siGis-21-2001 June 18, 2001

JEvolution: Evolutionary Algorithms in Java

Technical Report
JEvolution V0.4

Helmut A. Mayer
helmut@cosy.sbg.ac.at

Department of Computer Science
University of Salzburg

Correspondence to:

Helmut Mayer
Universität Salzburg
Institut für 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 Science
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` V0.4 [1] is of β–quality, as it has been reliably used in a framework for *Evolutionary Feature Selection* (Mayer et al., 2000; Mayer and Somol, 2000). It is currently being integrated in *netJEN*, a framework for the evolution of Artificial Neural Networks (ANNs).

The current features of `JEvolution` are:

- Lean and compact Java package
- Supports bitstring-, integer-, permutation, and real encoding (native chromosomes)
- User may provide costum chromosomes
- Genotype may comprise an arbitrary number of different chromosomes
- Chromosome shuffling

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

- Supports tournament selection with arbitrary tournament size (native selection method)
- User may provide costum selection method (implementing the Selection Interface)
- No recomputation of already evaluated individuals (fitness repository)
- 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
- Reports simple statistics on evolutionary progress

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.

Create a JEvolution object

```
JEvolution GA = new JEvolution();
```

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. However, by default the fitness repository is not used. If you decide to use it, you have to request it from the reporter.

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 costum selection methods by implementing the `Selection` Interface. The selection method is set by /tt JEvolution's `setSelection()` method.

Register Phenotype class with JEvolution

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

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 *ParaboloidePhenotype.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` always interprets a solution A having a higher fitness value than a different solution B to be superior to B. Otherwise there is no restriction (besides the type `double`) on the fitness values.

3 Caveats

The fitness repository keeps track of all genotypes evaluated during a single evolutionary run. If the repository is used, `JEvolution` checks, if a genotype of unknown fitness is already in the repository. If it is there, the fitness value is retrieved directly from the repository and the `Phenotype` methods are never called. Thus, if there is any randomness associated with decoding your genotype or

evaluating its fitness, do not use the fitness repository, as it would always assign the same fitness to the same genotype.

The repository may reduce computational cost considerably, however, at the cost of memory. Therefore, some status information on the memory used by the *Java Virtual Machine* (JVM) is reported by `JEvolutionReporter`, when using the repository. Moreover, the repository is built anew with each `JEvolution` run, as test runs (rather surprisingly) have shown that solutions from a previous run are rarely found in a next run. Thus, the repository would grow without any speed advantage. Still, if you are doing "long" evolutionary runs (common with *Genetic Programming*), you could run out of memory even during a single run, when using the repository (of course, there is also some room for future optimization of the repository in `JEvolution`).

Of course, `JEvolution` may be used within a Java thread created by the application. If the application issues an `interrupt()` to the thread, `JEvolution` finishes computations of the current generation and returns normally (with a smaller number of evaluated generations than originally set by the user).

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:

- Tree (*Genetic Programming*) native chromosomes
- More selection methods
- Dynamic population sizes and chromosome lengths
- More statistics by `JEvolutionReporter`

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*)

GATest.java – Small test program

ParaboloidePhenoptype.java – Simple `Phenotype` implementation used by the test program

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 **Chromosome**
Chromosome – Part of a **Genotype**
Custom – Java code extending **JEvolution** provided by **User**
EA – Evolutionary Algorithm
Genotype – Encoded problem solution with one to many **Chromosomes**
JEvolution – Java package for **EAs**
Mutation – Random alteration of a **Base** value
Native – Part of **JEvolution** package
Population – Number of different problem solutions
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.