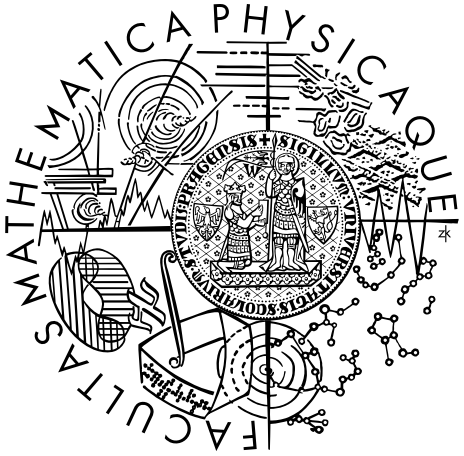
Charles University in Prague
Faculty of Mathematics and Physics

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

Introduction **3**

1 Object-oriented Design **4**
1.1 Random Genereration . . . 4
1.1.1 Data Structures . . . 4
1.1.2 Randomizable Interface . . . 4
1.1.3 Discrete Interface . . . 4
1.2 Genetic Operators . . . 4
1.2.1 Operator Life Cycle . . . 5
1.2.2 Custom Interfaces . . . 5
1.3 Selections . . . 5
1.4 Algorithms . . . 5

2 Library Implementation **6**
2.1 Chromosome Data Structures . . . 6
2.1.1 Strings . . . 6
2.1.2 Trees . . . 7
2.1.3 Custom Types . . . 9
2.2 Genetic Operators . . . 11
2.2.1 Reproduction . . . 11
2.2.2 Mutation . . . 11
2.2.3 Crossover . . . 11
2.3 Selections . . . 11
2.3.1 Roulette Selection . . . 11
2.3.2 Rank Selection . . . 11
2.3.3 Tournament Selection . . . 11
2.3.4 Extensions . . . 11
2.3.5 Optimizations . . . 11
2.4 Algorithms . . . 11
2.5 Event-driven Approach . . . 11
2.6 Extensions . . . 12

3 Usage Demonstration **13**
3.1 Trivial Examples . . . 13
3.2 Self-driving Car Simulation . . . 13
3.3 QWOP Player . . . 13

Conclusion **14**

Bibliography **15**

List of Figures **16**

List of Listings **17**

List of Abbreviations **18**

Attachments **19**

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] **TODO**

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.

TODO

1.1.2 Randomizable Interface

TODO

1.1.3 Discrete Interface

TODO

1.2 Genetic Operators

TODO

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 is a technical documentation of individual components of the library.

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. Apart from that, the library also provides implementation of the most frequently used data structures. This leaves users free to decide, whether to use a structure supplied by the library, implement a custom one, or combine multiple data structures together.

The question of efficiently encoding solutions into chromosomes is a separate topic, worthy of further investigation. The library is merely intended as a tool to build containers for such chromosomes, once the user's mind is set on a specific method of data representation. This section explains, how to achieve some often used configurations, and how to define custom data types for storing proprietary information.

2.1.1 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 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 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.

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 supports three basic genetic operators (for definition, see section 2.2) and can generate random instances of itself. Range-initialized arrays can

```
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.

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 Booleans, it can be encoded as a binary string, portions of which would be later interpretted[1] as integers by the application.

While this approach succeeds in encoding the chromosome into a binary string, 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 in 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 2.1.3), 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.

2.1.2 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.

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

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 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 commands, 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.

```
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.

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 elements 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 a static value of any type, unchanging during program execution. Operation nodes are generic templates for any unary or binary operation applied on a set of value nodes.

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

```
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.

2.1.3 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 Swift reference type can become a chromosome data structure, if it conforms to the `Randomizable` protocol. 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 allows users to customize the way mutations are applied to the internals of the data structure. For full listing of such protocols, see section 2.2.

Declaration of custom types can be demonstrated on The Hamburger Restaurant Problem, mentioned in the introduction of Koza [1992] (for the purposes of this work, the example has been slightly altered). 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 2.1.1), it is much safer and more elegant to declare a dedicated type to hold its information. Such declaration is shown in Listing 4.

```
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.

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 the user 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 in the data structure.

Lastly, types which are capable of listing all their possible values in a finite sequence can utilize the `Discrete` protocol, which implements the `Randomizable` protocol by randomly selecting values from a discrete uniform distribution of all values. A good demonstration of this is the `Drink` type, which is declared as a Swift enumeration in Listing 5.

```
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 as syntax sugar 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 Genetic Operators

TODO

2.2.1 Reproduction

TODO

2.2.2 Mutation

TODO

2.2.3 Crossover

TODO

2.3 Selections

TODO

2.3.1 Roulette Selection

TODO

2.3.2 Rank Selection

TODO

2.3.3 Tournament Selection

TODO

2.3.4 Extensions

TODO

2.3.5 Optimizations

TODO

2.4 Algorithms

TODO

2.5 Event-driven Approach

TODO

2.6 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

1 Range-initialized array used to solve the Knapsack problem. . . . 7
2 Example implementation of the GO command action node. 8
3 Tree factory declared in an entropy generator subclass. 9
4 Example declaration of custom chromosome type. 10
5 Declaration of a randomizable type through a discrete listing of values. 10

List of Abbreviations

Attachments