

JCLEC: a Java framework for evolutionary computation

Sebastián Ventura · Cristóbal Romero ·
Amelia Zafra · José A. Delgado · César Hervás

© Springer-Verlag 2007

Abstract In this paper we describe JCLEC, a Java software system for the development of evolutionary computation applications. This system has been designed as a framework, applying design patterns to maximize its reusability and adaptability to new paradigms with a minimum of programming effort. JCLEC architecture comprises three main modules: the core contains all abstract type definitions and their implementation; experiments runner is a scripting environment to run algorithms in batch mode; finally, GenLab is a graphical user interface that allows users to configure an algorithm, to execute it interactively and to visualize the results obtained. The use of JCLEC system is illustrated through the analysis of one case study: the resolution of the 0/1 knapsack problem by means of evolutionary algorithms.

Keywords Evolutionary computation software tools · Framework · Java · Object oriented design

1 Introduction

The use of evolutionary computation (EC) algorithms for problem solving is a widespread practice. Examples such as

industrial design (Benedetti et al. 2006), the identification of biochemical networks (Cho et al. 2006), the learning of boolean queries (Cordón et al. 2006), the learning of controllers in robotics (Mucientes et al. 2006) or the improvement of e-learning systems (Romero et al. 2004) show their suitability as problem solvers in a wide range of scientific fields.

Although evolutionary algorithms (EAs) are powerful for solving a wide range of scientific problems, their use requires certain programming expertise along with considerable time and effort in order to write a computer program for implementing the often sophisticated algorithm according to user needs. This work can be tedious and needs to be done before users can start the task they really should be working on. A simple solution is to get a ready-to-use EC software system, which is often developed for general purposes but has the potential to be applied to any specific application. By doing this, user removes the tedious job of having to codify the commonalities himself and he can concentrate on his specific needs, such as specialized functions for fitness evaluation, reproductional operators, or high-performance representations.

For the last few years, a large number of EC software tools have been developed. Some of them are specialized in a concrete EC flavor: genetic algorithms (GA) (Chuang 2000; Jin et al. 2005), memetic algorithms (Krasnogor and Smith 2000), genetic programming (GP) (Punch and Zongker 1998, 2005), distributed EAs (Tan et al. 2003), parameter control in EAs (Liu et al. 2004), evolutionary multiobjective optimization (Tan et al. 2001) and learning classifier systems (Meyer and Hufschlag 2006). Others are generic tools, that is, they can be used both to develop a variety of EAs and to be applied to different problems. This category of tools includes *ECJ* (Luke et al. 2006), one of the most popular tools at present. Its open architecture allows a great variety of EAs to be represented. However, standard distribution does not

S. Ventura (✉) · C. Romero · A. Zafra · J.A. Delgado · C. Hervás
Department of Computer Sciences and Numerical Analysis,
University of Córdoba, Campus Universitario de Rabanales,
edificio Albert Einstein., 14071 Cordoba, Spain
e-mail: sventura@uco.es

C. Romero
e-mail: cromero@uco.es

A. Zafra
e-mail: io2zagoa@uco.es

J. A. Delgado
e-mail: i92deosj@uco.es

C. Hervás
e-mail: chervas@uco.es

provide a variety of ready-to-use components (algorithms or genetic operators). *Evolvica* (Rummler 2006; Rummler and Scarbata 2001) is another interesting EC tool. This system has a graphical user interface (GUI), that lets users specify EAs by manipulating program elements graphically. This visual model allows EC models to be developed quickly, but its use is found to be complicated by non-experts in the EC field. Other interesting tools are *Open Beagle* (Gagné and Parizeau 2006b) and *EO* (Keijzer et al. 2001), both coded in C++. The first has an architecture that resembles ECJ, and can be used in the same applications, but it does not have a GUI to configure algorithms and visualize results. The system *EO* has components which makes algorithm configuration easier (Collet et al. 2000), although is difficult to extend.

As we can see, there are numerous EC tools, but most of them are mainly meant for people experienced in the EC field. Furthermore, although there are excellent generic tools, most of them do not have a variety of ready-to-use components which allows the EC researchers to carry out the comparison between their own algorithms and others reported in the bibliography. Finally, with the exception of PPCea system (Liu et al. 2004), there do not seem to be any systems that deal with experimental studies in EC.

This paper presents the JCLEC system which was developed to address some of the previously mentioned problems involved in the design of an EC tool. This system can be used by people who are inexperienced in the EC field, because it has a GUI which eases such tasks as the configuration, execution and verification of results, and it has a great variety of EAs and ready-to-use representations. Also, this system can be used by EC researchers, because it is easy to extend and allows test suites to be defined. The objective of this article is to present its design principles and system characteristics, as well as to show several examples of how this application can be used both by people unfamiliar with EC as well as by experts in the field.

We have organized this paper as follows. In the next section, we will analyze some considerations about the design of an EC software system. Then, we will present JCLEC, its architectonic principles and the subsystems that comprise it. After this presentation, we will illustrate some of the previously discussed ideas by means of one example: the 0/1 knapsack problem. We will finalize exposing conclusions and the improvements foreseen for the tool.

2 Design of an EC software system

The design of a generic EC software system is not an easy task. First, EC is a diverse paradigm and the system should take on all its variants [GA, GP, evolution strategies (ES), evolutionary programming (EP)]. The system should also make possible the addition of other new paradigms. Furthermore, if the system is used by EC expert researchers, it has to

allow the realization of experimental studies and the development of reports in a flexible and configurable way. On the other hand, if the system is used by less experienced researchers in the EC field, it is more appropriate to have a GUI, where the algorithm configuration can be done easily and a visual monitoring of the evolutionary process can be carried out. Finally, the system's components should be available in a library so that they can be used for developing self-reliant applications.

In spite of being an important issue, there are too few publications about the design of an EC software system. The papers of Cona (1995) and Keith and Martin (1994) analyze different ways of coding the representation of genetic programs. More recently, Lenaers and Manderick (1998) make an in depth analysis of the development of a GP framework. Also, the work of Krasnogor and Smith (2000) discusses the use of design patterns in the development of a memetic algorithm framework. Finally, the work of Gagné and Parizeau (2006a) explains the design principles which should be a generic EC framework.

In this section we analyze the application of object oriented programming (OOP) ideas in the development of EC software systems. First, we introduce the framework term and how this concept fits the idea of a robust, reusable and extensible software for EC. Then, we will analyze how the design patterns can help to make an EC framework in a flexible way.

2.1 Framework design

From an OOP perspective, an EA can be seen as an abstract class of algorithms, and its different flavors such as GA, GP, ES or EP can be seen as some of its concrete instantiations (Gagné and Parizeau 2006a). Building a robust and reusable design for this model is a difficult task, because there are multiple aspects to take into account: representation of individuals, mating selection procedure, crossover and mutation operators and survivor selection procedure. Certain types of operations can be applied to all individuals while others, like crossover and mutation are specifically characteristic of the representation used. Also, to guarantee software reusability, different EA elements must be uncoupled as much as possible. Finally, our system should be easy to extend, that is, the incorporation of new features must be able to be performed without requiring important system modifications. From the software engineering point of view, the best way of modeling it is in the form of a framework (Gagné and Parizeau 2006a; Krasnogor and Smith 2000).

