JEAF: A Java Evolutionary Algorithm Framework

P. Caamaño, R. Tedín, A. Paz-Lopez, and J. A. Becerra

Abstract—There are not many tools in the evolutionary computing field that allow researchers to implement, modify or compare different algorithms. Additionally, those tools usually lack flexibility, maintenance or some other characteristic, so researchers program their own solutions most of the time, reimplementing algorithms that have already been implemented hundreds of times. This paper introduces a new framework for evolutionary computation called JEAF (Java Evolutionary Algorithm Framework) that tries to offer a platform to facilitate the tasks of comparing, analyzing, modifying and implementing evolutionary algorithms, reusing components and programming as few as possible. JEAF also aims to be a tool for evolutionary algorithm users that employ these algorithms to solve other problems not related with evolutionary computation. In this sense, JEAF provides methods to distribute an evolutionary process and to plug external tools to perform the evaluation of candidate solutions.

I. INTRODUCTION

Whenever a researcher wants to implement a new evolutionary algorithm or to try to solve a particular problem with different evolutionary algorithms, they usually suffer the same problem: the lack of an updated, flexible and complete framework that permits such task with a minimum effort.

There exist some tools implementing evolutionary algorithms, but they present some typical problems:

- Few evolutionary algorithms implemented, many times there is just one.
- Outdated and unmaintained, so there are not recent algorithms or there are problems with recent compilers.
- Implementation of the evolutionary algorithms included is not validated against a reference benchmark.
- Inflexibility. Researchers need a profound knowledge of the tool to implement their algorithms, or they need to input almost the same lines of code as if they were coding from scratch, or their algorithms have to be fitted into an scheme that, although it may be adequate for the algorithms included in the tool, does not suit their own algorithms so well.
- Lack of documentation.

Therefore, the most usual situation is that researchers usually implement their algorithms from scratch, contributing to making all these problems even larger. That is, making it difficult to compare, improve, adapt and use evolutionary algorithms by other people.

The objective of the work presented in this paper is to provide a framework with a broad and tested collection of already implemented evolutionary algorithms, a suit of common benchmarks and a design that makes it easy to implement new algorithms, so that the necessary time to do that is as short as possible. We called this framework Java Evolutionary Algorithm Framework (JEAF) [1].

Java was chosen as the programming language due to the requirement of being able to run the framework everywhere without compiling anything at all (grid computing in mind). Nevertheless, it has been developed also trying to make it as fast as possible. Evolutionary algorithms are usually employed to solve very complex problems, so the framework supports many distributed possibilities to speed up evolutions and it has been developed to make the use of external programs in the individual evaluation phase easy.

Some representative examples of public evolutionary algorithm frameworks are PGAPack, GAUL, TEA, Open BEAGLE, IDEAL and ECJ.

PGAPack [2] was probably the first public library for evolutionary computation. It was made available on February 1996, although received bug fixes (not announced, as there is not a web page for the project anymore) on March 2008 and June 2009. It was written in C and contains only a Genetic Algorithm.

GAUL [3] is written in C for Linux and focused on Genetic Algorithms (although there is also a Differential Evolution implementation) and distributed / parallel computing (island model). Its last version was on October 2006, although it seems there is some coding efforts under the hood (CVS changes on March 2009)


Open BEAGLE [5] is another good evolutionary computation framework developed in C++. It shares some objectives with JEAF, although they select a different programming language, for the good and for the bad. Its development is very intermittent as the last stable version is dated on December 2007, and the last development version (alpha, CVS) is dated on February 2009.
JDEAL [6] is written in Java, but it seems dead since October 1999. It implemented Genetic Algorithms and Evolution Strategies and it supported the master – slave distribution model.

The most similar solution to JEAF is ECJ [7], an excellent environment for evolutionary computation also programmed in Java, but ECJ is somewhat less flexible, and slower with long chromosomes. It is also necessary to emphasize that JEAF includes not only many already implemented common algorithms, but also many benchmarks and utilities, trying to provide a complete solution that minimizes efforts for both evolutionary field researchers and researchers that just use evolutionary algorithms as a tool.

