteration of the GA operates. This behavior creates an iterative loop, which can be summarized by a diagram shown in Figure 2.1.

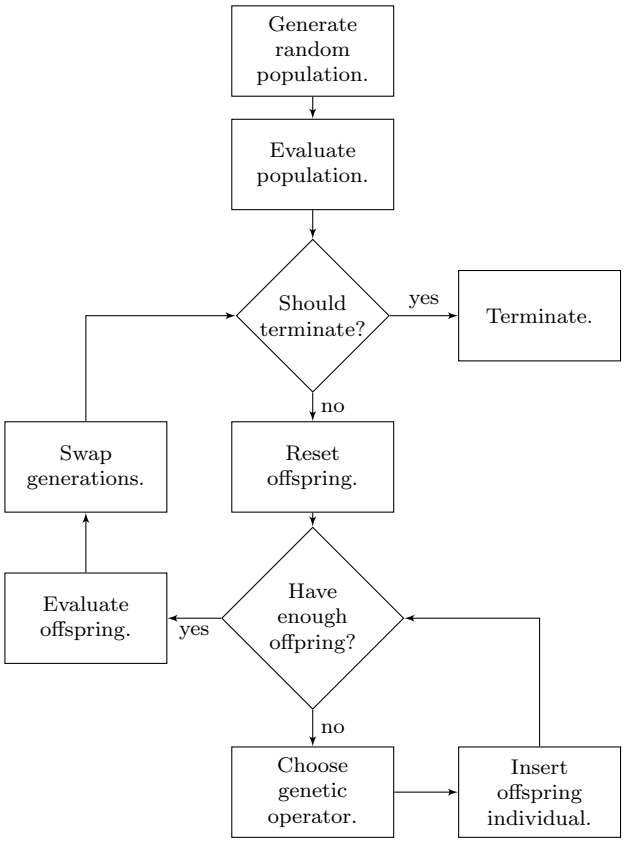


Figure 2.1: Generalized decision diagram of a GA.

[1]In this work, genetic algorithms are equivalent to *generational models* defined in [1].

Although there are many variants of GA, each with different properties and applications, the basic notion stays the same. Genetic algorithms iteratively evaluate and modify a population of chromosomes until a set *termination condition* met. This condition can for instance ascertain the fitness of the best chromosome so far or simply count the number of iterations performed. Individuals are inserted in the population at two instances throughout the execution of the GA: first during the initialization when a random population is generated, and second when the population is modified to prepare grounds for the next iteration. The latter of the two is the key phase of the GA. At this point in execution, the algorithm explores new points of the domain space by reusing points which have already been discovered. Such process is often based on non-deterministic inputs and can utilize the fitness of the chromosomes discovered so far as a heuristic. Instances of this process are referred to as *genetic operators*.

The *selection pressure* is the degree to which the better individuals are favored: the higher the selection pressure, the more the better individuals are favored. This selection pressure drives the GA to improve the population fitness over succeeding generations. However, if the selection pressure is too low, the convergence rate will be slow, and the GA will unnecessarily take longer to find the optimal solution. If the selection pressure is too high, there is an increased chance of the genetic algorithm prematurely converging to an incorrect (suboptimal) solution. [2]

2.2 Neural Networks

A neural network is an interconnected assembly of simple processing elements, *units* or *nodes*, whose functionality is loosely based on the animal neuron. The processing ability of the network is stored in the interunit connection strengths, or *weights*, obtained by a process of adaptation to, or *learning* from a set of training patterns. [3]

A neural network is called *feedforward* (denoted FFNN), if its interunit connections do not form a cycle. In such case, it makes sense to clasify its nodes in separate *layers* with respect to their topological ordering. The first layer of the network is customarily called the *input layer*, whereas the last layer is called the *output layer*. The remaining layers are referred to as the *hidden layers*. This is illustrated in Figure 2.2.

2.2.1 Mathematical Representation

In feedforward neural networks, signals travel between connected nodes in a way resembling the action potential of biological neural systems. At first, nodes of the input layer are evaluated with real numbers. From these values, nodes of the second layer calculate their outputs. Then, nodes of the third layer calculate their outputs based on the outputs of the second layer, and so on. This process propagates through the rest of the network until the output layer is reached. Since there are no cycles in the interunit connections of the FFNN, a finite number of steps is required to achieve such state. The outputs of the nodes located in the output layer are considered to be the outputs of the neural network. By this description, it is possible to think of the FFNN as a real vector function from

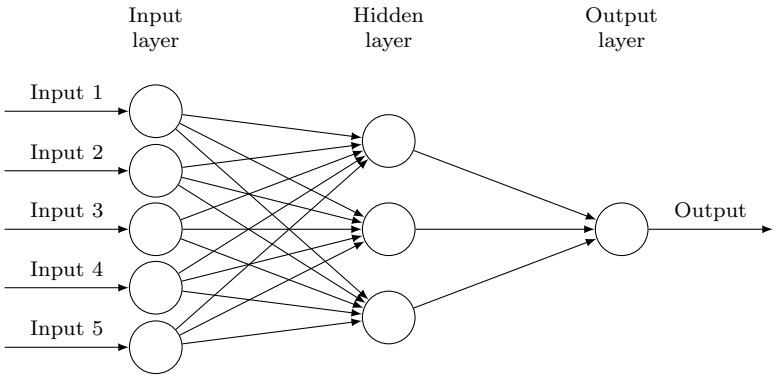


Figure 2.2: A feedforward neural network comprised of three layers of nodes.

m-dimensional to n-dimensional space, where m, n denote the number of nodes in the input and the output layer respectively.

The process of calculating the output of a single node with respect to its inputs (which are in fact the outputs of the nodes of the previous layer) is quite straightforward. The output is defined as

$$y = f\left(b + \sum_{i=1}^{n} w_i x_i\right) \tag{2.1}$$

where $\{x_i\}_{i=1}^{n}$ denote the input values, $\{w_i\}_{i=1}^{n}$ denote the interunit connection weights, b denotes the *bias parameter* of the node and f denotes the *activation function*. The combination of all these parameters is illustrated in Figure 2.3.

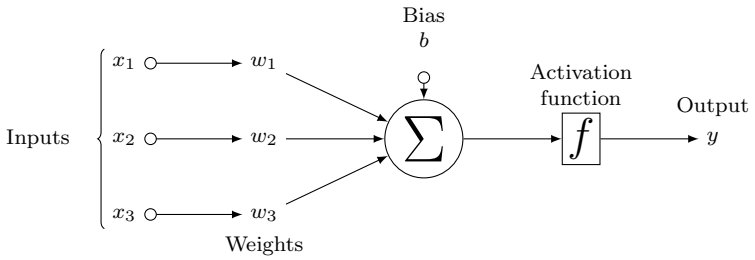


Figure 2.3: A diagram of FFNN node evaluation with three inputs.

It is worth noting at this point that a constant number of weights can be achieved by assigning non-existant interunit connections zero weights. For the purposes of implementation, it is also possible to approximate the effects of the bias parameter by creating one additional node in every layer and configuring it to produce constant output. The bias of every node in the layer is then simply encoded into connection weights of such node.

2.3 Redistributable Applications

TODO

2.4 Analysis

TODO

2.5 Requirements

TODO

3. Library Documentation

This chapter contains technical documentation of individual components of the library. To better illustrate some concepts, examples and code demonstrations are included.

The overall architecture of the library is based on generics and object polymorphism. Since the library offers object definitions as well as their implementation, it often defines Swift *protocols* (similar to interfaces in other programming languages) or "abstract classes"[1], which are implemented by some of its objects. The purpose of this approach is to offer users a selection of ready-to-use building blocks as well as the option of customization, useful in certain marginal situations.

3.1 Chromosomes

In the context of GA, *a chromosome* (also known as *genotype*) is a piece of information describing a solution to a problem. [4] Within the library, chromosomes can be represented by any reference types, which conform to the `ChromosomeType` protocol. This protocol requires them to

1. be immutable,[2]
2. be capable of randomly generating new instances of themselves.[3]

This section explains in detail, how to achieve common configurations using preimplemented types, how to customize them for different applications, and how to define custom data types for storing proprietary information.

3.1.1 Data Representation Problem

When designing chromosome data structures, users first need to decide which information should be stored within chromosomes and how should such information be encoded into primitive types. These questions might not always be trivial to answer and it is possible to show that unfortunate choices could potentially impact the rate of convergence of the GA significantly. This is known as *the problem of data representation.*

It is worth noting at this point that the complexity of this problem extends far beyond the scope of this work, and is thus not addressed. For more information on this topic, readers are referred to [4].

3.1.2 Strings

A popular method of storing chromosomes is to encode them as strings of values of the same type, e. g. binary or numeric. The library represents such strings in

[1] In conventional programming languages, abstract classes contain unimplemented method definitions. Since Swift does not support this paradigm, it is emulated through the mechanism of static precondition failures.