The term framework can be defined as a set of co-operating classes that make up a reusable design for a specific software domain (Gamma et al. 1994). The framework dictates the architecture of the application, i.e., it defines the overall structure, its partitioning into classes and objects, its

key responsibilities and collaborations, and the thread of control. In other words, it filters out what parts are common in the domain and which are problem dependent. A framework can be considered as a puzzle which is almost finished where you still have to put in the remaining pieces to complete the puzzle although the resulting image can vary depending on which pieces you use.

As we can see, this kind of framework-based design resolves some of the issues mentioned with respect to the designing of an EC software system. First, we will define objects that represent individuals evolving in the system and their components (genotypes, phenotypes, and fitnesses). On the other hand, we define the EA control flow. This second part provides hooks for problem-specific or specialized functions and operators. The user must provide the system with all operators and functions necessary to perform the evolutionary process. The framework will take care of the functions' points of entry (where they are called and executed on the aggregate objects), and will describe the interface (how certain parts can be extended or reused) of all the variable parts. For example, the implementation of crossover or mutation is not hard-coded in a specific class in the algorithm. The user implements a reproduction operator creating a new class which satisfies a number of interface prerequisites and connects it to the framework. When the application is executed the framework will instantiate the operator and apply it.

2.2 Design patterns

The design of an EC framework can be greatly improved using design patterns (Gamma et al. 1994; Grand 1998). A design pattern is a description of communicating objects and classes that is customized to solve a general design problem in a specific context. Each pattern represents a common and recurring design solution which can be applied over and over again in different problem-specific contexts.

Patterns provide the designer with: (1) abstract templates on how to make specific parts of a framework more flexible towards changes (2) a mechanism to document the architecture of a framework using a high abstraction level vocabulary and (3) a mechanism to impose rules on how to reuse or extend the framework, i.e., outline a specific interface on how to incorporate extensions. On the other hand, they provide the user with (1) a higher level of documentation for a complex framework consisting of numerous heavily interconnected classes and objects and (2) a guidance on how to extend the framework with new variations and whether or not the extensions can be made.

There are several design patterns that can be used in the design of an EC framework. In the following, we explain patterns that have been used in the development of several existing frameworks (Gagné and Parizeau 2006b; Krasnogor and Smith 2000; Lenaers and Manderick 1998; Luke et al. 2006):

- *Singleton* is used to restrict instantiation of a class to one object. This is useful when exactly one object is needed to coordinate actions across the system.
- *Abstract factory* provides an interface for creating families of related objects without specifying their concrete implementations. In this way one can guarantee that the system is independent with respect to how specific objects are defined, created or manipulated.
- *Factory method* defines an interface for creating an object, but lets subclasses decide which class to instantiate. This pattern allows a class to defer instantiation to subclasses.
- *Builder* allows a user to separate the construction of an aggregated object from its representation. This allows the user to use the same construction process to build different representations.
- *Prototype* is used when the type of objects to be created is determined by a prototypical instance, which is cloned to produce new objects. This is useful when the inherent cost of creating a new object in the standard way (e.g., using the 'new' keyword) is prohibitively expensive for a given application.
- *Flyweight* allows a user to avoid the expense of multiple instances that contain the same information by sharing one instance.
- *Strategy* defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently of the clients that use it.
- *Template method* defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. This pattern lets subclasses redefine certain steps of an algorithm without changing the algorithm.
- *Visitor* represents an operation to be performed on the elements of an object's structure. Visitor lets you define new operations without changing the classes of the elements where it will operate.

3 JCLEC

JCLEC is an EC framework developed in the Java programming language. The project started as a class library in 1999 (Ventura et al. 2002). In the years 2003–2004 the software has been completely re-written in order to resolve some fundamental problems in the architecture and today it is in its third major version. It has been released with the GNU general public licence (GPL) and it is hosted as a free software project in the SourceForge page.¹

Three layers comprise the JCLEC architecture [see the UML package diagram (Fowler 2003) in Fig. 1]. The *system core* is in the lowest layer. It has the definition of the abstract types, its base implementations and some software modules that provide all the functionality to the system. The

¹ <http://jclec.sourceforge.net/>.

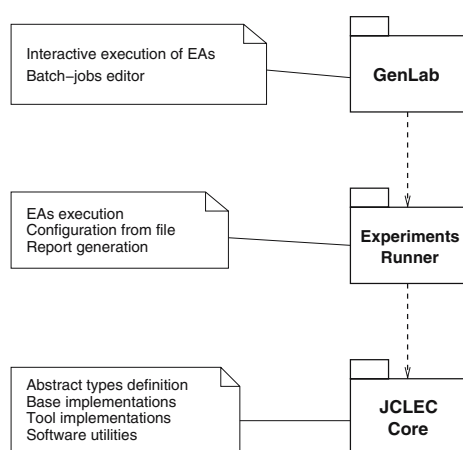


Fig. 1 JCLEC architecture

experiments runner system is built on the base of the core layer. It reads a specification file that contains the configuration of one or several algorithm executions and, after checking its correctness, create the necessary objects, executes the algorithms and saves the results in one or several report files. Finally, *GenLab* is a GUI for EC built on experiments runner and core subsystems. This interface allows the user to configure an algorithm, execute it and visualize the results on-line. The system can also be used to define experiments that will be executed by the experiments runner module. Next, we are going to describe the main features of each subsystem.

3.1 JCLEC core

The JCLEC core defines the data types that define the functionality of the framework. This section discusses class hierarchy and design patterns used as well as the package structure in order to get an overview of this tool's capabilities.

3.1.1 Class hierarchy

Figure 2 is a UML class diagram (Fowler 2003) that shows the interfaces that define the functionality of the JCLEC system. As can be seen, there are objects related to individuals (*Individual* and *IFitness*), their commonalities (*ISpecies* and *IEvaluator*), the evolutionary system (*ISystem*), actions performed in the course of evolution (*IProvider*, *ISelector*, *IRecombinator* and *IMutator*) and the EA itself (*IAlgorithm*).

The *Individual* interface represents one of the individuals that lives in a system in evolution. This interface does not declare methods related to the individual's genotype or to its phenotype, granted that this functionality is defined in the lower classes of the hierarchy. In fact, JCLEC's core contains several implementations of the interface *Individual* that are distinguished in the genotype that they present (bit string, int

or double arrays, expression trees or syntax trees). Such classes can be used directly in the implementation of EAs or they can be extended by the user (defining the phenotype that maps to a given genotype). Obviously, the user can also define new types of individuals associated with new representations. As a matter of fact, if a class implements the *Individual* interface, then integration with the other system components is taken for granted. All the *Individual* instances contain an object that implements the interface *IFitness*, which denotes the individual's fitness. There are several implementations for this interface, that represent fitness in single-objective and multi-objective problems.