II. MAIN CHARACTERISTICS

This section contains a summary of the main properties of JEAF, which are directly related to the design objectives established for its development:

- **Multiplatform**: JEAF is implemented in Java, so it can be used on any hardware platform where a Java Virtual Machine (JVM) exists without any change. The required version is 1.5 or higher, as generics, autoboxing and other recent features are used.
- **Modular**: the design is profoundly object oriented, maintaining the maximum possible independence between the different components, so that a given component implementation / behavior does not affect the other components.
- **Flexible**: this feature, related with the previous one, is present in every system piece. JEAF was designed not only to support current evolutionary algorithms, but also whatever combination of operators a given user could imagine. Thus, future evolutionary algorithms can be implemented without forcing them to fit into a classical evolutionary algorithm structure and without re-implementing existing components. In fact, it facilitates reusing them. This is achieved by splitting the general concept of evolutionary algorithm into four stages (selection, reproduction, evaluation and replacement) and implementing each stage with an operator chain of arbitrary length. Length 1 or 2 is the most common situation, but every positive integer value is possible, 0 included, as long as interfaces are respected. In addition, many aspects of the already implemented common operators can also be configured with plug-ins, which is a more powerful mechanism than a predefined list of values. Of course, the user can also use this mechanism when implementing new operators.
- **Efficiency**: it was taken into account from the very beginning. Evolutionary algorithms are time consuming, especially if they are applied to some engineering field, and the choice of Java already has a negative impact if we compare that language to C or C++. So, some decisions were carefully taken to maximize the efficiency of the implementation (for instance, chromosomes are arrays of doubles always in the [-1:1] range)
- **Easy-to-use**: taking into account that JEAF was going to be used by other researchers who do not belong to computer science area, it was designed so that users do not need to know its inner working, only the global behavior and the holes that have to be filled for solving a particular problem.
- **Analysis tools**: besides different evolutionary algorithms and benchmark functions, this framework provides several analysis tools and support to implement even more. These tools are developed with the aim of facilitating the comparison of results using different algorithms or different versions of the same algorithm. Evolutionary algorithm analysis tools and function analysis tools are included. They will be commented in detail in section III.C.
- **Distributed processing supported**: JEAF supports a very flexible distribution model for the evolution process, using MPI as the communication protocol (again, efficiency had something to do with this decision). Several models are supported (master-slave, islands…) with different topologies (grid, ring…) and ways of exchanging genetic material. The models supported can also be combined to create hybrid distribution schemes that adapt to the particular problem and / or hardware availability.

III. DESIGN AND IMPLEMENTATION

There are three principal modules in this framework. The first one is the *evolutionary algorithm module*, which implements the main component of this framework. The second one is the *benchmark functions module*, where the user can find the most widely used benchmark functions. Finally, there is the *analysis tools module*, which implements several useful tools that allow users to test the algorithms. All of these modules will be presented in detail in the following sections.

A. *Evolutionary algorithms module*

![Fig. 1- Core of the Java Evolutionary Algorithm Framework (JEAF)](image)

The core of the most important module of this framework is shown in Fig. 1. Its main element is the *EvolutionaryAlgorithm* class. This class is the center of all the operation and it manages the behavior of the algorithm. In order to work, an evolutionary algorithm developed in this framework needs:
- A population, which is made up of a list of individuals.
- Four operator chains, for the selection, reproduction, evaluation and replacement phases of the algorithm. Each of them is made up of a list of operators that are chained when run.
- A problem to be solved, which is defined by, at least, one objective function and a list of constraints if necessary.
- An evaluation strategy, which is in charge of evaluating each individual with the objective functions and the constraint functions, if they exist. It also manages the behavior of the constraint handling method.
- A stop test, which defines the stopping criteria of the algorithm.

All of these components will be explained in detail in the following subsections.