[2] This is a semantical requirement implying that every chromosome modification will require a new instance of the type to be created.

[3] This is achieved by requiring conformance to the `Randomizable` protocol.

the form of *range-initalized arrays*.

A range-initialized array is a generalization of a regular array. It is a generic list structure, which is capable of holding finite amounts of ordered homogeneous items. However, at the time of initialization, the number of elements in the array is set to a value, which randomly selected from a given number interval. This allows for more flexibility, since in some applications it is desirable to vary not only the contents of the chromosome, but also its size. If this behavior is not wanted, the array can be reconfigured to a constant length by specifying any interval of length zero.

A simple usage of range-initialized arrays can be demonstrated on finding solutions to the Knapsack Problem. Suppose that there are 10 things of different sizes and values and a knapsack of a limited capacity. The objective is to select things in order to maximize the total value of the knapsack contents, while not exceeeding its capacity. Clearly, all solutions of this problem can be described as strings of 10 Boolean values, indicating whether items 1-10 are selected. These values can be stored in a range-initialized array with interval $[10; 10]$, implying that the array has fixed size 10. The array class is declared in Listing 1.

```
1 struct KnapsackChromosome: RangeInitializedArray {
2     typealias Element = Bool
3     static let initializationRange = 10...10
4
5     let array: [Element]
6     init(array: [Element]) {
7         self.array = array
8     }
9 }
```

Listing 1: Range-initialized array used to solve the Knapsack problem.

In a similar way, range-initialized arrays can store integers to encode number sequences or floating-point decimals to describe connection weights of neural networks. Thanks to Swift type extensions, every instance of range-initialized array automatically conforms to the `ChromosomeType` protocol and supports three basic genetic operators (for definition, see Section 3.3). Range-initialized arrays can thus be directly used as chromosomes in the GA without any further modification.

It is worth noting at this point that strings are **not designed to hold heterogeneous information**. In spite of that, it is possible to use them for such purpose. For instance, if a chromosome is required to contain numbers as well as bits, it can be encoded as a binary string, portions of which would be later interpreted[4] as integers by the application. While this approach succeeds in its purpose, it is strongly discouraged as it may also become a cause to various subsequent problems. For example, when applying genetic operators on the chromosome, the bundled implementation mutates range-initialized arrays by selecting a random element and modifying its value. In conventional situations, this is the desired behavior. However, if the algorithm happens to select an item of the

[4]Interpretation can be performed in compliance with any known encoding, e. g. conventional signed encoding, BCD or the Gray code (RBC).

array, which is merely a part of a greater whole (e. g. number), unfortunate modification of such item could cause the chromosome to become undecodable. Instead, the recommended alternative is to use custom types (see Section 3.1.4), which not only avoid this issue, but also allow strongly-typed information to be checked at the time of compilation, discovering any possible type conversion errors.

3.1.3 Trees

Tree structures are commonly used in applications, which require automatic code generation. In such applications, individuals often carry chromosomes which contain control programs, mathematical formulas or similar information that can be represented by tree graphs. The library allows to represent such type of data by a collection of *tree nodes*.

A tree node is an abstract data structure, which can be configured to contain information of any type. In addition, tree nodes can point to multiple other tree nodes, linking the information they contain together, in order to form a forest. The library recognizes two fundamental types of tree nodes:

Value Nodes (generic) The purpose of a value node is to produce a value of some kind. While the means of producing the value may differ (e. g. constant, function or binary operation) as well as its type, every value node must offer a way to retrieve its value at runtime. This type of node is represented by the generic class `ValueNode<T>`.

Action Nodes The purpose of an action node is to perform an action at runtime. The action may be a command of some kind, or may call other action, possibly requiring arguments in the form of other value nodes. This type of node is represented by the class `ActionNode`.

Both types of nodes are intentionally left abstract, guiding users to define their own node types for functions, operations and commands depending on their applications. This procedure is very simple and can be demonstrated on a maze robot simulation. Suppose that there is a robot, which can receive `WAIT`, `GO`, `STOP`, `TURN-LEFT` and `TURN-RIGHT` instructions in order to navigate a 2-dimensional maze. The robot is also capable of determining whether its front side is facing an obstacle. To auto-generate a control program for such robot, its instructions can be formalized as 5 subclasses the class `ActionNode` and the sensor output can be represented by a subclass of the class `ValueNode<Bool>`. Such formalization is shown in Listing 2.

It is conceivable that combinations of various tree nodes can be translated into a language, which is similar to LISP in its architecture (as illustrated in Figure 3.1). To produce fundamental building blocks of such language, *a tree factory* object is required. Factories create new randomized instances of tree nodes, and can thus restrict or extend types of generated nodes depending on the application. The library contains various bundled node types, ready to use:

Constants Constant nodes (`ConstantNode<T>`) contain constant values of any type, unchanging during program execution.

```
class GoCommandNode: ActionNode {
    override func perform(interpreter: TreeInterpreter) {
        guard interpreter.running else { return }
        // Tell the robot to go forward.
        // The interpreter contains the current context.
    }

    override func propagateClone(factory: RandomTreeFactory,
        mutateNodeId id: Int) -> ActionNode {
            let clone = GoCommandNode(id: id, maximumDepth: maximumDepth)
            // This node contains no descendants.
            return clone
    }
}
```

Listing 2: Example implementation of the GO command action node.

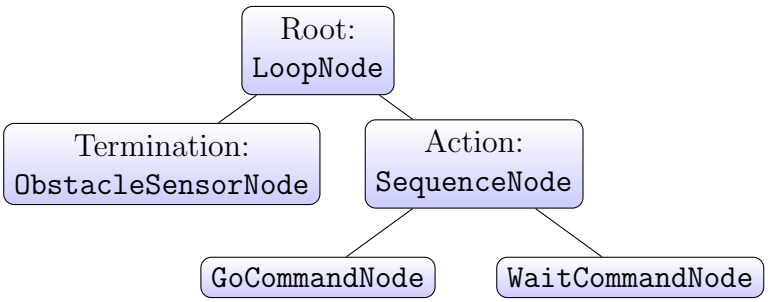


Figure 3.1: Example program for the maze robot simulation, which makes the robot go forward until it encounters an obstacle.

Operations Operation nodes (descendants of classes `UnaryOperation<T1,T2>` and `BinaryOperation<T1,T2,T3>`) are generic templates for functions. Arguments of such functions are represented by other value node instances.

Comparisons Comparison nodes represent equality (`EquationNode<T>`) and inequality predicates (`ComparisonNode<T>`), operating on tuples of other value node instances.

Arithmetic and Boolean Operations For any numeric value nodes, addition, subtraction, multiplication, division and modulation are supported. In analogy, Boolean value nodes support negation, conjunction, disjunction, implication and equivalence.

Control-flow Primitives Action nodes can be combined in sequences, loops or simple conditional expressions. Names of the node types responsible for this functionality are analogous to those listed above.

It is recommended that tree factories are instantiated in the global context, or in subclasses of entropy generators (see Listing 3). Apart from controlling the type of generated nodes, factories allow to specify upper bounds of the depth and width of the tree, restricting the number of generated structures.

```
class RobotProgramFactory: TreeFactory {
    /* ... */
}

class MyGenerator: MersenneTwister {
    let robotProgramFactory: RobotProgramFactory

    override init(seed: Int) {
        robotProgramFactory = RobotProgramFactory(generator: self)
        super.init(seed: seed)
    }
}
```

Listing 3: Tree factory declared in an entropy generator subclass.

3.1.4 Custom Types

If the chromosome information is not compatible with strings or trees, or is heterogeneous in its nature, it is recommended that a custom data type is declared to hold it. This allows users to label, document and describe individual parts of the chromosome, as well as to customize its behavior at important points of evaluation.

Any reference type can become a chromosome data structure, if it conforms to the `ChromosomeType` protocol (and its inherited protocols). No other protocol conformance is formally required. Nevertheless, it is worth noting that some genetic operators may require chromosomes to conform to other proprietary protocols, in order to operate on them. For instance, the `Mutable` protocol, which is required by the `Mutation` operator. For description of such protocols, see Section 3.3.

Declaration of custom types can be demonstrated on The Hamburger Restaurant Problem, mentioned in the introduction[5] of [5]. The objective is to find a business strategy for a chain of hamburger restaurants, which yields the biggest profit. A strategy consists of three decisions:

Price What should be the price of the hamburger? Should it be 50 cents, 10 dollars or anywhere in between?

Drink What drink should be served with the hamburger? Water, cola or wine?

Speed of service Should the restaurant provide slow, leisurely service by waiters in tuxedos or fast, snappy service by waiters in white polyester uniforms?

