EVOLUTIONARY METHODS FOR DESIGN, OPTIMISATION AND CONTROL
K. Giannakoglou, D. Tsahalis, J. Periaux, K. Papailiou and T. Fogarty (Eds.)
© CIMNE, Barcelona, Spain 2002

INTRODUCING EALIB - A JAVA EVOLUTIONARY COMPUTATION FRAMEWORK

Andreas Rummler[⋆]
Technical University of Ilmenau, Germany
Department Electronic Circuits and Systems
PO Box 100565, 98684 Ilmenau, Germany
e-mail: arummler@acm.org
web: http://www.inf-technik.tu-ilmenau.de

Gerd Scarbata
Technical University of Ilmenau, Germany
Department Electronic Circuits and Systems
PO Box 100565, 98684 Ilmenau, Germany
e-mail: gerd.scarbata@tu-ilmenau.de
web: http://www.inf-technik.tu-ilmenau.de

Abstract. This article introduces the evolutionary computation framework *eaLib*. After a short introduction, in which the need for EA toolkits is shown, a general-purpose genetic representation of individuals is explained. In the next two sections the authors show the concept of breaking up evolutionary algorithms into several independent operators. These operators are transformed into software components with a common interface and used to create algorithms in a fast and flexible way. The papers concludes with a short outlook on future work.

Key words: Evolutionary Algorithm, Java, Toolkit, Class Library, Framework, Software Component

1 INTRODUCTION AND MOTIVATION

In recent years evolutionary algorithms have become more and more popular for the reason that they *can be* a powerful tool for solving different optimization problems. The accentuation in this sentence lies on the words *'can be'*. The creation of a well-performing evolutionary algorithm is still a problem, although some work has been done in this field.[1] In case the optimization problem to be solved fits into one of the well-studied problem classes, it is easy to look up appropriate publications to find suitable operators and parameters for algorithm creation. But if this is not possible, algorithm development means experimentation with genetic operators.

However, until an overall and general theory of evolutionary algorithms has been developed, something can be done to offer help to engineers and scientists who are going to create EAs: the supply of flexible and easy to use toolkits. One of such evolutionary computation frameworks is *eaLib*, which is introduced in this paper, in association with some considerations on how genetic operators can be fitted into a common structure.

The toolkit introduced in this paper is written in the programming language Java. This language was chosen for several reasons. Java is widely spread and has become one of the major programming languages. The software development kit[2] (SDK) is freely available on a number of different computer platforms. The language is object-oriented and relatively easy to learn. The last point had to be considered because a user of the *eaLib* toolkit is forced to have some programming knowledge in the chosen language. Another important point is the built-in support for multithreading and distributed computing.

The speed at which a new algorithm can be assembled, and the flexibility new operators can be defined with, are of major importance in toolkit design. By using the advantages of object-oriented software design it is possible to keep the tradeoff between easy handling and extensibility that is always present in such toolkits as close as possible.

2 REPRESENTATION AND EVALUATION

Before starting the concrete implementation of a toolkit, some thoughts have to be spent on requirements and architecture. Interesting work on design aspects has been done by Matthew Caryl.[3]

First of all, a general form of genetic representation must be found. Surely there has to be support of the traditional representation forms (binary string and real numbered). These forms are still used in various cases, but in the authors opinion these forms are neither very flexible nor very descriptive. A toolkit relying on these forms alone would exclude other forms of genetic representation from being implemented and used. For this reason a generic and more general form has to be found.

The type of genetic representation used in *eaLib* leans very close against the structure of real creatures known from nature. The representation is shown in figure 1. An instance of the class *Individual* contains exactly one instance of the class *ChromosomeSet*. The chromosome set is a collection of chromosomes, which are able to hold data of any arbitrary type. This can be primitive types like integer or floating point numbers, but more complex data types are also possible, i.e. lists, matrices or trees. For the support of binary representations a special class called *BitVector* is provided. In case that chromosomes of a data type that is not already supported should be used, a user simply has to derive a new class from the abstract base class *Chromosome* and implement the methods for cloning and equality checking.