The *ISystem* interface represents an evolutionary system (for example, a population) in an abstract way. This class will contain, among other things, the beings that inhabit the system and the current generation, as well as information on the individuals. The information on the individuals is not encoded directly in the class *ISystem* or in its subclasses, but rather it is *delegated* to the classes *ISpecies* and *IEvaluator*. This allows the use of the same subclass of *ISystem* to represent systems that only differ in the type of individuals that inhabit the system. The interfaces that extend to *ISpecies* define methods that provide information on the structure of the individuals (for example, the length of the chromosome and the schema in the case of linear genotypes or the maximum size of tree and the token set in GP). These methods will be used by the genetic operators to handle individuals correctly. These interfaces also define a method to create new instances of the *Individual* subclass they represent, given their genotype. This use of the *abstract factory* and *factory method* patterns allow genetic operators to create instances of a specific class without having to know what class it is. On the other hand, *IEvaluator* defines the method *evaluate(List)* that performs the evaluation of the individuals, that is, it computes and assigns their fitness. As we will see in Sect. 4, in order to solve a problem with JCLEC, it is necessary for the user to implement a class that extends to this interface, providing the system with a way to obtain the individuals' fitness.

Elementary operations performed in the course of the evolution are represented by the *ITool* interface. As we can see in Fig. 2, there are several interfaces that extend *ITool*: *IProvider* represents the action of creating new individuals, *ISelector* is a selection procedure, *IRecombinator* is a recombination method and *IMutator* represent a mutation operation. *ITool* interface defines the method *contextualize*, that associates an object *ITool* with an object *ISystem* (its execution context). This association relationship allows the object *ITool* to access information that is contained in the object *ISystem* and which is necessary to carry out its work correctly.

The *IAlgorithm* interface represents EAs in an abstract way. This class has a reference to an *ISystem* object (the system that experiences evolution) as well as some references to

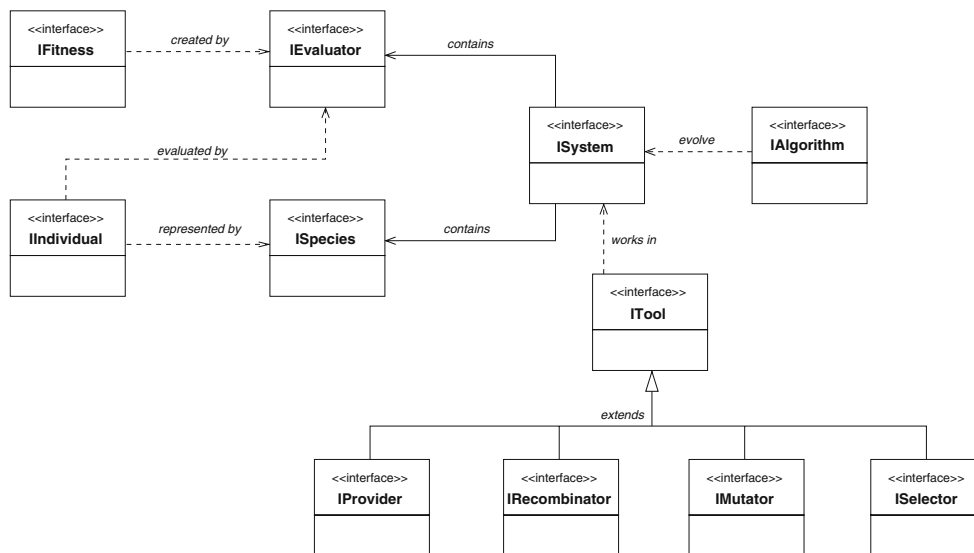


Fig. 2 Abstract types in JCLEC core hierarchy

ITool objects. These references are defined generically, and they should be set just before the algorithm is ready to be executed. With this type of design, the same *IAlgorithm* class can represent variants of an algorithm, that differs in the type of individuals that exist in the system or in the genetic operators employed, (but not in the course of evolution, which is implemented or codified directly within the class). As we can see, we again use the *delegation pattern* to make our implementation as generic as possible. The *IAlgorithm* interface applies also the *strategy pattern* to define the control flow of any EA in a generic way. This interface contains three methods related to the execution of an EA: *doInit*, that initializes the algorithm, *doIterate* that makes an iteration and *isFinished* that checks when an algorithm has finished. A user only has to know these methods to run an algorithm, without specific knowledge about their implementation. This design pattern has also been applied in the case of the populational algorithms, that extend to the *PopulationAlgorithm* abstract class. This class implements the *doIterate* method based on four abstract methods: *doSelection*, that performs parents selection, *doGeneration*, that produces new individuals from the parents, *doReplacement*, that decides which individuals must be replaced and *doUpdate*, that updates population.² The *PopulationAlgorithm* subclasses will implement these four methods to define the concrete algorithm flow.

3.1.2 System configuration

As we have already seen, before an algorithm is ready to run, it is necessary to carry out a set-up process in which the elements that have been defined in a generic way (for exam-

² These four steps in which each iteration of the algorithm is divided have been proposed by Deb (2005).

ple, the genetic operators that will apply) are setup. Other objects such as *ISpecies*, *IEvaluator* and some subclasses of *ITool* also need to be configured before their use. JCLEC implements two alternative configuration mechanisms: one is based on the interpretation of a configuration file, and the other is based on the concept of Java Beans (Rodrigues 1998).

The *file configuration mechanism* is based on the use of the interface *IConfigure*. This interface defines the method *configure*, that set up an object from the information contained in a *Configuration*³ object. The advantage of this approach is that it avoids interaction with the user during the set-up process, carrying this process out more quickly. Other systems, such as OpenBeagle (Gagné and Parizeau 2006b) or ECJ (Luke et al. 2006) have configuration mechanisms similar to this.

Besides the file configuration mechanism, all JCLEC objects that present configurable fields implement accessor methods (*getXXX* and *setXXX*) that give read-and-write access to these fields. This allows the establishment of an interactive setup process, in which the application requests the user to give the configuration values, and it does not allow an algorithm to run until the system has been configured correctly. This mechanism, used in the *GenLab* system, is similar to the one that other graphic applications present as *Evolvica* (Rummler 2006).

3.1.3 Algorithms listeners and events

In order to obtain information about the execution of an evolutionary process we have defined a listeners system similar to the one used in the management of events in Java. This

³ This object is defined in the Jakarta Commons Configuration class library (Apache Software Foundation 2006).

system consists of the *IAlgorithmListener* interface and the *AlgorithmEvent* class. The *IAlgorithmListener* objects take charge of picking up all the events related to an algorithm execution (algorithm started, iteration completed and algorithm finished) and react depending on the events. The *AlgorithmEvent* represents the events that happen during an algorithm execution. This class has a reference to the algorithm in order to access the current state and to react according to the object it has been created for.

