Computational Intelligence, Simulation, and Mathematical Models Group

CISMM-21-2002 May 19, 2015

JEvolution: Evolutionary Algorithms in Java

Technical Report
JEvolution V0.98

Helmut A. Mayer

helmut@cosy.sbg.ac.at

Department of Computer Sciences

University of Salzburg

Correspondence to:

Helmut Mayer
Universität Salzburg
Fachbereich Computerwissenschaften
Jakob–Haringer–Straße 2
A–5020 Salzburg
AUSTRIA

Telephone: +43-662-8044-6315
FAX: +43-662-8044-611

JEvolution: Evolutionary Algorithms in Java

Helmut A. Mayer
Department of Computer Sciences
University of Salzburg
A–5020 Salzburg, Austria

Abstract

We present the basic ideas and structure of `JEvolution`, a compact Java package for applications using Evolutionary Algorithms (EAs) as an optimization tool. `JEvolution` has been written with the intent to suit the needs of both, the experienced Evolutionary Computation (EC) expert, and the novice to this exciting field of computer science. Hence, `JEvolution` can be parametrized in many ways, but it can also be used by only writing a few lines of code. Inevitably, there are two main tasks an evolutionary engineer has to work on, when constructing an EA: problem encoding and fitness function. While the encoding of the problem is supported by `JEvolution` native, ready–to–use Chromosomes, the fitness function has to be provided by the applicationist, as `JEvolution` is (not yet) capable of knowing details about the problem it is working on. The usage of `JEvolution` is described by means of a simple test program searching for the minimum of a paraboloide function.

1 Introduction

`JEvolution` is a Java package for EAs (Goldberg, 1989; Koza, 1992; Michalewicz, 1991; Schwefel, 1995; Mitchell, 1996) which has been implemented with a focus on (presumably) correct object–oriented design. In less technical words that means that execution speed has been of minor importance during development. However, it is well known that an EA spends most of the time evaluating the fitness of individuals which is not an intrinsic part of `JEvolution` (the user has to implement the `Phenotype` Interface in order to provide problem–specific code).

The current version of `JEvolution` V0.98[1] is of β–quality having been reliably used in a system for *Evolutionary Feature Selection* (Mayer et al., 2000; Mayer and Somol, 2000). It also serves as the evolutionary engine in the following frameworks: *netJEN* and *netEVO* (evolution of Artificial Neural Networks), *fuzJEN* (evolution of Fuzzy Controllers), *evAlloc* (generic solver for allocation and scheduling problems), *GOjen* (evolution of artificial Go players), and *DIGO* (digital organisms).

The current features of `JEvolution` are:

- Lean and compact Java package
- Supports bitstring, integer, permutation, real encoding (native chromosomes)
- Supports tree chromosomes (native) for Genetic Programming including a basic set of function and terminal nodes

[1] Please, do not be bothered by our restrictive approach to version numbering.

- Supports evolution of chromosome length (bitstring encoding) by means of non–homologous crossover (experimental)
- User may provide custom chromosomes
- Genotype may comprise an arbitrary number of different chromosomes
- Chromosome shuffling
- Records a "Star Pool" (containing all intermediate best individuals)
- Individuals may be stored to and loaded from an XML file
- Flexible termination criteria (number of generations, fitness threshold, evolution time)
- Supports splitting of a population into sub-populations
- Handles maximization and minimization problems without fitness scaling
- Supports tournament selection with arbitrary tournament size (native selection method)
- User may provide custom selection method (extending the abstract class Selection)
- Simple Java Interface for problem–specific code
- Direct access to genotype for Lamarckian evolution or repair methods
- Set up for distributed computation of fitness
- Set up for thread interrupt
- Default reporter for simple statistics on evolutionary progress
- User may provide costum reporter (implementing the Reporter interface)

2 Using JEvolution in an Application

The basic steps to make `JEvolution` work are illustrated by some lines of code from the test program *GATest.java* coming with your `JEvolution` distribution.