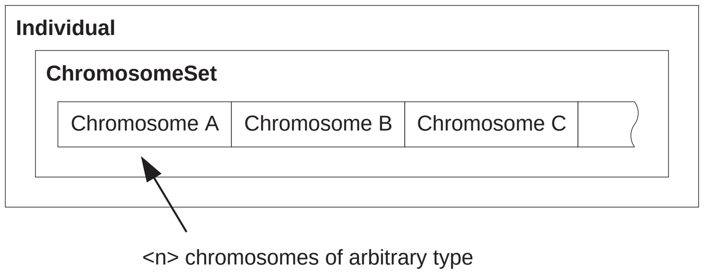

Figure 1: Genetic representation used in *eaLib*

This type of representation is flexible enough to be used for both simple and complex solution representations. An individual containg a simple binary string can be implemented as well as a more complicated form, for instance the one used in messy genetic algorithms.[4]

Beside several informative attributes, like unique ID or age the class *Individual* contains two instances of classes that are much more important: *Score* and *Fitness*.

The class *Score* represents the target objective value. This value is usually defined as a number, so predefined classes for that case are provided. But it is conceivable, that the score of an individual is defined in another way. In algorithms for multiobjective optimization normally several values/criteria for quality indication of individuals are used. For this reason the predefined score definition can be changed by the user by implementing own classes for score representation and comparison.

The fitness is usually defined as a mapping of the target objective value into a non-negative range of values.[5] This is also the definition that the implementation in the toolkit follows.

The evaluation of the target objective value is always problem-specific and has to be done by the user. This is the reason for the expectation of Java programming knowledge of the user. For the purpose of the individual evaluation an abstract base class called *ScoreEvaluation* is provided. The user has to derive an own class from that base class and implement one single method: *evaluate(Individual i).* Within this method the calculation of a valid score and the assignment of the score to the individual is performed.

The fitness evaluation is a quite similiar procedure with the difference that there are several existing approaches for definining a fitness value. Some of these approaches are already provided by *eaLib*, while the user is able to introduce his own functions. Examples for already supported fitness functions are linear and reciprocal scaling,[6] linear[7] and nonlinear ranking scaling.[8]

3 GENETIC COMPONENTS

It is necessary at the beginning of this section to make some remarks concerning the term component. During the last years the term was heavily used in connection with software development without there being any universal definition as to what a component is. The authors follow the definition given in a publication by Claudia Piemont.[9] According to that, a component is a runnable piece of software with a describable functionality. It has a black box character – the information contained within a component is hidden from the user. Access to methods and/or attributes of a component is provided by a standardized interface. This interface is both a communication channel and a plug to which other components can be connected to. The solution to a given task is achieved by creating a net of components that are coupled to each other. This net in its totality is able to solve the given task.

In the attempt to view genetic operators as components and to define a unified interface it is necessary to become clear what these operators have in common. Score evaluation, selection, mutation and all the other operators that have been introduced and used in evolutionary algorithms may be quite different at first sight. But from an abstract perspective they have one important thing in common: they serve to process individuals in a particular manner. For instance the selection operator selects good individuals for mating from a group of individuals. In contrast, the mutation operator alters the contents of individuals, one by one sequentially.

Within the *eaLib* toolkit individuals are grouped together inside classes implementing the interface *IndividualStream.* An individual stream is a collection of individuals which provides several methods for putting individuals on the stream, removing them from the stream and iterating over all individuals contained in the stream. The underlying data structures are hidden from the user. Using this structure as a collection of individuals permits interfaces to be defined for so-called stream processors. The most simple case is a *SingleStreamProcessor*. A single-stream processor takes an individual stream as an argument, processes this stream in a particular way and passes the processed stream to the succeeding operator(s). This can be extended to processors that split up and merge streams. In most cases it is appropriate to apply a single-stream processor. The processors for splitting and merging streams can be used in algorithms which either involve parallel-running evolution cycles or implement the island population model.