1) Population

A population consists in a list of individuals. In the basic implementation, an individual represents a solution to the problem and it may be encoded as one or more chromosomes. Each chromosome is a list of double values that belong to the range [-1.0, 1.0]. It is the responsibility of the objective function class to normalize them to the corresponding values. This basic individual could be extended in order to add more features in addition to the basic ones, which are the chromosomes and the fitness value. The main advantage of this representation is that the basic individuals are not dependent on the problem. Consequently, if the user does not require more than the basic features, they do not need to encode an individual and they are only responsible for defining the objective functions. Also, this way, the implementation of the evolutionary algorithms is independent of the problem to be solved and that implementation can be optimized taking into account that the representation range will always be the same.

2) Operator Chains

In an evolutionary algorithm, there may be several phases that execute different types of operators: selection, reproduction, evaluation and replacement. In JEAF, each of these phases runs a chain of operators. So, during these phases, the algorithm can execute one or more operators with different characteristics (behaviors) depending on the phase. All the operators should implement a selection behavior in the selection phase, whereas, in the reproduction phase, operators should implement a reproduction behavior, which normally corresponds to the generation of new solutions from the parent population, and so on. The input of the operators is always a list of individuals. The first operator of a given chain receives the list of individuals from the algorithm, while the other operators receive the input from the previous operator. Finally, the algorithm receives the list of individuals from the last operator of the chain. Fig. 2 shows the behavior of the algorithm when a chain is executed.

![Operator chain sequence diagram.](image)

Due to the flexibility of this mechanism, it is not necessary to implement a specific class to represent a new algorithm in JEAF, this can be done by just defining the right chains of operators. Regarding the typical evolutionary algorithms, JEAF currently includes operators and configurations for Genetic Algorithms, Evolutionary Strategies, Differential Evolution, CMA-ES, Micro-genetic Algorithms, Macroevolutionary Algorithms and NSGA2.

As explained in section II, in order to achieve a high level of flexibility, a plug-in system was implemented, which is mainly employed to modify evolutionary algorithm operators, but it can be used at any point in JEAF. This system allows a high degree of customization without barely any coding effort. For example, it allows using a constant parameter or an adaptive parameter without implementing a new operator merely configuring the current operator with a fixed number of values. The current class diagram of the plug-in parameter is shown in Fig. 3. Right now, the following plug-ins are implemented:

- Individual Chooser plug-in: the behavior of this plug-in consists in choosing an individual in the population following a given criteria, for instance, the best individual, the closest individual to another or a random individual.
- NSGA2 Ranking plug-in: this class implements a plug-in that is responsible for calculating the rank position of an individual in the population. It is necessary in the NSGA2 algorithm implementation.
- Crowding plug-in: it is used in the implementation of the NSGA2 algorithm. It calculates the crowding distance of a NSGA2 Individual. There are two subclasses, each one of them calculates the crowding distance based on different parameters of the individual. One of them considers the objective values and the other one the parameter values of the individual.
- Parameter plug-in: this type of plug-in implements different ways of calculating the values of any parameter. Currently, four subclasses are implemented: constant parameters, random valued parameters and two subclasses that represent parameters whose values decrease during the evolution: linear annealing and log
annealing.

- **Stop test plug-in:** the plug-in that extends this class is responsible of returning the number of generations or function evaluations (FEs) currently performed by the algorithm.

**Fig. 3 - Plug-in package class diagram.**

3) **Problem**

A problem in the JEAF framework is defined by one or more objective functions and zero or more constraints. The objective functions are the functions the algorithm should optimize. The constraints are the restrictions of the problem and they may be divided into two classes: boundary constraints and constraint functions. The boundary constraints should be implemented by the user in the objective functions normalizing the gene values in the range of the variable values. The constraint functions, more complex constraints than simple limits on the gene values, should be implemented as a subclass of an equality or inequality constraint and they will be managed by the ConstraintMethod class. To complete the configuration of the problem, the user should also indicate whether it is a maximization or a minimization problem.