Clearly, every strategy is a heterogeneous data structure. Although it could be encoded into a binary string as proposed in Section 3.1.2, it is much safer and more elegant to declare a dedicated type to hold its information. Such declaration is shown in Listing 4.

Note that in the example declaration, every property is named and strongly-typed, clearing up any possible confusion about their purpose, and preventing

[5]For the purposes of this work, the example has been slightly altered.

```swift
class RestaurantStrategy: Randomizable {
    let hamburgerPrice: Double
    let drink: Drink
    let waiterSpeed: Speed

    init(generator: EntropyGenerator) {
        // Generate random values.
        hamburgerPrice = generator.nextInRange(min: 0.5, max: 10)
        drink = generator.next()
        waiterSpeed = generator.next()
    }
}
```

Listing 4: Example declaration of custom chromosome type.

type casting issues in the future. Moreover, the custom implementation of the randomization initializer allows users to specify clear bounds for fields, such as the hamburger price. Thanks to Swift generics, fields of type `Drink` and `Speed` can also be randomly initialized through the entropy generator, provided that they do conform to the `Randomizable` protocol. This way, the randomization call is propagated to all fields of the data structure.

Lastly, it is worth mentioning that types which are capable of listing all their possible values in a set of finite cardinality can utilize the `Discrete` protocol. This protocol functions as a simple time-saving shorthand for the `Randomizable` protocol, since it produces random values from the discrete uniform distribution of all values in the set. A good demonstration of this is a possible implementation of the `Drink` type, which is declared as a Swift enumeration in Listing 5.

```swift
enum Drink: Discrete, Randomizable {
    case Water, Wine, Cola
    static let allValues: [Drink] = [.Water, .Wine, .Cola]
}
```

Listing 5: Declaration of a chromosome type through a discrete listing of values.

As shown by the demonstrations, declaration of custom types for heterogeneous chromosomes in Swift is effortless, safe and efficient. However, the reader should not be misled into thinking it only serves for creating nicely annotated vessels for information. This technique can be also used to create more complex genotype containers with customized behavior and proprietary internal structure, which is most notably exemplified by strings and trees, as both types are implemented in this way.

3.2 Population Evaluation

In order to assess and compare the quality of chromosomes with respect to the optimization problem at hand, a common fitness evaluation model is used. In

this model, every chromosome is assigned a value from the $[0; 1]$ interval by a *fitness function*, which is heavily dependent on the application and thus required to be specified by the user.

The fitness function is encapsulated in an *evaluator* object, which is active for the entire duration of evaluation. The purpose of this encapsulation is to enable the possibility of accelerating the evaluation process by minimizing computing overhead needed to set up and tear down other components required to perform the evaluation itself. Since fitness functions often involve resource-expensive simulations and randomized testing scenarios, such optimization may be efficient.

3.2.1 Mating Pool

In the context of population evaluation, a *mating pool* represents a collection of individuals relevant to the current iteration of the GA. The pool is divided in two parts: the *current generation* and the *offspring*. Evaluators operate only on the first of these two.

Apart from chromosomes, individuals in the mating pool also have an optional field dedicated for their fitness value. When a new individual is inserted into the population, value of this field is not present. However, individuals transitioning between generations carry their previously set fitness with them. In a single iteration of the GA, the fundamental purpose of an evaluator object is to ensure that all individuals in the current generation have a non-null fitness value.

3.2.2 Sequential and Parallel Evaluators

The library supports two evaluation modes: *sequential* and *parallel*. While the sequential mode is easier to implement but leads to slower evaluation, the parallel mode is faster but requires the internals of the fitness function to be compatible with multi-threaded processing, which may not always be feasible with respect to the problem definition.

To demonstrate implementation of a simple sequential evaluator, recall the chromosome structure, which was proposed earlier[6] for the Knapsack Problem. Suppose the fitness function is defined as

$$f(c_1, c_2, \ldots, c_{10}) = \begin{cases} 0 & \text{if } \sum_{i=1}^{10} c_i s_i > S_{max} \\ \sum_{i=1}^{10} c_i v_i / \sum_{i=1}^{10} v_i & \text{otherwise} \end{cases} \tag{3.1}$$

where S_{max} represents the maximum capacity of the knapsack, $\{s_i\}_{i=1}^{10}$ are sizes of things, $\{v_i\}_{i=1}^{10}$ are values of things and $\{c_i\}_{i=1}^{10}$ are 0/1 coefficients generated from the Boolean values of the chromosome. A simple implementation of a sequential evaluator using this function is shown in Listing 6.

In the example, the evaluator is a descendant of the generic abstract class `SequentialEvaluator<T>`, which is a common base class for all sequential evaluators. Similarly, all parallel evaluators have to be descendants of the class `ParallelEvaluator<T>`, which instantiates multiple sequential evaluators operating on different threads and manages internal producer-consumer queue to

[6]The Knapsack Problem is defined in Section 3.1.2. For chromosome structure, see Listing 1

```
class KnapsackEvaluator: SequentialEvaluator<KnapsackChromosome> {
    // These values are part of the problem instance.
    let thingValues: [Double], thingSizes: [Double]
    let knapsackCapacity: Double

    // This value is only calculated on the first time it is needed.
    lazy var maxValue: Double = self.thingValues.reduce(0, combine: +)

    override func evaluate(chromosome: KnapsackChromosome) -> Fitness {
        let size = zip(chromosome.array, thingSizes).reduce(0)
            { $0 + ($1.first ? $1.second : 0) }
        if size > knapsackCapacity { return 0 }

        let value = zip(chromosome.array, thingValues).reduce(0)
            { $0 + ($1.first ? $1.second : 0) }
        return value / maxValue
    }
}
```

Listing 6: Example of a sequential evaluator for the Knapsack Problem.

facilitate parallel evaluation of chromosomes. Moreover, both types of evaluators inherit from `Evaluator<T>`, an abstract class which defines the formal requirements on all evaluator objects.

When implementing fitness evaluator classes, it is recommended that the class `Evaluator<T>` is directly subclassed only in cases, when the evaluation scheme is incompatible the other already existing subclasses. A good example of such scenario would be an evaluator utilizing a distributed computing cluster. However, directly subclassing `Evaluator<T>` only to create a custom implementation of sequential evaluator is not advisable, since `SequentialEvaluator<T>` is integrated into other components of the library and avoiding its use would introduce subsequent issues.

3.2.3 Cyclic Evaluators

Every descendant of the class `SequentialEvaluator<T>` is eligible to be combined with a *cyclic evaluator* (represented by the class `CyclicEvaluator<T>`). Instances of cyclic evaluators encapsulate other evaluators, which are called multiple times in order to evaluate a single chromosome. Such sub-evaluations are then statistically aggregated to produce a final fitness value. This procedure mimics a commonly used technique in GA fitness evaluation, which yields more stable and reliable fitness values, especially in randomized simulations.

Usage of cyclic evaluators is straightforward. At the instantiation time, a cyclic evaluator receives 3 parameters: a sequential evaluator, and numbers n and m. Upon evaluation of an individual, the sequential evaluator is called n times, producing a vector of n fitness sub-evaluations. From this vector, m of the highest (or the lowest) fitness values are then selected. The final fitness value returned by the cyclic evaluator is the average calculated from the selected values.

3.3 Genetic Operators

Genetic operators are procedures, which are performed on collections of individuals during the evaluation of the GA in order to transition between iterations. When operators are applied, the entire mating pool is available to them. While the current generation can be merely accessed for reading, the offspring generation also supports writing. Every operator can thus read an arbitrary number of chromosomes from the current generation, and is expected to insert at least one chromosome into the offspring generation.

The selection of chromosomes is carried out through *selection objects*, which are specified as configuration parameters of individual operators. There are various types of selections, each providing the selection pressure in a different way, thus being suitable for different classes of applications. For the description of supported selection methods, see Section 3.4.

The library offers implementation of three common genetic operators: *reproduction*, *mutation* and *crossover*. However, users are by no means limited to only these three. This section shows the usage of the preimplemented operators and gives details and recommendations on creating custom ones.

3.3.1 Reproduction

The reproduction operator mimics the asexual reproduction of natural organisms, which have survived long enough to mature. Unlike others, this operator does not introduce any novelty into the offspring generation. Instead, its purpose is to simply stabilize the population by carrying certain traits between generations. This in effect prevents the loss of diversity and thereby avoids premature convergence of the GA, which may lead to a suboptimal solution.

When applied on the population, the reproduction operator copies arbitrary number of selected chromosomes from the current generation to the next one without any modifications. Since the selection of individuals is independent of the operator implementation, it is technically possible to use any selection object with this operator. Nevertheless, it is worth noting that only fitness-proportionate strategies make sense in this context. A good example of such strategies is *elitism*, which is more thoroughly described in Section 3.6.1.