3.1.4 Package structure

The structure of JCLEC core is organized in packages, that is, sets of classes and interfaces grouped by a specific criterion. In this section we are going to describe the main packages. We do not describe them exhaustively but instead deal with the functionality of each of them in order to get an overview of the system.

net.sf.jclec This package is the root of the JCLEC hierarchy, containing all abstract datatypes described in the previous section. It also defines the *IConfigure* interface, that allows to initialize the JCLEC objects from a configuration file.

net.sf.jclec.base This package contains base implementations (abstract classes) for all the interfaces defined in the *org.jclec* package and other generic classes largely used in the EA library.

net.sf.jclec.fitness This package contains several implementations of the *IFitness* interface. The package contains also the definition for several classes that establish ordering relationships between *IFitness* objects (they implement the *java.util.Comparator* interface). Such objects are used to sort individuals in different algorithm phases like, for instance, parents selection or system update.

net.sf.jclec.selector This package has implementations for several selection methods (implementations for the *ISelector* interface). At the present time, we have implementations for the following methods: roulette selection, Boltzmann selection, stochastic remaining selection, universal stochastic selection, range selection, tournament selection, NAM selection and UFS selection.

net.sf.jclec.binarray This package defines the classes needed to implement binary encoded GA (Goldberg 1989). For instance, the *BinArrayIndividual* class defines individuals with a bits array as genotype. On the other hand, the *BinArraySpecies* class defines the structure of *BinArrayIndividuals* (its length and the schema that represent them). The package also has implementations for operators that work selectively over individuals with binary lineal genotype: one point,

two points and uniform crossovers, one allele and uniform mutations.

net.sf.jclec.intarray This package defines the *IntArrayIndividual* class that represents an individual with a list of integer values as genotype, and the *IntArrayIndividualSpecies* class, that represents this kind of individuals. It also contains the implementation for several operators that work with this type of individuals: one point, two points and uniform crossovers and one allele and uniform mutators.

net.sf.jclec.rearray This package contains the necessary classes to implement a real coded genetic algorithm. It has the *RealArrayIndividual* class that represents an individual with a vector of real values as genotype. It also has the *RealArrayIndividualSpecies* class that defines the structure of a set of real encoded individuals (number of alleles and range of allowed values for each allele).

The package has some operators (creation of new individuals, crossover and mutation) that work specifically over this type of individuals. It has two-armed recombination operators [arithmetic, BGA linear, BLX- α , fuzzy, extended linear, extended fuzzy, SBX, UNDX and others (Herrera et al. 1998)] and mutation operators [random, not uniform, modal continuous, modal discrete and Muhlenbein mutation (Herrera et al. 1998)]. It also has some multi-parent crossover operators (panmictic discrete, intermediate generalized, recombination of a set of genes, recombination by mixing m-tuples, majority mix, half mix, uniform crossover, crossover based on occurrences and aptitude, diagonal crossover, mass center crossover, seed crossover, UNDX-n crossover) and crossover based on confidence intervals (CIXL1 and CIXL2) (Hervás and Ortiz 2005).

net.sf.jclec.exprtree This package defines a type of individual, called *ExprTreeIndividual*, that can be used in conventional (Koza 1992) and strongly typed (Montana 1995) GP algorithms. The package also defines the *ExprTreeIndividualSpecies* class that defines the structure of a set of individuals of this type and operators to manipulate them in a consistent way: the branch crossover and branch mutation operators. Also, the package contains other mutation operators (one node, all nodes, expand branch, truncate branch, promote node, demote node and gaussian) used in the implementation of EP algorithms (Chellapilla 1997).

net.sf.jclec.syntaxtree This package has an implementation for Grammar Based GP (Ratle and Sebag 2001; Wong 2005). In this paradigm, individuals have a syntactic tree (that belongs to a user-defined grammar) as genotype. This grammar contributes to have a better control over the structure of individuals and over genetic operators. It lets incorporate knowledge about the problem domain and to bias the search

toward the most appropriate regions of the search space. The package has implementations for typical operators (selective crossover, selective mutator and directed mutation) and others proposed that have shown its utility in the resolution of some problems of symbolic regression.

net.sf.jclec.gep This package has an implementation for Gene Expression Programming (Ferreira 2002). In this paradigm, individuals present an integer lineal genotype that maps to an expression tree. This tree will be used in the evaluation of individuals. The package has the typical operators for the following paradigms: mutation, one point and two points recombination, gene recombination, gene transposition, IS transposition and RIS transposition.

net.sf.jclec.ge This package contains an implementation for Grammatical Evolution (O'Neill and Ryan 2003). In this paradigm, individuals contain a binary array as genotype that maps to a sequence of productions of a context-free grammar. The phenotype is obtained starting from the terminal symbol of the grammar and applying the change defined by the individual genotype. The package also has the typical genetic operators for this paradigm.

net.sf.jclec.algorithms and related packages. This package has an abstract implementation for the *IAlgorithm* interface and final implementations for several types of EAs. In the current version of JCLEC, the implemented algorithms are:

- *Classic algorithms*: simple generational, steady state and CHC (Eshelman 1990).
- *Multi-objective algorithms*: NSGA-II and SPEA2 (Coello et al. 2002; Deb 2002).
- *Memetic algorithms*: generational and steady state (Krasnogor and Smith 2005).
- Scatter search algorithm (Laguna et al. 2002).
- *Niching algorithms*: clearing, sequential and fitness sharing (Sareni and Krähenbuhl 1998).

3.2 JCLEC experiments runner

The JCLEC experiments runner (JER) can be seen as a simple EA scripting environment. This application reads an EA script file in XML format and executes all the indicated algorithms, generating one or several report files as output.

The internal operation in JER can be seen as a use case of the application programming interface (API) provided by the *IAlgorithm* and *IConfiguration* interfaces. First, the application extracts one or several *process* elements of the input file. For each *process* element, it extracts a subelement *algorithm* and a subelement *listeners*. The first one is used to create and configure an instance of an *IAlgorithm* subclass. The second one consists of one or several *listener* elements,

used to create and configure instances of *IAlgorithmListener*. Object creation and configuration is accomplished by means of the Java reflection mechanism (Forman and Forman 2004) and the *configure* method, respectively. Once the *IAlgorithm* object is created and the *IAlgorithmListeners* are attached to their respective algorithm, the system performs the algorithm execution phase. To do that, *experiments runner* uses the API provided by the *IAlgorithm*, that is, the *doInit*, *doIterate* and *isFinished* methods.

The main advantage of JER is that we can define several runs in a single experiment file. This allows experimental studies to be carried out easily. Also, as user interaction is not required, the experiments can be planned outside the work-hours, taking advantage of the moments when the servers have less activity. The disadvantage is that the structure of the configuration files is not very user friendly. This problem can be partially solved by using the graphical editor *GenLab*.

3.3 GenLab: a graphical user interface for EC

GenLab is a graphical user application included in the JCLEC distribution. Its main objectives are (1) to interactively execute EAs and (2) to edit experiment files used by the JER.

Figure 3 (at the left) shows the main window of the *GenLab* application when we have selected the interactive mode, that is, executing only one algorithm and visualizing the results in execution time. As we can see, we have the typical operations in the main menu (for example, to create a new application or to save the current execution in a file) as well as an input data area with three different zones:

- *Algorithm selection*. In this zone, the user chooses one algorithm from among all the available algorithms in the system.
- *Algorithm configuration*. This zone is different for each available algorithm. In this zone the user sets the algorithm configuration parameters.
- *Visualization results*. In this area the user can select some items to be visualized. Each of these items is associated with a listener that will gather and visualize the information during the algorithm execution.