For a better illustration of how these processors work, a closer look at the recombination mechanism follows. The rest of this section explains how the recombination mechanism functions with regard to the genetic representation introduced in section 2. As individuals can contain any number of chromosomes with any data type, common crossover procedures cannot be employed directly on an instance of the class *Individual*. Therefore the abstract base class *Recombination* is subclassed by the two classes *IndividualRecombination* and *ChromosomeRecombination*. The former is responsible for recombining individuals, while the latter recombines chromosomes. The subclasses of *ChromosomeRecombination* are well-known operators for recombining pairs of data of types integer, float, binary string, lists and so on. In contrast, the class *IndividualRecombination* is a collection of chromosome recombination operators, one for each chromosome inside the individuals to be recombined, having the ability to handle the appropriate data type. The functionality is shown in Fig. 2. The operator takes two (or more) individuals from the input stream and constructs one (or more) child individual(s) by combining the parent chromosomes with the corresponding chromosome recombination operators. The newly created child individuals are put on the output stream which is produced as the result of this operator. The stream is passed to the following operator (for instance a mutation operator) which handles the stream in a similiar way.

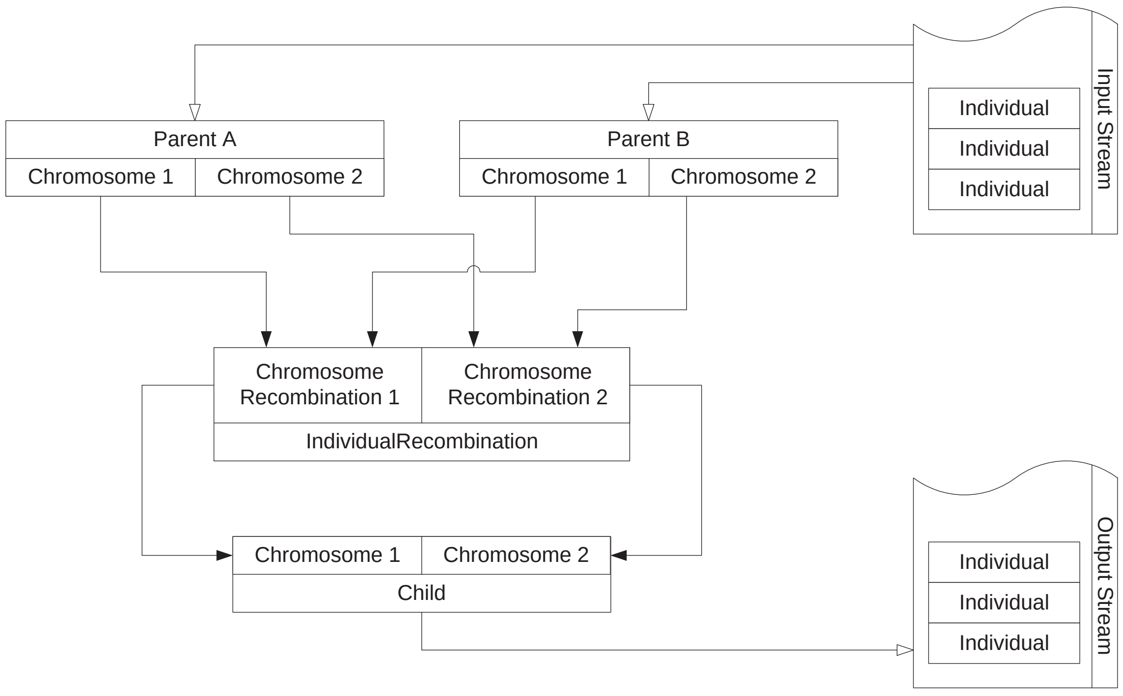

Figure 2: Recombination of individuals contained in a stream

All components contained in the toolkit work in this vein, even the score and fitness evaluation. For an overview of available components the documentation of the Application Programming Interface (API) should be consulted.[10]

4 ALGORITHM CREATION

After having determined how genetic operators can be transformed into software components, it is now possible to create algorithms by using these components. One of the common cases is an evolutionary algorithm with only one population. A more abstract look at the structure of such an algorithm reveals that the algorithm can be separated into three stages. The first includes the initialization of a population and an optional preprocessing. The second one is the transformation stage, where individuals evolve within an evolution cycle. In the third and last stage, the individuals can be optionally postprocessed and the result of the algorithm extracted. Genetic components can be assigned to each of the stages of this basic scheme.