**Fig. 4 - Class diagram of the Problem package.**

4) **Evaluation Strategy**

During the evaluation phase, the list of individuals is evaluated using the objective functions and constraints, if they exist. There are several types of problems that an evolutionary algorithm could solve; these types include single and multiobjective problems and constrained and unconstrained problems. The behavior of the evaluation strategy is explained in the sequence diagram displayed in Fig. 5. The evaluation strategy allows dealing with all these different types of problems.

- **Single and Multi-objective problems.**

Single and multiobjective problems are both directly considered in this framework, as the evaluation strategy is implemented to evaluate a list of objective functions. Therefore, it is easy to create a single or a multi-objective problem. In addition to using a list with several objectives or just one objective, the user is in charge of choosing the appropriate operators for each strategy. There is currently an implementation of the NSGA2 algorithm available in the framework.

- **Unconstrained and constrained problems.**

The framework is able to deal with both unconstrained and constrained problems. In the literature we can find different strategies to deal with the constraints (as discussed in the papers presented to the CEC 2007 Special Session [8]) and all of them may be implemented as a subclass of the ConstraintMethod class, which is in charge of evaluating the individuals with the defined constraints and applying the specific constraint handling method.

Every combination of problems is possible; the user could define a single objective or a multi-objective problem, either with or without constraints.

- **Parallelization.**

The evaluation phase is often the most computationally expensive phase of an evolutionary algorithm. To speed up this phase, we have parallelized the evaluation process. Two parallel strategies are implemented within the framework and a more detailed explanation will be provided in section D.

**Fig. 5 - Evaluation strategy sequence diagram.**
5) Stop Test
Like every other iterative optimization method, the evolutionary algorithm needs to have a stopping criterion, which represents a condition to finish execution. Possible stopping criteria are, for instance, a maximum number of generations, a maximum number of calls to the evaluation function, an objective value to be reached, etc. In this framework, the stopping criterion may be implemented as a single one or as composition of criteria. The composition of criteria is implemented as an or operation, i.e., when one of the criteria is fulfilled the algorithm finishes its execution.

B. Benchmark functions module
A benchmarking module is implemented with the aim of bringing together the most commonly used benchmark functions. Currently, the benchmark sets implemented are the following:

- The CEC 2005 Special Session on Real-Parameter Optimization [9].
- The CEC 2006 Special Session on Constrained Real-Parameter Optimization [8].
- The CEC 2007 Special Session & Competition on Performance Assessment of Multi-Objective Optimization Algorithms [10].
- Other well known and highly used benchmark functions that could be found in [12][13].

These benchmark sets include functions with different and interesting features, such as different types of modality, separable and non-separable functions, single and multi-objective problems and constrained and unconstrained problems.

Currently, the benchmark functions corresponding to the CEC 2008 and 2009 Special Sessions [14][15][16] are being adapted so that they can be used in JEAF.

C. Analysis tools module
In addition to be a solver tool for optimization problems using evolutionary algorithms, this framework was developed to be used as an analysis tool with the aim of easily analyzing different evolutionary algorithms or different versions of the same evolutionary algorithm.

1) Evolutionary algorithm analysis tools
In order to make the analysis of an evolutionary algorithm easier, several tools were developed:

- **Population analysis tool**: it allows the user to perform a population analysis in an easy way, executing the same algorithm and the same objective function with different population sizes.
- **Dimensional analysis tool**: it is very similar to the previous one; the difference is that, in this tool, the value that changes is the dimensionality of the problem.
- **Multiple functions analysis tool**: it allows defining a list of objective functions and analyzing an evolutionary algorithm with all of these functions.

- **Composite analysis tool**: this tool allows to concatenate analysis tools and to execute them as a chain.

2) Objective functions analysis tools
This framework is part of a project whose main goal is to perform a characterization of evolutionary algorithms in terms of which type of functions are more suitable to be solved by them. With this aim in mind, two analysis tools for the characterization of objective functions were developed in this framework.

- **Separability analysis tool**: this tool analyzes an objective function in terms of separability or, in other words, in terms of dependency between variables. Separability is a widely used feature in the field of evolutionary algorithms as it is related with epistasis. This tool allows measuring the degree of separability of a function.
- **Modality analysis tool**: modality (how many local and global optima exist and where) is another important feature when characterizing functions to be solved with evolutionary algorithms. With this goal in mind, an analysis tool was developed that estimates the distribution of optima in a given function starting from a random sample of points in the search space.