When the algorithm is configured the user can choose either to save the configuration in a XML file or to execute it. If the user chooses to execute it, then the system will show several auxiliary windows with the results obtained during the execution. Figure 3 (at the right) shows two charts: one with the fitness of the best individual for each generation and the other with the average fitness in the whole evolutionary process.

The look of the experiments edition window is basically the same as the previous interactive mode, but now there is a different window for each algorithm. In this mode, the

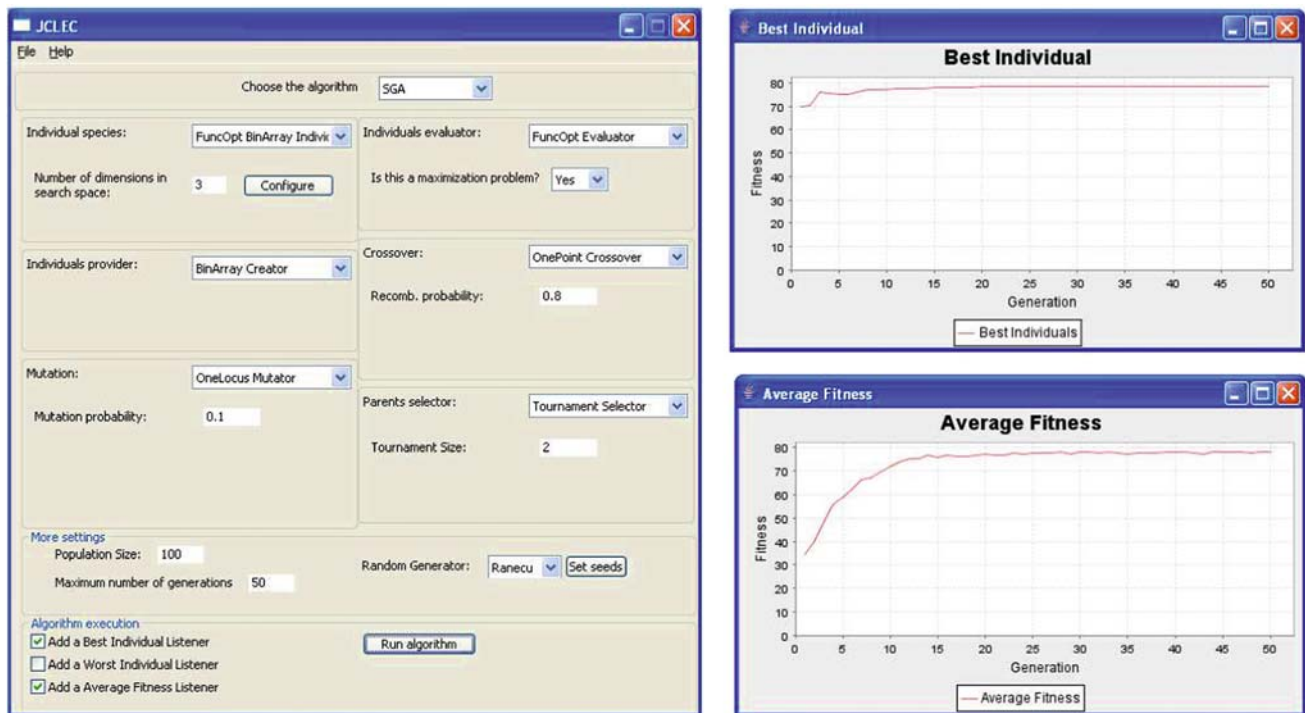


Fig. 3 Algorithms run window (SGA view)

system provides tools to copy and paste the algorithm configurations in order to ease the configurations of experiments with several executions.

4 Case study: the 0/1 Knapsack problem

The 0/1 knapsack problem (Martello and Toth 1990) is a classic problem in combinatorial optimization. It derives its name from the maximization problem of choosing possible essentials that can fit into one bag (of maximum weight) to be carried on a trip. A similar problem very often appears in business, combinatorics, complexity theory, cryptography and applied mathematics. Given a set of items, each with a cost and a value, the number of each item is then determined to be included in a collection so that the total cost is less than some given cost and the total value is as great as possible.

This problem is NP-hard, and it has been solved with dynamic programming techniques, although it can also be solved with EC algorithms. In this section, we are going to show how to use JCLEC to solve this problem from two different view points of view: a classic viewpoint in which the fitness function is the total value of the knapsack (and we have to deal with the restriction of the maximum weight) and a multi-objective viewpoint (Steuer 1989; Zitzler and Thiele 1998) with two confronted objectives (the value and the weight of the knapsack).

4.1 First solving approach

The most common way to solve the above problem is to use a binary encoding scheme. In this scheme, the individual contains a binary genotype with many genes as different items that we can lump together in the knapsack. The meaning of value 1 for the i th gene is that the i th item has been put in the knapsack, and the meaning of value 0 is the opposite. The fitness of the individual can be calculated very easily. We only have to add the values of the items whose associated bit is 1. And if we want to use the weight restriction, we have to apply the same fitness function only if the total knapsack weight (calculated as the addition of the individual weights) does not surpass the maximum weight established.

As we have seen previously, the JCLEC system has a package to represent binary individuals with crossover and mutator genetic operators. So, we only have to write the source code to evaluate the fitness of the individual (*IEvaluator* object). Figure 4 shows the corresponding code of this class denominated *KnapsackEvaluator* to solve the problem with 30 items.

As we can see, the class implements two methods. The first one is the *evaluate()* method that sets the fitness value of individuals which is received as an argument. This method uses 30 bits as the length of the individual and it uses a *SimpleValueFitness* object that is assigned to the individual in execution. The second method is *getComparator()* and it returns


```

public KnapsackEvaluator
    extends AbstractEvaluator<BinArrayIndividual>
{
    /** Article values */

    private static final int [] PROFIT =
    {
        10, 10, 10, 10, 10, 10, 15, 15, 15, 15,
        15, 15, 20, 20, 20, 20, 20, 20, 20, 40, 40,
        40, 40, 40, 40, 50, 50, 50, 50, 50, 50
    };

    /** Article weights */

    private static final int [] WEIGHT =
    {
        3, 3, 3, 3, 3, 3, 5, 5, 5, 5,
        5, 5, 9, 9, 9, 9, 9, 9, 9, 40, 40,
        40, 40, 40, 40, 80, 80, 80, 80, 80, 80
    };

    /** Genotype length */

    private static final int GENOTYPE_LENGTH = 30;

    /** Weight restriction */

    private static final int MAX_WEIGHT = 300;

    /** Evaluation method */

    public void evaluate(BinArrayIndividual ind)
    {
        // Individual genotype
        byte [] genotype = ind.getGenotype();
        // Total weight and profit
        int totalweight = 0;
        int totalprofit = 0;
        // Calculate weight
        for (int i=0; i<GENOTYPE_LENGTH; i++) {
            totalweight += WEIGHT[i];
        }
        // Calculate profit (if necessary)
        if (totalweight <= MAX_WEIGHT) {
            for (int i=0; i<GENOTYPE_LENGTH; i++) {
                totalprofit += PROFIT[i];
            }
        }
        ind.setFitness(new SimpleValueFitness(totalprofit));
    }

    /** Return a valid comparator for generated fitnesses */

    public Comparator<IFitness> getComparator()
    {
        return new ValueFitnessComparator();
    }
}
    
```

Fig. 4 Simple evaluator used in the 0/1 knapsack problem

an object which establishes an ordering relation between the *IFitness* objects produced during the evaluation process (*ValueFitnessComparator* object). This object is used to sort individuals in several algorithm parts. The *SimpleValueFitness* and *ValueFitnessComparator* classes are defined in the system, so the user only has to know their meaning and to use them whenever he needs them.

