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

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

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 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 homogoeneous values. 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 interval. This allows for more flexibility, since in some applications, it is beneficial to vary not only the contents on 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 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 of size $[10; 10]$, which is declared in Listing 1.

In a similar way, range-initialized arrays can store integer tuples to encode permutations or floating-point decimals to describe weights of neural networks. Thanks to Swift extensions, every 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 thus be directly used as chromosomes in genetic algorithms.

```
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.

2.1.2 Trees

TODO

2.1.3 Custom Types

TODO

```
class Vector3D: Randomizable {

    let x: Double
    let y: Double
    let z: Double

    init(generator: EntropyGenerator) {
        // We only consider vectors
        // in a 3-dimensional cube from -10 to 10.
        let range = -10..10

        // Generate random values.
        x = generator.nextInRange(range)
        y = generator.nextInRange(range)
        z = generator.nextInRange(range)
    }

}
```

Listing 2: Example from external file

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

List of Abbreviations

Attachments