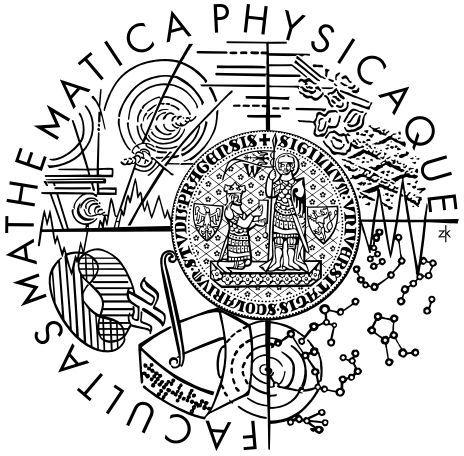
Charles University in Prague
Faculty of Mathematics and Physics

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 techniques, which are inspired by the process of natural selection. In the context of GA, points in the domain space are likened to *individuals* of a biological species. Every individual carries a *chromosome*, which describes its location in the domain space, thus defining its properties and capabilities.

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 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.

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. [1]

2.2 Redistributable Applications

TODO

2.3 Analysis

TODO

2.4 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. [2] 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 [2].

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 interpretted[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

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

the 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 concievable that combinations of various tree nodes can be translated into a language, which is similar to LISP in its architecture. 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 preimplemented 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.

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.

TODO

Comparisons Comparison nodes represent equality and inequality predicates, operating on tuples of value nodes.

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.

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 control the depth and width of the tree, bounding the number of generated structures.

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 name, 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 require chromosomes to conform to other proprietary protocols,

```
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.

in order to operate on them. For instance, the `Mutable` protocol, which is required by the `Mutation` operator. For full listing 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 [3]. 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 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.

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

```
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.

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

Listing 5: Declaration of a randomizable 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 genotype containers with customized behavior and proprietary internal structure, which is most notably exemplified in 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 problem at hand, a common fitness evaluation model is used. In this model, every chromosome is assigned a decimal value from the $[0; 1]$ interval by a *fitness function*, which is heavily dependent on the application and thus specified by the user. The fitness function is encapsulated in an *evaluator* object, which is active for the entire duration of evaluation.

This approach allows users to possibly speed up the evaluation process by minimizing overhead needed to set up and tear down other components and structures required to perform the evaluation, such as simulation environments, inter-process communication sockets, etc. The library supports evaluation in two modes: sequentially or in parallel. While the sequential mode is easy to implement but more time-consuming, the parallel mode requires the internals of the fitness function to be compatible with multi-threaded processing, which may not always be possible.

To demonstrate implementation of a simple sequential evaluator, recall the

chromosome structure[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.

```
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.

In the example, the evaluator is a descendant of `SequentialEvaluator<T>`, which is a common generic base class for all sequential evaluators. Similarly, all parallel evaluators are descendants of the `ParallelEvaluator<T>` class, which instantiates multiple sequential evaluators operating on different threads and manages internal producer-consumer queue to 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 existing subclasses. A good example of such scenario would be an evaluator utilizing 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 parts of the library and avoiding it would prevent users from interacting with such parts.

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

For instance, every subclass of `SequentialEvaluator<T>` can be combined with *a cyclic evaluator*. This type encapsulates the other evaluator, which is called N times for a single evaluation of every chromosome. From the result, M of the best (or the worst) fitness values are selected. The final fitness of the chromosome is the average calculated from the selected values.

3.3 Genetic Operators

Genetic operators are procedures, which are performed on sets of chromosomes during the evaluation of a GA. When operators are applied, two sets of chromosomes are available: *the current generation* and *the offspring generation*. While the first can be only accessed for reading, the latter also supports writing. Every operator can thus select and 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 with different properties and effects. For their detailed description, 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 utilizing only these three in their applications. This section shows the usage of the implemented 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 algorithm.

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.

Since chromosomes are immutable by definition, the reproduction operator requires their underlying 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 themselves.

3.3.2 Mutation

The mutation operator serves the desirable function of introducing occasional variety into a population and restoring its lost diversity. [3] It is fundamentally similar to the reproduction operator, as it copies 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 mutated chromosomes generally resemble their original counterparts, but are not completely identical.

In order to be used, the mutation operator requires the chromosome data structure it works on to conform to the `Mutable` protocol. Every container can thus have its own, slightly different implementation of mutation, which should be explained in its documentation. As an example, this section describes the implementation for containers, which are distributed with the library.

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 arrays and range-initialized arrays, a part is thought to be any item of the array, whereas a part of a tree structure may be any of its subtrees.

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, equivalent 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 customization are:

1. Select two distinct chromosomes in the current generation.
2. Choose pairs of *equivalent 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:

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 `OnePointCrossoverable` protocol.

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 `TwoPointCrossoverable` protocol.

For tree structures, crossover is implemented by selecting two random subtrees rooted in nodes, which are descendants of the same base class (either both nodes are action nodes or both are value nodes specialized in matching generic types). The execution of the operator is performed by swapping pointers to both subtrees.

3.3.4 Custom Operators

By creating descendants of the generic abstract class `GeneticOperator<T>`, users are free to implement and experiment with any operators of their own. This section gives details 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 `apply()` method, which receives a population to work on 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 chromosomes needed for its execution. The selection then returns a list of the selected chromosomes indices, which can be used to access chromosome contents through the `individualAtIndex()` method. 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.

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

```
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.

3.3.5 Pipelines

Pipelines describe the logic of sequential application of operators in the GA. The library includes Swift syntax extensions to facilitate simple customization of operator sequences. Every pipeline resembles a control-flow diagram, it is comprised of individual *nodes*, which are connected by oriented edges. There are two types of nodes:

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

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

Pipelines are defined by custom Swift operators. In order to concatenate pipeline 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 branching nodes. The syntax of both operators can be seen in 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

```
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))
```

Listing 9: Example of pipeline definition.

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 chromsomes. When accessing fitness evaluations, the library uses lazy-loading optimizations, in order to prevent unnecessary sorting and data aggregation. For that reason, selection objects are not required to specify fitness-related dependencies. Instead, additional caluclations 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. [4]

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. [1]

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.

```
1  class MySelection<Chromosome: ChromosomeType>: Selection<Chromosome> {
2      override init() { }
3
4      override func select(generator: EntropyGenerator,
5          population: MatingPool<Chromosome>,
6          numberOfIndividuals: Int) -> IndexSet {
7              // Always select the first 3 individuals.
8              return IndexSet([0, 1, 2])
9      }
10 }
```

Listing 10: Example of custom selection implementation.

3.5 Algorithms

TODO

3.6 Event-driven Approach

TODO

3.7 Extensions

TODO

4. Usage Demonstration

TODO

4.1 Trivial Examples

TODO

4.2 Self-driving Car Simulation

TODO

4.3 QWOP Player

TODO

5. Conclusion

5.1 Deployment

TODO

5.2 Applications

TODO

5.2.1 Teaching

TODO

5.2.2 Portable Applications

TODO

Bibliography

[1] 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.

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

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

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

List of Figures

List of Listings

List of Abbreviations

Attachments