Once the evaluator of the problem is defined (*KnapsackEvaluator* object in our case) we can resolve the problem without it being necessary to write more source code. Indeed, we can execute a simple generational algorithm (SGA), a steady state algorithm (SSA) or a CHC algorithm. For each of them, we can also choose among different selection methods, crossover operators (one-point, two point and uniform) and mutation operators (one locus or uniform). We can use the *GenLab* application if we want to do an interactive execution

or we can write a JER configuration file to execute a batch of executions. Figure 5 shows the configuration file of a SGA algorithm with a population of 50 individuals, a selection scheme by means of tournament of size 2, and it uses as genetic operators the one point crossover operator (with a probability of 0.8) and the one locus mutation operator (with a probability of 0.1). The algorithm will be iterated during 100 generations. With respect to the listener, we have used a basic report generator that produces a file with the best, worst and median individuals, the average fitness and its variance, every five generations. In Fig. 6 we can see a fragment of the generated report to the 15th generation of the evolutionary process.

4.2 Second solving approach

We can also solve the knapsack problem using a multi-objective perspective (Steuer 1989; Zitzler and Thiele 1998). In this case, there are two confronted objectives: the knapsack value to maximize and the knapsack weight to minimize. Obviously, these two objectives are conflicting and cannot be optimized at the same time: maximizing the overall profit means putting as many items as possible in the knapsack and minimum weight is achieved when no item is in the knapsack. There is a trade-off between profit and weight. Thus, in contrast to the single-objective 0/1 knapsack problem, there is not a single optimal solution but rather a set of optimal trade-offs which consists of all solutions that cannot be improved in one criterion without degrading another. The corresponding set is denoted as *Pareto-optimal set*.

Mathematically, the concept of Pareto optimality can be defined in terms of a dominance relation (with regard to the 0/1 knapsack problem):

- Given a set of solutions and two members A,B of the set. A is said to dominate B if and only if the profit of A is equal or greater than the profit of B and the weight of A is equal or less than the weight of B; and A is better in one objective, i.e., either the profit is greater or the weight is less.
- A solution A is denoted as nondominated regarding a given set if and only if no member of the set dominates A.
- Those solutions that are nondominated regarding the entire search space are called Pareto optimal.

Therefore, the optimization goal of the multiobjective 0/1 knapsack problem is to find the set of Pareto-optimal solutions. In this case, we can also use the binary representation described above, but the evaluation process is different because it has to calculate a multi-objective fitness. This process is performed by the *KnapsackMultiObjectiveEvaluator* class, whose code is shown in Fig. 7. As can be seen, this

Fig. 5 Configuration for a SGA run

```

<experiment>
  <process>
    <algorithm type="SGA">
      <rand-gen-factory="org.ayrna.jclec.util.random.RanmtFactory" seed="123456789"/>
      <population-size>50</population-size>
      <max-of-generations>100</max-of-generations>
      <species type="org.ayrna.jclec.binarray.BinArrayIndividualSpecies" genotype-length="50"/>
      <evaluator type="mypackage.KnapsackEvaluator"/>
      <provider type="org.ayrna.jclec.binarray.BinArrayCreator"/>
      <parents-selector type="org.ayrna.jclec.selector.TournamentSelector">
        <tournament-size>2</tournament-size>
      </parents-selector>
      <recombinator type="org.ayrna.jclec.binarray.OnePointCrossover" rec-prob="0.8"/>
      <mutator type="org.ayrna.jclec.binarray.OneLocusMutator" mut-prob="0.10"/>
    </algorithm>
    <listeners>
      <listener type="BasePopulationReport">
        <report-dir-name>report1</report-dir-name>
        <report-frequency>5</report-frequency>
        <include-individuals>false</include-individuals>
      </listener>
    </listeners>
  </process>
</experiment>

```

Fig. 6 Fragment of a report file obtained in a SGA run

```

<base-population-report-entry xmlns:j="http://javolution.org" generation="15">
  <absolutely-best j:class="org.ayrna.jclec.binarray.BinArrayIndividual">
    <genotype value="1 1 0 1 1 1 0 1 0 0 0 0 0 0 1 0 0 0 1 0 1 1 1 1 0 1 1 1 1 0 1 1 1 0"/>
    <fitness j:class="org.ayrna.jclec.fitness.SimpleValueFitness" value="1110.0"/>
  </absolutely-best>
  <best-individual j:class="org.ayrna.jclec.binarray.BinArrayIndividual">
    <genotype value="1 1 0 1 1 1 0 1 0 0 0 0 0 0 1 0 0 0 1 0 1 1 1 1 0 1 1 1 1 0 1 1 1 0"/>
    <fitness j:class="org.ayrna.jclec.fitness.SimpleValueFitness" value="1110.0"/>
  </best-individual>
  <worst-individual j:class="org.ayrna.jclec.binarray.BinArrayIndividual">
    <genotype value="1 0 1 1 1 1 0 1 1 0 1 1 0 1 1 0 1 0 0 0 1 1 1 1 0 1 1 0 1 0 0 1 1 1"/>
    <fitness j:class="org.ayrna.jclec.fitness.SimpleValueFitness" value="0.0"/>
  </worst-individual>
  <median-individual j:class="org.ayrna.jclec.binarray.BinArrayIndividual">
    <genotype value="0 0 0 1 1 0 1 0 0 0 0 0 0 0 1 0 0 0 1 0 1 1 1 1 0 1 1 0 1 1 0 0 1"/>
    <fitness j:class="org.ayrna.jclec.fitness.SimpleValueFitness" value="955.0"/>
  </median-individual>
  <average-fitness value="974.9"/>
  <fitness-variance value="23658.49"/>
</base-population-report-entry>

```

class also implements the *evaluate* and the *getComparator* methods but, in this case, the methods return objects from the *CompositeFitness* and *ParetoComparator* classes respectively. The first of these classes represents a fitness that is formed by several *ISimpleFitness* objects. In our case, the class stores two *SimpleValueFitness* objects (the two objectives to optimize) that have been generated when applying the *evaluate0* and *evaluate1* methods on the individual's genotype (see Fig. 7). The *ParetoComparator* class implements the dominance relation previously discussed.

Finally, in order to solve the problem, we can use one of the multiobjective algorithms provided by the system (in our case, SPEA2, NSGA-II or MOGLS algorithms) or implement our own multiobjective algorithms. As the experiment file for any of these experiments is very similar to the one shown in Fig. 5, it is not repeated here.

