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.

Dedication.

Contents

Introduction

Evolutionary Algorithms

TODO

Genetic Programming

TODO

The Swift Language

TODO

Practical Application

TODO

Structure of This Document

TODO

1. Object-oriented Design

In this chapter, the high-level design of individual components of the library is described.

Koza [1992] <mark>**TODO**</mark>

1.1 Random Genereration

Randomness plays a crucial role in evolutionary algorithms. Since the properties of pseudo-random generators impact the quality of produced solutions significantly, the library gives users full control over the algorithm, which is used to produce random sequences. In object design, this is achieved by simple abstraction.

The functionality of a random number generator is facilitated by *an entropy generator* object. In runtime, only a single instance of such object is created. This instance is then passed on to other components of the library, which require its capabilities. These components access the entropy generator by reference. Users are responsible for instantiating this object, and can thus specify a seed for the generator or choose an algorithm particularly suitable for their application.

For the sake of minimality, entropy generators are only required to produce positive floating-point decimals from the $[0; 1]$ interval. In spite of that, they can be used to generate random values of various types. This mechanism provided that the generated decimals can be mapped onto the type while maintaining uniform distribution of generated values. This is further discussed in section 1.1.1.

1.1.1 Data Structures

Every individual in a generation is repesented by a separate instance of a class. The primary responsibility of such object is to store genetic information, which defines the individual. This information does not need to be held in a homogeneous data structure. In fact, it can be stored in any type suitable for the application. The only requirement on such type is that it can be generated randomly.

<mark>**TODO**</mark>

1.1.2 Randomizable Interface

<mark>**TODO**</mark>

1.1.3 Discrete Interface

<mark>**TODO**</mark>

1.2 Genetic Operators

<mark>**TODO**</mark>

1.2.1   Operator Life Cycle

TODO

1.2.2   Custom Interfaces

TODO

1.3   Selections

TODO

1.4   Algorithms

TODO

2. Library Implementation

This chapter contains technical documentation of individual components of the library, as well as examples of their usage.

2.1 Chromosome Data Structures

In the context of genetic algorithms, *a chromosome* (also known as *genotype*) is a piece of information describing a solution to a problem. Since the nature and representation of such information heavily depends on the application, the library allows full customization of the underlying data structures, which are used to hold chromosomes. The library also provides implementation of some of the most frequently used structures, leaving users free to decide, whether to use a structure supplied by the library, implement a custom one, or combine multiple data structures together.

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,[1]

2. be capable of randomly generating new instances.[2]

This section explains, 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.

2.1.1 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 genetic algorithm 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 ???. **TODO**

2.1.2 Strings

One of the most common ways of storing chromosomes is to encode them as strings of values of a same type, e. g. binary or numeric. Such strings are represented by *range-initalized arrays*.

A range-initialized array is similar to a regular array in many ways. It is a generic list structure, which is capable of holding finite amounts of ordered

---

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

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

homogeneous items. However, at the time of initialization, the number of elements in the array is set to a value, which non-deterministically selected from a given number interval. This allows for more flexibility, since in some applications, it is beneficial to vary not only the contents of the chromosome, but also its size. If this behavior is not required, the array can be configured to a constant length by using any interval of length zero.

A simple usage of range-initialized arrays can be demonstrated on 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 knapsack contents, while not exceeeding its capacity. 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), which is declared in Listing 1.

```swift
struct KnapsackChromosome: RangeInitializedArray {
    typealias Element = Bool
    static let initializationRange = 10...10

    let array: [Element]
    init(array: [Element]) {
        self.array = array
    }
}
```

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 extensions, every instance of range-initialized array automatically conforms to the `ChromosomeType` protocol and supports three basic genetic operators (for definition, see section 2.3). Range-initialized arrays can thus be directly used as chromosomes in genetic algorithms without any further modification.

It is worth noting at this point that strings are **not designed to store 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[3] 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 array, which is merely a part of a greater whole (e. g. number), unfortunate modification of such item could cause

---

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

the chromosome to become undecodable. Instead, the recommended alternative is to use custom types (see section 2.1.4), 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.

2.1.3 Trees

Tree structures are commonly used in applications, which require automatic code generation. In such applications, chromosomes often contain control programs or mathematical formulas, which can be represented by tree graphs. The library allows to store such data in 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 offers two basic 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.

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.

Both types of nodes are easily extensible, allowing users to define their own functions, operations and commands depending on the application. This can be demonstrated on a simple 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 in a 2-dimensional maze. The robot also carries a set of sensors capable of determining whether its front side is facing an obstacle. To auto-generate a control program for such robot, its commands can be formalized as 5 action nodes and the sensor output can be represented by a single Boolean value node. Example of 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 offers various frequently used node types, ready to use:

Constants and Operations Constant nodes contain static values of any type, unchanging during program execution. Operation nodes are generic templates for unary or binary operations applied on arguments, which are represented by other value nodes.