Since individuals are immutable by definition, the reproduction operator requires their underlying chromosome data structures to conform to the `Reproducible` protocol in order to work properly. This protocol is a simple extension of the `Copyable` protocol, which requires types to be capable of producing deep copies of their instances.

3.3.2 Mutation

The mutation operator serves the desirable function of introducing occasional variety into a population and restoring its lost diversity. [5] It is fundamentally similar to the reproduction operator, as it operates by copying selected chromosomes from the current generation to the next one. However before copying, the chromosomes are modified in a non-deterministic way (i.e. mutated), imitating random transcription errors during replication of genetic information in the nature. The degree of mutation in the chromosomes is static and determined by the

implementation. In general, mutated chromosomes can be expected to mostly resemble their original counterparts, yet not be completely identical.

In order to be used in the GA, the mutation operator requires chromosome data structures to conform to the `Mutable` protocol. Every container can thus have its own, slightly different implementation of mutation, which should be defined in its documentation. General guidelines for implementing mutation are:

1. Select one chromosome in the current generation.
2. Choose a "part" of the chromosome at random.
3. Copy the chromosome, substituting the chosen part for a randomly generated equivalent.
4. Insert the modified chromosome into the offspring generation.

Clearly, among various data structures the semantical definition of a "part" may differ. For instance, in range-initialized arrays, a part is defined as an item of the array. Since arrays are homogeneous, all their are equivalent by definition. In contrast, a part of a tree node structure is defined as a rooted subtree. Two subtrees are equivalent if the fundamental base classes of their root nodes match.

3.3.3 Crossover

The crossover operator emulates the act of sexual reproduction of individuals in the nature (also known as *recombination*). Unlike the previous two operators, it requires the input of exactly two chromosomes from the current generation, which are referred to as *the parent chromosomes*. During the execution of the operator, parts of the parent chromosomes are randomly chosen and exchanged, producing two new chromosomes, which are inserted into the offspring generation. These chromosomes carry a mixture of traits of the parent chromosomes, and can therefore be thought of as their *children*.

Similarly to the mutation operator, in order for the crossover to be used with chromosomes, their underlying data structure must conform to a dedicated Swift protocol, which allows users to customize the behavior of the operator with respect to the architecture of the data structure. The general guidelines for implementing such customizations are:

1. Select two distinct chromosomes in the current generation.
2. Choose pairs of "parts" of the chromosomes at random.
3. Copy both chromosomes, swapping the parts in each pair.
4. Insert the modified chromosomes into the offspring generation.

Depending on the number and size of interchanged parts, multiple classes of crossover operators[7] can be defined. For arrays and range-initialized arrays, two crossovers are implemented:

[7]Each crossover operator has a dedicated Swift protocol.

One-point crossover A single point is randomly chosen to divide both arrays in two parts. While the first pair of parts is kept at its original position, the second pair is swapped. This crossover is represented by the protocol `OnePointCrossoverable`.

Two-point crossover In analogous way to the one-point crossover, two points are randomly chosen to divide both arrays in three parts. The middle pair of parts is swapped between the chromosomes, whereas the remaining two pairs are left unmodified. This crossover is represented by the protocol `TwoPointCrossoverable`.

For tree node structures, crossover is implemented by selecting two random subtrees rooted in nodes, which are descendants of matching fundamental base classes. The execution of the operator is performed by swapping pointers to both subtrees. This crossover is represented by the protocol `TreeCrossoverable`.

TODO

3.3.4 Custom Operators

By creating descendants of the generic abstract class `GeneticOperator<T>`, users are free to implement and experiment with any genetic operators of their own. This section gives details and recommendations on implementing such subclasses.

The base class contains a selection object and an initializer method, which is used to configure the selection at the time of creation. This initializer must be called from any descendants as it is crucial to operator execution later on. The internal logic of the operator is controlled by the abstract method `apply()`, which receives a mating pool and an entropy generator object. In this method, the operator is expected to call the selection object exactly once and provide it with both mentioned objects as well as the number of individuals needed for its execution. The selection then returns a list of the selected indices, which can be used to access the objects of individuals containing chromosome data and fitness evaluations. To further illustrate this approach, an example of a custom operator implementation is shown in Listing 7.

It is strongly recommended that genetic operators exert no additional selection logic on top of the results returned by the selection object. Instead, such logic is recommended to be resolved by creating custom selection objects, which are capable of encapsulating other selection objects. If required, this technique can be applied in the operator initializer, forcing all selections to undergo such encapsulation, as shown in Listing 8.

In order to better work on the chromosome data structures, genetic operators can also define custom protocols, to which such structures can conform. By usage of Swift extensions, existing structures can be then altered to comply with any additional requirements specified by these protocols.

3.3.5 Decision Trees

Decision trees are used to describe the sequential application of operators in the GA. The library includes Swift syntax extensions to facilitate simple customization of such operator sequences. Two types of nodes are supported:

```swift
class MyCustomOperator<Chromosome: ChromosomeType> {
    let parameter: Int // You can specify custom parameters.

    init(_ selection: Selection<Chromosome>, parameter: Int) {
        self.parameter = parameter
        super.init(selection)
    }

    override func apply(generator: EntropyGenerator,
        pool: MatingPool<Chromosome>) {
            // Select 42 chromosomes from the current generation.
            let selectedIndices = selection.select(generator,
                population: pool, numberOfIndividuals: 42)

            // Access the individuals.
            for index in selectedIndices {
                let individual = pool.individualAtIndex(index)

                // Do something with the individual...
            }
    }
}
```

Listing 7: Example of a custom genetic operator implementation.

Operator nodes Operator nodes correspond to instances of application of genetic operators.

Chance nodes Chance nodes contain non-deterministic switches between multiple choices. Every choice specifies its probability and a decision subtree to execute, should it be selected.

To ease their usage, decision trees are defined by custom Swift operators. In order to concatenate tree nodes in a sequence, the three-dash arrow operator (e.g. `--->`) is used. Sequences produced by this operator resemble linked lists in their structure. The three-bar operator (e.g. `|||`) serves to determine choices in chance nodes. The syntax of both operators is illustrated by Listing 9.

3.4 Selections

The purpose of selection objects is to separate the methods of chromosome selection from the genetic operators. This approach allows users to easily combine operators with selection methods without the need for unnecessary subclassing.

As input, selection objects receive three parameters from their genetic operators: the current generation (together with fitness evaluations), an entropy generator and the number of requested chromosomes. In return, selection objects are expected to produce a list of indices of the selected chromosomes or fail with error should the population contain insufficient number of chromosomes. When accessing fitness evaluations, the library uses lazy-loading optimizations, in order

```
class MyCustomSelection<Chromosome: ChromosomeType> {
    /* ... */
}

class MyCustomOperator<Chromosome: ChromosomeType> {
    override init(_ selection: Selection<Chromosome>) {
        let encapsulated = MyCustomSelection(selection)
        super.init(encapsulated)
    }
    /* ... */
}
```

Listing 8: Example of a selection object encapsulation.

```
// Apply reproduction, then mutation.
let p1 = reproduction ---> mutation
// Non-deterministically select the operator.
let p2 = Choice(mutation, p: 0.3) ||| Choice(crossover, p: 0.7)
// A combination of both techniques.
let p3 = reproduction ---> (Choice(mutation, p: 0.3)
                          ||| Choice(crossover, p: 0.7))
```

TODO

Listing 9: Example of pipeline definition.

to prevent unnecessary sorting and data aggregation. For that reason, selection objects are not required to specify fitness-related dependencies. Instead, additional calculations are performed on the first instance when the information is required.

Similarly to genetic operators, the library offers the implementation of common selections and allows its users to customize their behavior, possibly creating their own subclasses. Such techniques are described at the end of this section.

3.4.1 Roulette Selection

Roulette selection is one of the most basic fitness-proportionate selection methods used in the GA. When applied, each chromosome is assigned a normalized probability proportional to its current fitness value. Based on the assigned probabilities, a random generator then selects chromosomes from a discrete non-uniform distribution. This process can be likened to a spin of unfair roulette wheel, where every chromosome is allocated a sector with angle proportional to its fitness. [6]

The application of this method can be shown on a simple example. Suppose that there are four chromosomes with fitness values 0.05, 0.4, 0.8 and 0.1. In order to generate a distribution, the roulette selection method merely normalizes fitness values to sum up to 1. Chromosomes are therefore assigned probabilities 0.04, 0.3, 0.59 and 0.07 respectively.

In the library, roulette selection is represented by the `RouletteSelection`

class, which has no arguments and can be combined with any genetic operator.

3.4.2 Rank Selection

Rank selection is a modification of the roulette selection method, which is better suited for cases with extreme differences in fitness values. In such situations, often a small group of fit chromosomes receives the majority of the roulette wheel, causing the rest of the population to be mostly neglected, thus leading to premature convergence of the GA.