5 Conclusions and future work

In this work we have described JCLEC, a Java framework for Evolutionary Computing. We have shown its main features: a modular architecture, that is very easy to extend and that implements a lot of EC paradigms. We have analyzed

the development of applications using JCLEC components and the *GenLab* tool that executes algorithms defined by an XML configuration file. We have also shown an example about how to use JCLEC as a tool to resolve a problem using two different approaches.

The JCLEC system is continuously updated and improved. At the moment, we are working on the development of a real optimization toolkit with the following algorithms: ES (Beyer 2001), differential evolution (Storn and Price 1997), minimal generation gap (Higuchi et al. 2000) and generalized generation gap (Deb 2005; Deb et al. 2002) algorithms. We are also developing new GP algorithms such as the Token Competition algorithm (Wong 2005) used to discover classification rules with GP and improving the JER and the *GenLab* tools in order to be able to execute concurrently several EAs using the Java Threads API (Lewis and Berg 2000). In the case of JER, we will use parallel architecture to speed up the execution of batch-jobs and, in the case of *GenLab*, we will do a simultaneous pursuit of several algorithms. Finally, we are working on the development of a native version of JCLEC, using the compiler GCJ of GNU (Free Software Foundation 2006). This version aims to resolve problems that are heavily demanding from a computational point of view, and it will allow the performance of the JCLEC applications to be

```

public KnapsackMultiObjectiveEvaluator
  extends AbstractEvaluator<BinArrayIndividual>
{
  ...

  /** Pareto comparator for this problem */

  private static final Comparator<IFitness> COMPARATOR =
    new ParetoComparator (
      new ValueFitnessComparator(true),
      new ValueFitnessComparator(false)
    );

  /** Evaluation method */

  public void evaluate(BinArrayIndividual ind)
  {
    // Individual genotype
    byte [] genotype = ind.getGenotype();
    // Resulting fitness
    CompositeFitness fitness = new CompositeFitness(2);
    fitness.setComponent(0, evaluate0(genotype));
    fitness.setComponent(1, evaluate1(genotype));
    ind.setFitness(fitness);
  }

  /** Evaluate first objective (profit) */

  private final SimpleValueFitness evaluate0(byte [] genotype)
  {
    // Total profit
    int totalprofit = 0;
    // Calculate profit
    for (int i=0; i<GENOTYPE_LENGTH; i++) {
      totalprofit += PROFIT[i];
    }
    // Return partial fitness
    return new SimpleValueFitness(totalprofit);
  }

  /** Evaluate second objective (weight) */

  private final SimpleValueFitness evaluate1(byte [] genotype)
  {
    // Total weight
    int totalweight = 0;
    // Calculate weight
    for (int i=0; i<GENOTYPE_LENGTH; i++) {
      totalweight += WEIGHT[i];
    }
    // Return partial fitness
    return new SimpleValueFitness(totalweight);
  }

  /** Return a valid comparator for generated fitnesses */

  public Comparator<IFitness> getComparator()
  {
    return COMPARATOR;
  }
}

```

Fig. 7 Multiobjective evaluator used in the 0|1 knapsack problem

comparable to that of others developed in C++ systems (for example, Open BEAGLE or EO). The preliminary results are very promising (the increases in speed have reached up to ten times those of the pure Java version), although there are still many other questions to be resolved. These improvements will be incorporated into the future versions of the system.

Acknowledgments The authors gratefully acknowledge the financial support provided by the Spanish Department of Research of the Ministry of Science and Technology under TIN2005-08386-C05-02 Project and FEDER funds.

References

Apache Software Foundation (2006) Jakarta commons configuration project. <http://jakarta.apache.org/commons/configuration/>

Benedetti A, Farina M, Gobbi M (2006) Evolutionary multiobjective industrial design: the case of a racing car tire-suspension system. *IEEE Trans Evol Comput* 10(3):230–244

Beyer HG (2001) The theory of evolution strategies. In: *Natural computing*. Springer, Berlin

Chellapilla K (1997) Evolving computer programs without subtree crossover. *IEEE Trans Evol Comput* 1(3):209–216

Cho DY, Cho KH, Zhang BT (2006) Identification of biochemical networks by s-tree based genetic programming. *Bioinformatics* 22(14):1631–1640

Chuang AS (2000) An extensible genetic algorithm framework for problem solving in a common environment. *IEEE Trans Power Syst* 15(1):269–275

Coello C, van Veldhuizen DA, Lamont GB (2002) *Evolutionary algorithms for solving multi-objective problems*. In: *Genetic algorithms and evolutionary computation series*. Kluwer Academic Publishers, Norwell

Collet P, Lutton E, Schoenauer M, Louchet J (2000) Take it EASEA. In: *Parallel problem solving from nature—PPSN VI 6th international conference*. Lecture notes in computer science, Paris, France, vol 1917. Springer, Berlin, pp 16–20

Cona J (1995) Developing a genetic programming system. *AI Expert*, pp 20–29

Cordón O, Herrera-Viedma E, Luque M (2006) Improving the learning of boolean queries by means of a multiobjective IQBE evolutionary algorithm. *Inf Process Manage* 42(3):615–632

Deb K (2002) *Multi-objective optimization using evolutionary algorithms*. In: *Wiley interscience series on systems and optimization*, 3rd edn. Wiley, New York

Deb K (2005) A population-based algorithm-generator for real-parameter optimization. *Soft Comput* 9(4):236–253

Deb K, Pratap A, Agarwal S, Meyarivan T (2002) A fast and elitist multi-objective genetic algorithm: NSGA-II. *IEEE Trans Evol Comput* 6(2):181–197

Eshelman LJ (1990) The CHC adaptive search algorithm: how to have safe search when engaging in nontraditional genetic recombination. In: Rawlins GJE (ed) *Foundations of genetic algorithms. Proceedings of the 1st workshop on foundations of genetic algorithms*. Bloomington Campus, Indiana, USA, July 1990. Morgan Kaufmann, San Francisco, pp 265–283

Ferreira C (2002) *Gene expression programming: mathematical modelling by an artificial intelligence*, 1st edn. Gepsoft, Portugal

Forman IR, Forman N (2004) *Java reflection in action*. Manning Publications, Greenwich

Fowler M (2003) *UML Distilled: a brief guide to the standard object modeling language*. In: *Object technology*, 3rd edn. Addison-Wesley Professional, Reading

Free Software Foundation. Gcj: The GNU compiler for the Java™ programming language. <http://gcc.gnu.org/java/>, September 2006

Gagné C, Parizeau M (2006a) Genericity in evolutionary computation software tools: principles and case-study. *Int J Artif Intell Tools* 15(2):173–194

Gagné C, Parizeau M (2006b) Open BEAGLE: an evolutionary computation framework in C++. <http://beagle.gel.ulval.ca>

Gamma E, Helm R, Johnson R, Vlissides J (1994) *Design patterns: elements of reusable object oriented software*. Addison Wesley, Reading