D. Parallelization
Evolutionary algorithms are stochastic processes with a high parallelization potential. This potential may be exploited in order to reduce the temporal cost of finding a solution to problems or to try to generate more diversity during the search process. With these two objectives in mind, we have added several parallelization components to the framework that make the use of distributed systems for the execution of the algorithms easier.

These components facilitate the quick development, the concise configuration and the easy execution of distributed evolutionary algorithms of three of the four types that have been traditionally identified [17]. These three methods are: the distributed evaluation of the population (global model), the islands model, a coarse grain model, and the combination of the two previous models (hybrid model). These are appropriate for architectures with distributed memory, such as clusters. The fine grain models, that is, the models that require overlapped populations and for which it is highly recommended to use a shared memory architecture, are not supported in this framework.

The main class that manages the parallelization process implements a decorator pattern where the decorated object is any serial evolutionary algorithm. Therefore, every algorithm implemented in the framework and every algorithm developed by the users can be executed in a distributed manner without any additional coding. This is reflected in Fig. 6.
All parallelization classes are distributed into three layers. The lower layer is responsible for message passing. At the moment, the MPJ express library [18] is used in the implementation of this layer. This library is completely Java coded and it implements the MPI 1.2 version specification.

Over it, an intermediate layer is situated that abstracts the communications and allows the upper layer to be independent from the message passing library, minimizing the impact of a replacement of that library. This layer defines several topologies that can be directly used or extended to implement new interconnection models.

Finally, the upper layer has the classes that implement the previously mentioned parallelization models. It also allows the recording of evolution information during the process for each one of the nodes.

Currently, there exist predefined topologies for the island model (totally connected, ring, grid and hypercube connections) and for the distributed evaluation model (master-slave topology).

The island model permits having some quite isolated populations. They communicate through the migration of individuals with the aim of improving the global diversity. There is a migration operator (implemented as a replacement operator) supporting this model, which allows the user to choose the interconnection topology, whether the migration is synchronous or asynchronous, and which individuals are accepted and which are sent to other nodes. The aim of implementing the migration as a replacement operator is that this method allows designing an evolutionary algorithm with different migration models, each one of them represented by an operator.

In the global model, which uses a master-slave topology as a distributed evaluation model, the computation resources are exploited by using both master and slave nodes to evaluate the population. The master nodes share out the population amongst the nodes on demand, sending chunks (with a configured size) of the population. This method allows a more rational use of the distributed system in the case of heterogeneous hardware because it minimizes the timeout produced by a slow node.

Some of the possible configurations for distributing an evolution are shown in Fig. 7.

**Fig. 6 - JEAF parallelization.**

**Fig. 7 – Some parallelization models implemented in JEAF. From left to right: master-slave, islands with a grid topology and islands with a grid topology mixed with master-slave.**

### IV. Usage

JEAF is currently being used in multiple applications to deal with different problems that include evolutionary robotics, automatic design, evaluation of evolutionary algorithms and many others. In what follows we provide a brief description of some of these applications in order to give a sense of the extent of the possible application domain of the framework.

In the field of automatic design, we can cite two examples of projects that make use of this framework for their development. One such project is concerned with the design of the contraction of the tunnel outlet [19] in wind tunnels. The air vent of wind tunnels gives a non-homogeneous airflow, which is, of course, not desirable at all because causes turbulences. So, the objective of this project is to reach a uniform flow applying a contraction in the tunnel outlet, contraction that is obtained automatically through evolution. The evaluation process is carried out using a Computational Fluid Dynamics (CFD) software (OpenFoam [20]), which was easily and successfully integrated with the objective function using SWIG [21].