To resolve these cases, rank selection first sorts all chromosomes by their current fitness values. Every chromosome is then assigned a probability proportional to its rank in the sequence (hence the name of the method). For example, if rank selection had been used instead of roulette, the chromosomes in the example from the previous section would be assigned ranks 1, 3, 4, 2 respectively. These ranks would be simply normalized to probabilities 0.1, 0.3, 0.4 and 0.2.

In the library, rank selection is represented by the `RankSelection` class, which has no arguments and can be combined with any genetic operator.

3.4.3 Tournament Selection

Tournament selection provides selection pressure by holding a tournament among s competitors, with s being the tournament size (or order). The winner of the tournament is the chromosome with the highest fitness of the s tournament competitors. [2]

The library contains implementation of tournament selection, where competitors are chosen from the population by another selection object. By default, this secondary selection is random. However, by changing this argument, users can customize the behavior of the tournament selection significantly.

This selection method is represented by the `TournamentSelection` class, which receives the value of parameter s and the secondary selection object upon instantiation, and can be combined with any genetic operator.

3.4.4 Miscellaneous

In addition to the three methods described in previous sections, the library contains implementation of primitive selection objects, which serve as utilities for other selections or operators:

Random selection This method selects chromosomes at random with no regards to their fitness values. It is represented by the `RandomSelection` class, which has no arguments and can be combined with any genetic operator.

Best selection This method deterministically selects chromosomes in the descending order of fitness values. It is represented by the `BestSelection` class, which has no arguments and can be combined with any genetic operator.

Worst selection This method deterministically selects chromosomes in the ascending order of fitness values. It is represented by the `WorstSelection`

class, which has no arguments and can be combined with any genetic operator.

3.4.5 Custom Selections

To create a selection object for a custom selection method, users need to subclass the generic class `Selection<T>`.

The internal logic of any selection object is contained within the implementation of the abstract function `select()`. This function receives an entropy generator, a population, which serves as the domain for the selection, and the requested number of chromosomes to select. The expected output of the method is a set of zero-based indices pointing to the requested number of selected chromosomes in the population, which are not required to be distinct.

In the implementation of the method, users are free to assume that the fitness evaluation of all chromosomes is available. Moreover, it possible to declare additional parameters or secondary selection objects during instantiation. If necessary, selection objects can also declare auxiliary protocols for chromosome types, in order to better integrate with their contents. A basic implementation of a custom selection object is shown in Listing 10.

```
class MySelection<Chromosome: ChromosomeType>: Selection<Chromosome> {
    override init() { }

    override func select(generator: EntropyGenerator,
        population: MatingPool<Chromosome>,
        numberOfIndividuals: Int) -> IndexSet {
            // Always select the first 3 individuals.
            return IndexSet([0, 1, 2])
    }
}
```

Listing 10: Example of custom selection implementation.

3.5 Execution

The library provides a simple interface for definition and configuration of the GA through the class `GeneticAlgorithm<T>`. In instances of this class, various components of the library come together to form a robust runtime environment for the execution of GA.

To configure the environment, the following parameters are required:

Population size The desired number of individuals in every generation of the mating pool. Since the number of individuals produced by genetic operators is not defined, the actual number of generated individuals may differ. In such cases, the value of this parameter serves as a lower bound to the actual value.

Decision Trees The GA allows two decision trees to be specified. For reasons of clarity, these trees are labeled α and β. While the α-tree is executed repeatedly at the beginning of every iteration of the GA, the β-tree is executed multiple times per iteration in order to populate the offspring generation.

Evaluator The evaluator object is instantiated for the entire duration of the execution of the GA. It receives notifications about its state and requests to evaluate individuals in every iteration. **TODO**

Termination condition At the end of every iteration, the termination object is called to determine whether the GA should continue execution or terminate. For more information about this object, see Section 3.5.1.

Entropy generator This object provides sequences of pseudorandom numbers used for non-deterministic parts of the GA. For more information about this object, see Section 3.6.3.

Once all dependencies are initialized, the GA is ready to commence execution. In this context, a single instance of execution is referred to as a *run*. Runs are executed synchronously[8] and are comprised of multiple iterations, depending on the termination condition. Each iteration consists of the following steps:

1. Ensure that all individuals in the current generation are evaluated.
2. Execute the α-tree once.
3. Reset the offspring generation.
4. Execute the β-tree repeatedly until the offspring generation size is sufficient.
5. Replace the current generation with the offspring.

3.5.1 Termination Conditions

Termination objects are descendants of the class `TerminationCondition<T>`. Their instances encapsulate Boolean functions, which determine whether the GA should terminate its execution upon the end of every iteration.

Objects provided by the library can be divided into two groups: the *primitives* and the *contractions*. The purpose of this division is to allow users to formulate complex termination conditions by combining primitives with the help of contractions. The supported primitives are:

Fitness threshold This condition terminates the execution of the GA once the best (or average) fitness in the population exceeds a set threshold. It is represented by the class `FitnessThreshold<T>`.

Maximum number of generations This condition terminates the execution of the GA after a set number of iterations is performed. It is represented by the class `MaxNumberOfGenerations<T>`.

[8] Beware, the thread which calls the `run()` method is suspended until the execution finishes.

Termination date This condition terminates the execution of the GA after a set date and time comes to pass. It is represented by the class `AfterDate<T>`.

Although every of the listed primitive objects can be used as a stand-alone termination condition, it is desirable to combine primitives using contractions. The library supports three basic contractions based on the fundamental logical operations. They are represented by classes `NotCondition<T>`, `AndCondition<T>` and `OrCondition<T>`. To simplify their usage, the library overloads common Swift Boolean operators `!`, `&&`, `||` in order to make them compatible with any descendants of `TerminationCondition<T>`. This is illustrated in Listing 11.

```
// Terminate after the best fitness exceeds 0.6 or 200 generations pass.
let termination1: TerminationCondition<MyChromosome>
    = FitnessThreshold(0.6) || MaxNumberOfGenerations(200)

// Terminate if the best fitness does not reach 0.2 after the first
// 100 generations. Otherwise, terminate after 5000 generations.
let termination2: TerminationCondition<MyChromosome>
    = (!FitnessThreshold(0.2) && MaxNumberOfGenerations(100))
      || MaxNumberOfGenerations(5000)
```

Listing 11: Example definitions of termination conditions.

3.5.2 Event Hooks

Although the execution of the GA is synchronous on the level of individual runs, the library allows its users to directly respond to certain important events for the purposes of monitoring and logging. This is achieved by the application of a simple event-driven model.

Every event of interest declares a *hook*. Hooks are optional function pointers, which can be set by the users to introduce custom logic into the evaluation scheme. By default, all hooks are unset when the GA is initialized. If the `DEBUG` macro is defined at the time of compilation, hooks are initialized with functions which print log messages to the standard output. Hooks for the following events are supported:

Run started This event occurs after the `run()` method is called, but before the first population of random individuals is generated. It is represented by the field `hookRunStarted`.

Run finished This event occurs after the termination condition stops the execution of the algorithm, just before the `run()` method returns. It is represented by the field `hookRunFinished`.

Generation advanced This event occurs after every iteration of the GA. At this point, all individuals are evaluated and values such as the best or average fitness of the generation can be accessed. It is represented by the field `hookGenerationAdvanced`.

3.6 Extensions

In this section, several important extension functions of the library are described. These extensions are useful in practical applications as they mainly serve auxiliary purposes, easing the operation of the code, which interacts with the library and its components.

3.6.1 Elitism

In practical applications, it is desirable to ensure that the quality of the solutions produced by the GA does not decrease over consecutive iterations. If the quality is defined as the highest fitness value of the individuals in the current generation, such effect can be simply achieved by copying the fittest individual from the current generation to the offspring generation before transitioning to the next iteration of the GA. This approach is known as *elitism*.

The library provides a dedicated implementation of elitism represented by the class `Elitism<T>`, which is a simple restriction of the reproduction operator to the best individuals of the generation. For the best results, it is recommended that the class is directly referenced in the α-tree of a genetic algorithm instance.

3.6.2 Persistence

To allow continued operation of the GA, the library supports serialization and deserialization of object values. This consequently allows users to persist the state of the GA, terminate their application and restart its execution later on at the same point. Data is serialized into the JavaScript Object Notation format (JSON), which is abstracted by the `SwiftyJSON` framework, one of the dependencies of the library.

In the code, every type that supports persistence is declared to conform to the `PersistentType` protocol, which requires it to provide additional encoding and decoding procedures. For user convenience, the majority of types provided by the library already conforms to this protocol. Only those types which are dependent on user subclassing often offer two protocols (or base classes), so that users can ultimately decide whether to support object persistence in accordance with their needs.