Get the JEvolution singleton

```
JEvolution GA = JEvolution.getInstance();
```

Sets up an EA with default parameters for population size, number of generations, number of runs (evolutionary cycles).

Get the associated reporter

```
JEvolutionReporter GAStats = (JEvolutionReporter)GA.getJEvolutionReporter();
```

Sets up the object reporting on EA progress. If you do not want to change default settings, you do not even have to know about the reporter, i.e., you do not need that line at all.

Create the chromosome(s) needed

```
BitChromosome chromX = new BitChromosome();
```

Sets up a `JEvolution` native chromosome (bitstring encoding). All attributes and methods associated with mutation and crossover are part of a chromosome object. With the above line, the chromosome is ready to use, but of course, you may want to change parameters. If you want to change the basic operation of mutation and crossover, you must extend `JEvolution` native chromosomes or the abstract class *Chromosome* with your own code.

Pass the chromsome(s) to JEvolution

```
GA.addChromosome(chromX);
```

Hands your parametrized chromosome to `JEvolution` for future evolution. You can add as much chromosomes (even of different type) to the genotype as you want. The only thing you have to keep in mind is the order of addition, as this knowledge is necessary when decoding the genotype in your implementation of the `Phenotype` interface.

The `JEvolution` default native selection method is *Binary Tournament Selection* (without replacement). The only other selection method *NoSelection* has only been implemented for experimental comparisons. However, you may provide custom selection methods by extending the abstract class `Selection`. The selection method is set by `JEvolution`'s `setSelection()` method.

Register Phenotype class with JEvolution

```
GA.setPhenotype(new ParaboloidPhenotype());
```

Notifies `JEvolution` of the problem–specific code. When having a closer look at the `Phenotype` Interface, you will find three methods associated with fitness evaluation. Note that `JEvolution` creates a separate `Phenotype` object for every individual in the population (just like in real life). This is especially useful for distributed computation of fitness (see below).

First, `doOntogeny()` is called which maps the genotype to the specific phenotype (so here you will have to know about the chromosome order). Second, `JEvolution` calls `calcFitness()` of each individual to be evaluated. If you run your application on a single computer, you do not have to worry about anything, but just provide the fitness function. If you want to distribute computation of fitness functions, you may want to just pass the request for fitness computation to a dispatcher and return from `calcFitness()` without actually having calculated the fitness. Third, after all `calcFitness()` calls have been made by `JEvolution`, fitness values are collected by `getFitness()`. Again, just return the fitness value here for single–computer applications. For distributed systems it is important that `getFitness()` blocks in case of ongoing fitness computation and only returns, when fitness is available. Yes, it is a simple strategy and gives full responsibility to the user coordinating distributed computation which is intended..;-)

Start JEvolution

```
GA.doEvolve();
```

After having set up the genotype and provided the phenotype class, we simply start evolution and wait for the result... Thus, for our simple test problem, we would only need five lines of code to implement an EA. However, this is not the whole truth as we also have to provide the problem–specific code. Please, see *ParaboloidPhenotype.java* and a straight–forward implementation of searching the minimum of the function $f(x, y) = x^2 + y^2$. The use of two chromosomes encoding x and y, respectively, is simply for illustrative purposes.

Note that `JEvolution` can be configured to natively handle minimization problems via `setMaximization(false)`. In this case a solution A having a higher fitness value than a different solution B is considered to be inferior to B.

2.1 Support for Genetic Programming

In directory `Samples/GP` there are two test programs illustrating the implementation of a Genetic Programming application. It centers on the use of JEvolution's `TreeChromosome`, which is built by `ProgramNode`s. A small set of basic function and terminal nodes can be found in the package `evSOLve.JEvolution.gp.nodes`. The programmer has to provide the function and terminal set using `TreeChromosome.addNode()`. The only difference between a function and a terminal node is that the `children` array of the latter is `null` (see the `ProgramNode` API for details). Of course, you may provide custom nodes by simply extending `ProgramNode`.