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

```
1  class GoCommandNode: ActionNode {
2      override func perform(interpreter: TreeInterpreter) {
3          guard interpreter.running else { return }
4          // Tell the robot to go forward.
5          // The interpreter contains the current context.
6      }
7
8      override func propagateClone(factory: RandomTreeFactory,
9          mutateNodeId id: Int) -> ActionNode {
10             let clone = GoCommandNode(id: id, maximumDepth: maximumDepth)
11             // This node contains no descendants.
12             return clone
13     }
14 }
```

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

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.

```
1  class RobotProgramFactory: TreeFactory {
2      /* ... */
3  }
4
5  class MyGenerator: MersenneTwister {
6      let robotProgramFactory: RobotProgramFactory
7
8      override init(seed: Int) {
9          robotProgramFactory = RobotProgramFactory(generator: self)
10         super.init(seed: seed)
11     }
12 }
```

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

2.1.4 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, 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 2.3.

Declaration of custom types can be demonstrated on The Hamburger Restaurant Problem, mentioned in the introduction[4] of Koza [1992]. 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 2.1.2, it is much safer and more elegant to declare a dedicated type to hold its information. Such declaration is shown in Listing 4.

```
1  class RestaurantStrategy: Randomizable {
2      let hamburgerPrice: Double
3      let drink: Drink
4      let waiterSpeed: Speed
5
6      init(generator: EntropyGenerator) {
7          // Generate random values.
8          hamburgerPrice = generator.nextInRange(min: 0.5, max: 10)
9          drink = generator.next()
10         waiterSpeed = generator.next()
11     }
12 }
```

Listing 4: Example declaration of custom chromosome type.

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.

---

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

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.

```
1 enum Drink: Discrete, Randomizable {
2     case Water, Wine, Cola
3     static let allValues: [Drink] = [.Water, .Wine, .Cola]
4 }
```

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.

2.2 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[5] for the Knapsack Problem. Suppose the fitness function is defined as

$$f(c_1, c_2, \ldots, c_{10}) = \begin{cases} 0 & \text{if } \sum_{i=1}^{10} c_i s_i > S_{max} \\ \sum_{i=1}^{10} c_i v_i / \sum_{i=1}^{10} v_i & \text{otherwise} \end{cases} \qquad (2.1)$$

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

---

[5]The Knapsack Problem is defined in section 2.1.2. For chromosome structure, see Listing 1

the Boolean values of the chromosome. A simple implementation of a sequential evaluator using this function is shown in Listing 6.

```swift
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.

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.

2.3 Genetic Operators

Genetic operators are procedures, which are performed on sets of chromosomes during the evaluation of a genetic algorithm. 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 select and read an arbitrary number of chromosomes from the current generation, and is expected to insert at least one chromosome in the offspring generation.

The selection of chromosomes is carried out through *selection objects*, which are specified as configuration parameters when the operators are created. There are various types of selections, each with different properties and effects on the operators. For their detailed description, see section 2.4.

The library offers implementation of three common genetic operators: *reproduction*, *mutation* and *crossover*. However, users are by no means limited to only these three. By creating descendants of the generic class `GeneticOperator<T>`, users are free to implement and experiment with any operators of their own.

2.3.1   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 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 chromosome data structures are immutable by definition, the reproduction operator requires them 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.

2.3.2   Mutation

The mutation operator serves the desirable function of introducing occasional variety into a population and restoring its lost diversity. Koza [1992] 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 should generally resemble their original counterparts, while not being 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.

2.3.3 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[6] 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.

---

[6]Each crossover operator has a dedicated Swift 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.

2.3.4 Pipelines

==TODO==

2.4 Selections

==TODO==

2.4.1 Roulette Selection

==TODO==

2.4.2 Rank Selection

==TODO==

2.4.3 Tournament Selection

==TODO==

2.4.4 Extensions

==TODO==

2.4.5 Optimizations

==TODO==

2.5 Algorithms

==TODO==

2.6 Event-driven Approach

==TODO==

2.7 Extensions

TODO

3. Usage Demonstration

TODO

3.1 Trivial Examples

TODO

3.2 Self-driving Car Simulation

TODO

3.3 QWOP Player

TODO

Conclusion

Deployment

==TODO==

Teaching

==TODO==

Applications

==TODO==

Bibliography

John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA, 1992. ISBN 0-262-11170-5.

List of Figures

List of Listings

List of Abbreviations

Attachments