A good example of this division can be found in range-initialized arrays. In addition to the protocol `RangeInitializedArray<T>`, a persistent protocol `PersistentRangeInitializedArray<T>` is declared. The first protocol is a parent object to the latter. The only difference is that in the persistent version, type `T` is required to conform to `PersistentType`. In return, the entire persistent version of the range-initialized array also conforms to `PersistentType`, delegating serialization calls to its elements.

3.6.3 Entropy Generators

Entropy generators are objects used to introduce non-deterministic behavior into the GA. In their essence, they represent a generalization of the conventional pseudo-random number generators (PRNG), extending the range of types produced beyond decimal numbers.

All entropy generators provided by the library conform to the protocol `EntropyGenerator`, which requires types to be capable of generating pseudo-random floating-point decimals in the $[0; 1]$ interval. Any type can be randomly generated in a similar way, provided that it conforms to the `Randomizable` protocol. The library provides such extensions for many Swift primitive types including `Int`, `Double`, `Float`, `Bool` and enumerations, which can opt into this mechanism by conforming to the `Discrete` protocol as shown in Listing 5.

The library offers abstractions of random generation mechanisms provided by the standard library, most notably the `arc4random()` function (represented by `ArcGenerator`) and the `drand48()` function (represented by `DrandGenerator`). In addition, a custom implementation of the Mersenne Twister PRNG was translated from an existing Python implementation (represented by `MersenneTwister`).

4. Usage Demonstration

This chapter contains several demonstrations of the application of the library on practical problems with increasing difficulty. All of the presented examples are included in the library distribution package. To ensure that the mentioned results can be replicated, all instances of genetic algorithms use seeded entropy generators.

4.1 Trivial Example

The first example is a very trivial problem. It is defined as follows: *Given all bit strings of length between 10 and 100 characters, find the string which maximizes the number of ones.* Although the optimal solution is clearly a string of 100 ones, the simplicity of the problem is ideally suited for demonstration of the individual components of the library.

4.1.1 Chromosome and Fitness

The domain space is a finite set. Its points can be characterized as range-initialized arrays of Boolean values with the initialization interval $[10; 100]$, which is declared analogically to the array used in solving the Knapsack Problem (see Listing 1). Since range-initialized arrays already support basic genetic operators, they can be used as chromosomes in the GA.

To evaluate and compare the quality of chromosomes, a fitness function is required. For the purposes of this simple example, the fitness function can be defined as

$$f(s_1, s_2, \ldots, s_n) = \frac{1}{100} \sum_{i=1}^{n} s_i \tag{4.1}$$

where $\{s_i\}_{i=1}^{n}$ are the bits and $n \in [10; 100]$ is the length of the chromosome. A simple implementation of a sequential evaluator using this function is shown in Listing 12.

```
1 class MaxOneEvaluator: SequentialEvaluator<MaxOneChromosome> {
2     override func evaluateChromosome(chromosome: MaxOneChromosome)
3         -> Fitness {
4             let numberOfOnes = chromosome.array.reduce(0) {
5                 $0 + ($1 ? 1 : 0)
6             }
7             return Fitness(numberOfOnes) / Fitness(100)
8     }
9 }
```

Listing 12: Example of a sequential evaluator for the MAX-ONE Problem.

4.1.2 Algorithm

With both the chromosome data structure and the fitness function defined, the only remaining step is to declare and configure the instance of the GA before a run can be started.

```
let elitism = Elitism<MaxOneChromosome>()
let reproduction = Reproduction<MaxOneChromosome>(RandomSelection())
let mutation = Mutation<MaxOneChromosome>(RouletteSelection())
let crossover = OnePointCrossover<MaxOneChromosome>
    (TournamentSelection(order: 5))

let alg = GeneticAlgorithm<MaxOneChromosome>(
    generator: MersenneTwister(seed: 4242),
    populationSize: 200,
    executeEveryGeneration: elitism,
    executeInLoop: (Choice(reproduction, p: 0.5)
        ||| Choice(mutation, p: 0.3) ||| Choice(crossover, p: 0.2)),
    evaluator: MaxOneEvaluator(),
    termination: (MaxNumberOfGenerations(1000) || FitnessThreshold(1))
)

// Execute the algorithm.
alg.run()
print(alg.population.bestIndividual!.chromosome)
```

Listing 13: Example of the GA definition for the MAX-ONE Problem.

To clearly explain the syntax of the `GeneticAlgorithm<T>` class initializer, the algorithm shown in Listing 13 has the following properties:

- The number of individuals in every generation is 200.
- Elitism is used to preserve the best chromosome.
- The algorithm terminates after 1000 iterations or after the highest fitness value reaches 1.0.
- The β-tree contains a single chance node:
  - With the probability 0.5, apply the reproduction operator on a random individual.
  - With the probability 0.3, apply the mutation operator on an individual selected by the roulette selection.
  - With the probability 0.2, apply the one-point crossover operator on the winners of two randomized tournaments, each containing five contestants.

When executed, the presented algorithm performs 757 iterations before reaching the best fitness value 1.0 and yielding the optimal solution consisting of

100 ones. On the experimental computer[1], the evaluation of the algorithm took approximately 5.6 seconds.

To further increase its speed, it is possible to utilize parallelization of fitness evaluation as described in Section 3.2.2. By substituting the line no. 13 of Listing 13 with `ParallelEvaluator() { _ in MaxOneEvaluator() }`, the library creates a separate evaluator instance for every CPU core, instead of sharing a single instance among all cores.

After this modification, the number of performed iterations remains the same, however, the total execution time decreases to 4.2 seconds. Note even though the experimental computer has 8 core CPU, such a small decrease in evaluation time is acceptable due to the fact that the only parallelized part of the algorithm is the evaluation of the fitness function, which in this particular case does not represent a significant portion of the processing time. The convergence of fitness values is plotted in Figure 4.1.

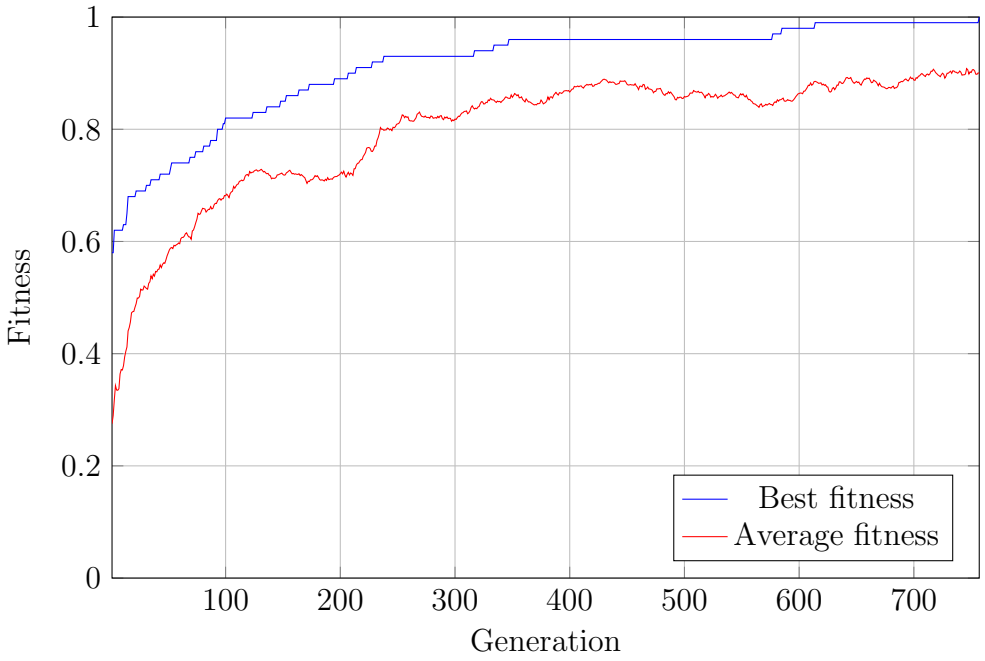


Figure 4.1: Fitness convergence chart of the GA from Listing 13.

4.2 Self-driving Car Simulation

The second example can be considered slightly more complex and closer to practical applications than the MAX-ONE Problem. Suppose that there is a robot car capable of navigating in a simulated two-dimensional environment (illustrated by Figure 4.2) which contains a closed curve resembling a *race track*. The objective is to find a way to steer the car, so that it discovers the track and follows its contour. In real-world applications, this objective would be an analogy to keeping a car or a car-like drone within the bounds of a marked road.

[1]The experimental computer was Apple Mac mini (model *Late 2012*) with Intel Core i7 CPU (2.3 GHz) and 8 GB RAM (1600 MHz DDR3).