Goldberg DE (1989) *Genetic algorithms in search, optimization, and machine learning*. Addison-Wesley Professional, Canada

Grand M (1998) *Patterns in Java. A catalog of reusable design patterns illustrated with UML*, vol 1, 1st edn. Wiley Computer Publishing, New York

Herrera F, Lozano M, Verdegay JL (1998) Tackling real-coded genetic algorithms: operators and tools for behavioural analysis. *Artif Intell Rev* 12(4):265–319

- Hervás C, Ortiz D (2005) Analyzing the statistical features of CIX-L2 crossover offspring. *Soft Comput* 4:270–279
- Higuchi T, Tsutsui S, Yamamura M (2000) Theoretical analysis of simple crossover for real-coded genetic algorithms. In: Schoenauer M, Deb K, Rudolph G, Yao X, Lutton E, Merelo JJ, Schwefel HP (eds) *Parallel problem solving in nature (PPSN-VI)*. Lecture notes in computer science, Paris, France, vol 1917. Springer, Berlin, pp 365–374
- Jin W, Tontiwachwunthikul P, Chan CW, Huang GH (2005) A genetic algorithms framework for grey non-linear programming problems. In: *Canadian conference on electrical and computer engineering 2005*, Saskatoon. IEEE, USA, pp 2187–2190
- Keijzer M, Merelo JJ, Romero G, Schoenauer M (2001) Evolving objects: a general purpose evolutionary computation library. In: Collet P, Fonlupt C, Hao J-K, Lutton E, Schoenauer M (eds) *Artificial evolution: selected papers from the 5th European conference on artificial evolution*. Lecture notes in computer science, London, UK, vol 2310. Springer, Berlin, pp 231–244
- Keith MJ, Martin MC (1994) Genetic programming in C++: implementation issues. In: Kinnear KE Jr (ed) *Advances in genetic programming*, Cambridge, MA, USA. MIT, Cambridge, pp 285–310
- Koza JR (1992) Genetic programming: on the programming of computers by means of natural selection. In: *Complex adaptive systems*. MIT, Cambridge
- Krasnogor N, Smith J (2000) MAFRA: a Java memetic algorithms framework. In: *Genetic and evolutionary computation 2000 workshops*, Las Vegas, Nevada, USA, pp 125–131
- Krasnogor N, Smith J (2005) A tutorial for competent memetic algorithms: model, taxonomy, and design issues. *IEEE Trans Evol Comput* 9(5):474–488
- Laguna M, Price Hossell K, Martí R (2002) *Scatter search: methodology and implementation in C*. Kluwer Academic Publishers, Norwell
- Lenaers T, Mandrick B (1998) Building a genetic programming framework: the added value of design patterns. In: Banzhaf W, Poli R, Schoenauer M, Fogarty TC (eds) *Proceedings of the EuroGP 98*. Lecture notes in computer science, Paris, France, vol 1391. Springer, Berlin, pp 196–208
- Lewis B, Berg DJ (2001) *Multithreaded programming with Java*. In: Java series. Prentice Hall, California
- Liu HS, Mernik M, Bryant BR (2004) Parameter control in evolutionary algorithms by domain-specific scripting language PPCEA. In: *Proceedings of the 1st international conference on bioinspired optimization methods and their applications (BIOMA'04)*, Ljubljana, Slovenia, pp 41–50
- Luke S, Panait L, Balan G, Paus S, Skolicki Z, Bassett J, Hubley R, Chircop A (2006) ECJ: a Java based evolutionary computation research system. <http://cs.gmu.edu/eclab/projects/ecj>
- Martello S, Toth P (1990) *Knapsack problems: algorithms and computer implementations*. Wiley, Chichester
- Meyer M, Hufschlag K (2006) A generic approach to an object-oriented learning classifier system library. *J Artif Soc Soc Simul* 9(3), <http://jasss.soc.surrey.ac.uk/9/3/9.html>
- Montana DJ (1995) Strongly typed genetic programming. *Evol Comput* 3(2):199–230
- Mucientes M, Moreno DL, Bugarín A, Barro S (2006) Evolutionary learning of a fuzzy controller for wall-following behavior in mobile robotics. *Soft Comput* 10(10):881–889
- O'Neill M, Ryan C (2003) *Grammatical evolution: evolutionary automatic programming in an arbitrary language*. In: *Genetic programming*, vol 3. Kluwer Academic Publishers, Boston
- Punch B, Zongker D (1998) lil-gp 1.1 beta. <http://garage.cse.msu.edu/software/lil-gp>
- Ratle A, Sebag M (2001) Grammar-guided genetic programming and dimensional consistency: application to non-parametric identification in mechanics. *Appl Soft Comput* 1(1):105–118
- Rodrigues LH (1998) *The awesome power of Java Beans*. Manning, Greenwich
- Romero C, Ventura S, de Bra P (2004) Knowledge discovery with genetic programming for providing feedback to courseware author. User modeling and user-adapted interaction. *J Personal Res* 14(5):425–465
- Rummler A (2006) Evolvica: a Java framework for evolutionary algorithms. <http://www.evolvica.org>
- Rummler A, Scarbata G (2001) eaLib—a Java framework for implementation of evolutionary algorithms. In: Reusch B (ed) *Fuzzy days 2001*. Lecture notes in computer science, vol 2206. Springer, Berlin Heidelberg, pp 92–102
- Sareni B, Krähenbuhl L (1998) Fitness sharing and niching methods revisited. *IEEE Trans Evol Comput* 2(3):97–106
- Silva S (2005) GPLAB: a genetic programming toolbox for MATLAB. <http://gplab.sourceforge.net>
- Steuer E (1989) *Multiple criteria optimization: theory, computation, and application*. Wiley, New York
- Storn R, Price K (1997) Differential evolution, a fast and efficient heuristic for global optimization over continuous spaces. *J Glob Optim* 11:341–359
- Tan KC, Lee TH, Khoo D, Khor EF (2001) A multiobjective evolutionary algorithm toolbox for computer-aided multiobjective optimization. *IEEE Trans Syst Man Cyber Part B Cyber* 31(4):537–556
- Tan KC, Tay A, Cai J (2003) Design and implementation of a distributed evolutionary computing software. *IEEE Trans Syst Man Cybern Part B Cybern* 33(3):325–338
- Ventura S, Ortiz D, Hervás C (2002) JCLEC: Una biblioteca de clases java para computación evolutiva. In: Alba E, Fernández F, Herrera F, Hidalgo JI, Lanchares J, Merelo JJ, Sánchez JM (eds) *Primer Congreso Español de Algoritmos Evolutivos y Bioinspirador*, Mérida, Spain, pp 23–30
- Wong ML (2005) Evolving recursive programs by using adaptive grammar based genetic programming. *Genet Program Evol Mach* 6(4):421–455
- Zitzler E, Thiele L (1998) Multiobjective optimization using evolutionary algorithms—a comparative case study. In: Eiben AE, Back T, Schoenauer M, Schwefel H (eds) *Parallel problem solving from nature—PPSN-V*. Lecture notes in computer science, Amsterdam, vol 1498. Springer, Berlin, pp 292–301