If you wish to supply your (tree) program with variable values, you have to use `ProgramNode.addValue()`, which puts the variable values in a static array, which can be accessed by terminal nodes representing a variable value, e.g., `VarDouble`. If you want to supply new values, you have to remove the old values first using `ProgramNode.clearValues()`.

There are two types of mutation implemented in the `TreeChromosome`. The mutation process iterates over all tree nodes and performs either tree mutation or node mutation using the mutation rate given. If tree mutation is activated a subtree at a mutation position is substituted with a random subtree generated using the generation method set with the `TreeChromosome` (and depth / 2). With node mutation a random node is mutated calling its `mutate()` method. Note that not all nodes may support mutation (in this case mutation does not change anything).

3 Caveats

`JEvolution` may be interrupted by issuing an `interrupt()` to the thread, where `JEvolution` is running in. Upon receiving an interrupt `JEvolution` finishes computations of the current generation and returns normally (with a smaller number of evaluated generations than originally set by the user).

If evolution is (re)started with a population from a previous run saved to an XML file, the user has to make sure that `JEvolution` is set up in a way it can deal with the restored population. Specifically, the phenotype model, of course, has to fit the individuals in the population.

4 Final Remarks

Though, `JEvolution` is intended to be a lean package, some additions surely will be made. Here is an (incomplete) list of possible extensions:

- More selection methods
- Dynamic population sizes and chromosome lengths

The `JEvolution` distribution *JEvolution.tar.gz* comes with the following parts:

README – Basic technical information

JEvolution.jar – The Java Archive

Doc/ – Directory containing the API documentation created by `javadoc` and this document (*jevolution.pdf*)

Samples/ – Directory containing three test programs using JEvolution to implement an EA, a GA, and a GP application

If you have any comments, suggestions, or more likely bug reports, feel free to contact us at `helmut@cosy.sbg.ac.at`.

5 Acknowledgments

This work has partially been supported by *AKTION Österreich – Tschechische Republik* under grant AKTION 29p7: "Conventional and Evolutionary Construction of Finite Mixture Models for Classification Problems in Remote Sensing".

A Glossary

Application – Java Program using **JEvolution**
Applicationist – Synonym for **User**
Base – Atomic information unit of a **Chromosome**
Chromosome – Part of a **Genotype**
Custom – Java code extending **JEvolution** provided by **User**
EA – Evolutionary Algorithm
GA – Genetic Algorithm
Genotype – Encoded problem solution on one or more **Chromosomes**
GP – Genetic Programming
Individual – A *Phenotype* described by its **Genotype**
JEvolution – Java package for **EAs**
Mutation – Random alteration of a **Base** value
Native – Part of **JEvolution** package
Phenotype – A solution decoded from the **Genotype**
Population – A number of **Individuals**
Selection – Implementation of survival of the fittest
User – Programmer using **JEvolution** for her application

References

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization & Machine Learning.* Addison-Wesley.

Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by means of Natural Selection.* Complex Adaptive Systems. The MIT Press, Cambridge, MA.

Mayer, H. A. and Somol, P. (2000). Conventional and Evolutionary Feature Selection of SAR Data Using a Filter Approach. In *Electronic Proceedings of the 4th World Multiconference on Systemics, Cybernetics, and Informatics (SCI 2000).*

Mayer, H. A., Somol, P., Huber, R., and Pudil, P. (2000). Improving Statistical Measures of Feature Subsets by Conventional and Evolutionary Approaches. In *Proceedings of the Joint IAPR International Workshops SSPR 2000 and SPR 2000*, pages 77–86. Springer.

Michalewicz, Z. (1991). *Genetic Algorithms + Data Structures = Evolution Programs*. Artificial Intelligence. Springer, Berlin.

Mitchell, M. (1996). *An Introduction to Genetic Algorithms*. Complex Adaptive Systems. MIT Press, Cambridge, MA.

Schwefel, H.-P. (1995). *Evolution and Optimum Seeking*. Sixth–Generation Computer Technology Series. Wiley, New York.