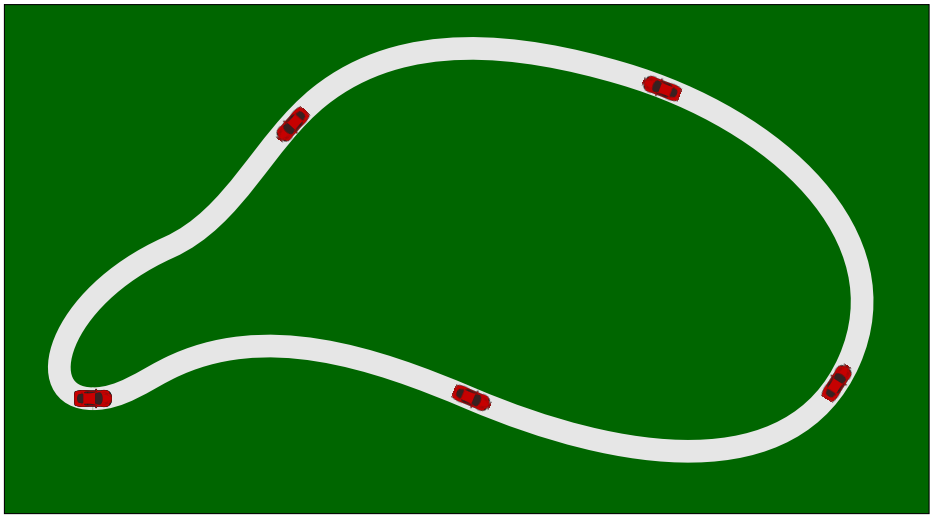


Figure 4.2: Illustration of the self-driving car simulation environment.

The simulated car is controlled by two parameters: the *steering* and the *acceleration*. Modelled after physical driving control systems like the steering wheel and the accelerator pedal, changes in these parameters cannot influence the heading and the velocity of the car directly. Instead, the control parameters affect car's heading and velocity gradually with respect to the time, behaving more like their first derivatives.

An event loop operates in the simulated environment, evaluating all variables periodically with a sufficient[2] frequency. This event loop is responsible for periodically altering the position of the car with respect to its instantaneous velocity and recalculating the instantaneous velocity with respect to the latest value of the acceleration control parameter. A similar process serves to adjust the heading of the car. However, while the acceleration is a continuous decimal parameter with values chosen within a set interval, the steering parameter is fundamentally different, being a discrete choice between set values:

Hard left Alter heading by $+\frac{1}{2}\pi$ radians with respect to the driver's viewport.

Left Alter heading by $+\frac{1}{4}\pi$ radians with respect to the driver's viewport.

Neutral Maintain current heading.

Right Alter heading by $-\frac{1}{4}\pi$ radians with respect to the driver's viewport.

Hard right Alter heading by $-\frac{1}{2}\pi$ radians with respect to the driver's viewport.

To recognize the race track in the environment, the robotic car is equipped with a set of simulated real-time detectors, which are positioned in a way to approximate the viewport of a real car driver. There are three detectors located in front of the car, one under the car and one behind it. All detectors are capable of producing Boolean values, signifying whether the track is located on their exact position at the time of measurement. This configuration approximates thresholding techniques frequently used in detectors of real self-driving car prototypes.

[2]For the experiments, the frequency of the event loop has been set to 40 Hz.

In all simulations, the environment has square shape with the side equal to 10 kilometers. If at any point throughout the simulation the center of the car leaves the bounds of the environment, the simulation terminates. The simulation also ends if set maximum duration is exceeded. The simulated properties of the car (such as dimensions, maximum acceleration, etc.) mimic[3] those of *Audi R8 5.2 V10 FSI Quattro (model 2010)*.

The race track is non-deterministically generated at the beginning of the simulation. First, 30 to 50 points are chosen within the environment at random. These points are in fact selected from a slightly smaller padded rectangle within the environment bounds in order to prevent the generated track from accidentally leading the car to the edges of the environment. In the next step, the convex hull of the selected set of points is calculated by the Melkman Algorithm. [7] The points of the convex hull are transformed to a Bézier curve by simple interpolation utilizing cubic Catmull-Rom splines. [8] Lastly, the thickness of the track curve is increased to 5 meters in order to improve tolerance in subsequent hit-testing.

4.2.1 Control Program

The control program of the car is a dedicated real-time software, which periodically interacts with the event loop of the environment in order to determine the values of control parameters based on the outputs of the on-board detectors.

To demonstrate how the presented library can be used in conjunction with other Swift components, it was decided that the car is to be controlled by a three-layer feedforward neural network with common sigmoid activation function (for definition, see Section 2.2). The input layer of the network is comprised of 5 nodes corresponding to binary read-outs of the on-board detectors, the hidden layer contains 10 nodes and the output layer contains 2 nodes which correspond to the control parameters of the car.

While the acceleration parameter is directly equal[4] to the output of its respective node, the steering parameter uses thresholds to determine the change in heading.

4.2.2 Chromosome and Fitness

It is possible to encode the neural network described in the previous section as a real vector from d-dimensional space, where d denotes the number of interunit connections and the components of the vector correspond to weight coefficients of such connections. By assuming that all connections between nodes from consecutive layers exist (substituting the weight 0 for non-existing connections), a fixed length of the vector can be ensured. The neural network can thus be characterized by a range-initialized array of real numbers with the initialization interval $[d; d]$. Accounting for 5, 10 and 2 nodes in the input, hidden and output layer respectively along with the bias parameters, $d = 6 \cdot 10 + 11 \cdot 2 = 82$.

[3]The simulation limits the acceleration and the instantaneous velocity of the car to match the capabilities of its real-world equivalent. However, it is worth noting that the restrictions are exerted unrealistically by clamping numerical values.

[4]The acceleration parameter is artificially limited to better approximate physical limitations of the car.

To evaluate and compare the quality of control programs, a simple simulated test is performed. Prior to the test, a random race track is generated and positioned within the environment. The car is placed in the center of the simulation with random heading and velocity. Since the race track is a closed curve, the car's initial straight-line trajectory intersects it in at least one point. The control program of the car is expected not to interfere significantly with the steering of the car up to this point. However, after reaching it, the control program should take action to prevent the car from leaving the track.

The described test scenario is executed multiple times, in each instance with different random initial conditions. Throughout every test, various parameters of the car are monitored and recorded by the event loop of the simulation. Every test is terminated if one of two conditions is satisfied:

1. the center point of the car leaves the bounds of the environment, or
2. the maximum duration of the simulation (1 hour) is exceeded.

A crucial parameter for the calculation of fitness values is the *total distance driven over the race track* (denoted $\hat{d}$). Upon initialization, the distance is set to zero. In every iteration of the event loop, the traveled distance Δd is calculated from the instantaneous velocity and the event loop frequency. If the detector located in the middle of the car reports contact with the race track, the calculated distance Δd is added to $\hat{d}$, otherwise $\hat{d}$ remains unchanged. Clearly, $\hat{d}$ has only nonnegative values. For the purposes of the simulation, $\hat{d}$ has an upper bound in the distance traveled at the highest allowed velocity for the maximum duration of the simulation, which is $d_{max} = 199,980$ m/s. At this point, the fitness function can be defined. The fitness function of the control program is evaluated as

$$f(\hat{d}_1, \hat{d}_2, \ldots, \hat{d}_5) = \frac{1}{5 \cdot d_{max}} \sum_{i=1}^{5} \hat{d}_i \tag{4.2}$$

where $\hat{d}_1, \hat{d}_2, \ldots, \hat{d}_5$ denote the total distances driven over the race track calculated in tests 1 through 5 respectively. All distances are specified in meters.

Clearly, the presented fitness function favors control programs which manage to keep the car on the track more than programs which do not follow it very well or veer off it eventually. In addition, since the function operates on distances, control programs are motivated to discover the race track as quickly as possible and to maximize the distance traveled over it. An implementation of a sequential evaluator using this function is illustrated in Listing 14.

In the evaluator, the control program along with the neural network itself[5] is encapsulated in an instance of the `NetDriver` type, which is conformant to the `CarDriver` protocol that formalizes requirements on automated car controllers. Since the evaluator is fully self-contained, it enables parallelization through the `ParallelEvaluator<T>` type. To visualize the simulation, the `CarSimulation` initializer can be called in the *verbose*, which generates a MATLAB script capable of displaying a plot of the car's position in time.

[5]The implementation of the FFNN was provided by the *Swift AI* open-source project, which is available online: `https://github.com/collinhundley/Swift-AI`

```
class CarEvaluator: SequentialEvaluator<CarChromosome> {
    let sim = CarSimulationEnvironment()
    let maxDuration = NSTimeInterval(30)
    let attempts = 3
    override func evaluateChromosome(chromosome: CarChromosome)
        -> Fitness {
            sim.reset()
            sim.randomizeCurve()
            sim.controlProgram = NetDriver(net: chromosome.toFFNN())
            var results = [Fitness]()
            for _ in 1...attempts {
                sim.randomizeCar()
                let outcome = sim.run(maxDuration: maxDuration)
                results.append(Fitness(outcome.timeSpentOnTrack)
                    / Fitness(maxDuration))
            }

            return results.average
    }
}
```

Listing 14: Implementation of the self-driving car evaluator.