The creation of an individual stream is always performed by an initialization operator. Such an operator can be seen as an individual source. At the other end of the flow, a stream can be captured by a sink (for instance for extracting results). The first operator provides the retrieval of individuals while the second one provides storage. A combination of both is a so-called gate, where individual streams are flowing through and evolving within. A gate is nothing but a population. These three types of operators are connected by stream processors as already discussed.

To generate a valid algorithm, therefore, all that is necessary is to create instances of suitable components and connect them together in a reasonable manner. It is up to the user of the toolkit to ensure that the connection structure makes sense. The advantage of that concept is a much higher flexibility, because there are no restrictions in creating the connection stucture.

To create an algorithm the user has to derive a class from the base class *Thread-Controller*. This class contains the abstract method *setup()*, which must be implemented by the user. In this method all necessary components are created and connected together. The controller is the central instance of the whole algorithm which monitors all components, starts and stops them.

The controller takes advantage of the built-in multithreading capabilities of Java. Every genetic component runs inside its own thread. All threads are started during the setup stage of the algorithm and are sleeping all the time. Every component knows about its successor component(s) and is able to notify those successors when its own task is completed and the processed individual stream can be passed to the following operator(s).

Another advantage of the use of multithreading is the implicit parallelism that the mechanism contains. With the wiring of components as described it is quite easy to create algorithms with parallel running evolution cycles or algorithms based on the island model. For that case special operators for splitting and merging individual streams are provided. This concept will, in due course, be extended to components which are able to pass streams to other components running on machines in a local network. Thus it will be possible to create real parallel-running algorithms using the same mechanisms as described above.

5 CONCLUSION

In this paper the evolutionary computation framework *eaLib* has been introduced. The motivation for the work on this toolkit was the need for flexible and fast creation of algorithms with little programming effort.

The toolkit's generic form of representing potential solutions of optimization tasks has been shown. The genetic representation introduced in *eaLib* is a general-purpose form. It can be used for the creation of simple individuals as well as complicated ones.

Furthermore a way of structuring evolutionary algorithms into software components has been shown. Based on a common interface for genetic operators, these software components can be connected together to form valid algorithms. The application of such components provides a high flexibility in creating algorithms originating from different evolution schemes.

The aim of this work was to provide a computation framework to support experimentation with genetic operators and fast creation of algorithms. An important point, that is not supported at the moment, is parallel computation support. Therefore future work will concentrate on the development of a generic model for distributing evolutionary algorithms over a local network.

REFERENCES

[1] Heinz Mühlenbein and Thilo Mahnig. Evolutionary computation and beyond. In *Foundations of Real World Intelligence.* CLSI Publications, Stanford, (2001).

[2] Java^{TM}2 SDK, Standard Edition, Version 1.3. `http://java.sun.com/j2se/1.3/docs/index.html`, (2000).

[3] Matthew Caryl. Mutants - a generic genetic algorithm toolkit for ada 95. `http://www.cultofcelebrity.com/matthew/projects/mutants/index.html`, (1997).

[4] K. Deb and D. E. Goldberg. A messy genetic algorithm in c. Technical Report 91008, University of Illinois at Urbana-Champaign, (1991).

[5] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison Wesley, Reading, Massachussetts, (1989).

[6] Hartmut Pohlheim. *Evolutionäre Algorithmen.* Springer-Verlag Berlin, Heidelberg, New York, (2000).

[7] J. E. Baker. Adaptive selection methods for genetic algorithms. In *Proceedings of the International Conference on Genetic Algoritms and their Application*, pages 101–111, (1985).

[8] T. Bäck and F. Hoffmeister. Extended selection mechanisms in genetic algorithms. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 92–99, (1991).

[9] Claudia Piemont. *Komponenten in Java.* dpunkt-Verlag, Heidelberg, (1999).

[10] Andreas Rummler. eaLib API Documentation. `http://www.inf-technik.tu-ilmenau.de/~rummler/eng/ealib.html`, (2001).