4.3 Automated QWOP Player

The third and final usage demonstration of the presented library is closely related to genetic programming techniques proposed in [9]. In the referenced publication, researchers describe their attempts at training artificial programs in playing an online computer game, achieving scores comparable to or exceeding those of human players. This section is dedicated to replicating parts of their results.

4.3.1 The QWOP Game

QWOP (shown in Figure 4.3) is a popular online game, available for free at Foddy.net. [10] In the game, the player controls movements of an Olympic athlete during a sprint race. The objective is to reach the longest possible distance, terminating at the 100-meter mark. If at any point throughout the race the head or any of the hands of the athlete come into contact with the ground, the athlete loses his balance, falls and the game is over. The control scheme of QWOP is very simple. By pressing keys Q, W, O, P on the keyboard (hence the name of the game), the player controls movements of different muscle groups within the athlete's body. Keys Q and W move forward the left and the right thighs and keys O and P move backward the left and the right calves respectively.

In spite of the simplicity of the game goals and the straightforward control scheme, QWOP is well-known for its notorious difficulty. This is mainly due to its "ragdoll physics" engine, which heavily oversimplifies the mechanics of th simulated runner to the point where certain behaviors might seem unintuitive or even unpredictable. [11] The biggest challenge of the game can be described as

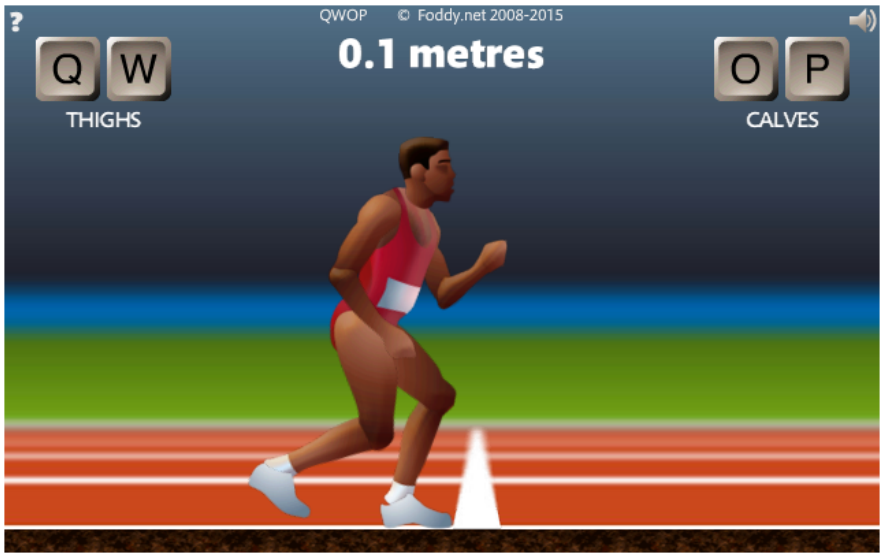


Figure 4.3: The QWOP online game. [10]

to devise a repeatable strategy to achieve and maintain precise synchronization of key presses, receiving only limited sensory feedback from the game.

4.3.2 Previous Work

QWOP has been mentioned in several publications, mostly in relation to machine learning. The challenge of the game has been subject of study primarily because of its appearent similarity to the problem of evolving bipedal gaits in physical robots and other cybernetic applications.

To describe QWOP strategies, the authors of [9] have used string encodings. Since one of these encodings is used[6] in this demonstration, its definition is included in this section. The encoded strings represent sequences of instructions to the player without any regards to the state of the game. The encoding uses symbols "`q,Q,w,W,o,O,p,P,+`". A capital letter represents pressing the corresponding key on the keyboard, a lowercase letter represents a key release. The "`+`" symbol stands for a delay in which the current state of inputs is maintained for 150 milliseconds. [9]

Upon interpretation, the encoded string is read from the left to the right, executing one instruction at a time. When the end of the string is reached, the interpretation starts again from the beginning. An example of a strategy encoded in this way is shown in Listing 15.

```
QO+qPW+wpo+QPW+wO+qp+P+Q+++qp+QPW+wo+qp+POQ+q+W+Qp+qwo
```

Listing 15: Example encoded QWOP game strategy, which translates to *"Press Q and O, hold them for 150ms, release Q, press P and W, hold for 150ms, release W, P and O, wait..."* [9]

To consistently interpret QWOP strategies encoded into strings, the authors

[6]The used encoding is the *Encoding 1*, which was originally proposed by [12].

of [9] have used automated Java program called the *Qwopper*, which has been originally developed by [12]. The application is comprised of three components:

Strategy interpretter The interpretter is responsible for creating artificial user inputs for the QWOP game in compliance with a given strategy string.

Control interface The control interface is a user application, which serves users to configure and test the interpretter.

GA engine By default, Qwopper includes its own implementation of the GA, which has been used to produce strings by means of genetic programming.

Apart from blindly simulating user inputs, the interpretter component of Qwopper includes a basic OCR algorithm (illustrated in Figure 4.4), which is capable of determining the current state of the game (*running* or *paused*) and the distance traveled by the athlete. Combining this information with the duration of the run, Qwopper can periodically estimate the instantaneous velocity of the runner achieved by a given strategy.

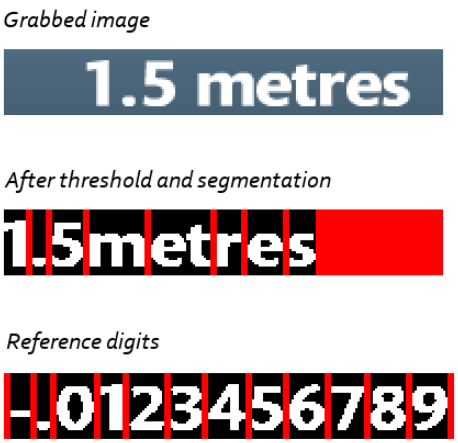


Figure 4.4: Illustration of the OCR process used by the Qwopper software. [12]

It is worth noting at this point that Qwopper already has all necessary components to generate, evaluate and recombine QWOP strategy strings on its own. However, for the purpose of demonstration of the GA provided by the presented library, its functions are reduced to only serve as a strategy string evaluation tool.

4.3.3 Interfacing Qwopper with Swift

asd

5. Conclusion

5.1 Deployment

TODO

5.2 Applications

TODO

5.2.1 Teaching

TODO

5.2.2 Portable Applications

TODO

Bibliography

[1] John H. Holland. *Adaptation in Natural and Artificial Systems.* MIT Press, Cambridge, MA, USA, 1992.

[2] Brad L. Miller, Brad L. Miller, David E. Goldberg, and David E. Goldberg. Genetic algorithms, tournament selection, and the effects of noise. *Complex Systems*, 9:193–212, 1995.

[3] Kevin Gurney. *An Introduction to Neural Networks.* CRC Press, 1997.

[4] Lance D. Chambers. *Practical Handbook of Genetic Algorithms: Complex Coding Systems, Volume III.* CRC Press, 1998.

[5] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA, 1992.

[6] Kim-Fung Man, Kit Sang TANG, and Sam Kwong. *Genetic Algorithms: Concepts and Designs (Advanced Textbooks in Control and Signal Processing).* Springer, 2001.

[7] Avraham A. Melkman. On-line construction of the convex hull of a simple polyline. *Inf. Process. Lett.*, 25(1):11–12, April 1987.

[8] Edwin Catmull and Raphael Rom. A class of local interpolating splines. *Computer aided geometric design*, 74:317–326, 1974.

[9] Steven Ray, Vahl Scott Gordon, and Laurent Vaucher. Evolving qwop gaits. In *GECCO '14 Proceedings of the 2014 conference on Genetic and evolutionary computation*, pages 823–830, Vancouver, 2014.

[10] Foddy B. QWOP. `http://www.foddy.net/Athletics.html`, 2016. [Online; accessed 11-March-2016].

[11] G. Brodman and R. Voldstad. QWOP Learning, Department of Computer Science CS 229 class project. `http://cs229.stanford.edu/projects2012.html`, 2012. [Online; accessed 11-March-2016].

[12] L. Vaucher. Genetically engineered QWOP (part 1). `http://slowfrog.blogspot.com/2011/03/genetically-engineered-qwop-part-1.html`, 2011. [Online; accessed 11-March-2016].

List of Figures

List of Listings

List of Abbreviations

Attachments