Another project related with automatic design and the field of fluid dynamics is presented in [22]. The aim of this project is the automatic design of control surfaces in ships. The main contribution of this work is the hybridization of Multiobjective Evolutionary Algorithms (MOEA) and a neural correction procedure in the fitness evaluation stage, which permits obtaining solutions that are precise enough while drastically reducing the computational cost of the simulation stage for each individual. The MOEA searches for the optimal solutions and the neuronal system corrects the deviations of the simplified simulation model to obtain a more realistic design. The proposed hybrid system is successfully applied in the design of a 2D control surface for ships and extended to a 3D one. The MOEA used in this work is the NSGA2 algorithm implemented within the JEAF framework.

A totally different field, which also benefits from evolutionary algorithm research, is the autonomous robotics field. Two projects within this field were developed using the JEAF framework. The Multilevel Darwinist Brain (MDB) is a cognitive architecture that follows an
evolutionary approach to provide autonomous robots with lifelong adaptation. This architecture has been tested in online real robot learning with several examples [23][24] obtaining successful general results that reinforce the evolutionary principles that constitute the main original contribution of the MDB. In particular, evolution is applied in the learning of world, internal and satisfaction models that constitute the knowledge the robot acquires during its lifetime. These models are represented by artificial neural networks (ANNs). In the last version of the architecture, these ANNs were evolved using the JEAF framework, specifically the Differential Evolution implementation [25][26].

Another work in progress at the Integrated Group for Engineering Research, this time in the field of autonomous robotics, consists in the design of modular robotic systems for application in highly unstructured production environments, such as shipyards or building environments. Modular robotic systems are characterized by the low functionality of each individual module and a broad versatility, as modules can be attached with others to obtain different morphologies. However, producing feasible morphologies for a specific task is a complicated job because of the combinational explosion of solutions that can be obtained and due to the fact that the fitness of the morphology is related with the control system. In this project, the authors use the JEAF framework to co-evolve the morphology and the control system of the robot. A robotic simulator, Gazebo, is successfully integrated with the objective function in the evaluation process.

Finally, another field where JEAF has been used is the routing problems field. These algorithms generally consist of a sequential or parallel assignment of nodes to constructed roots. Prioritization rules for assignments are critical for a good performance of the algorithm. In [27][28], the authors present a hyper-heuristic technique based on a neural network model for prioritization rules, which is evolved using the JEAF framework. Although these projects are focused on the VRPTW problem (Vehicle Routing Problem with Time Windows), this technique could be applicable to general routing problems. The algorithm is tested with the Solomon instance benchmark using tests aimed at performance evaluation and generalization capability of the evolved neural network.

As shown, in this section, the JEAF framework is prepared to address different problems providing simple tool that is highly configurable and could be integrated with other applications or software.

V. CONTRIBUTIONS AND IMPROVEMENT

To facilitate the incorporation of user contributions was one of the design objectives from the very beginning of this project. This has led to some design decisions, for instance, the use of plug-ins to personalize algorithms. There are several ways to contribute in this project:

- Improving and extending the evolutionary algorithms contained in this framework. This is possible by adding new algorithms, operators or plug-ins.
- Adding new constraint handling methods. Many real life problems are easier to deal with when using constraints in the definition of objective functions. So, providing additional constraint handling methods will reduce the effort of the users when they define their problems.
- Developing graphical tools like an interactive visual builder for experiments.

VI. CONCLUSIONS AND FUTURE WORK

The framework presented in this work (JEAF) allows developing new evolutionary algorithms (or modifying existing algorithms) with little effort, just re-arranging the chains of operators, modifying the operators using plug-ins or implementing new operators. This way, evolutionary researchers can program their ideas in a small amount of time. JEAF already includes many common algorithms, benchmark functions and utilities so researchers can test and compare algorithms out of the box.

In the performance area, JEAF supports many ways of distributing an evolution and it is developed in a multi-platform language (Java), so users can distribute evolutions over any platform without doing anything special. In general, JEAF is designed to be efficient in order to minimize the disadvantage compared to other frameworks developed in C/C++.

JEAF is not only useful for evolutionary computation researchers; it is designed to be used by researchers of other fields who use evolutionary algorithms just as a tool. In this sense, we enumerate several examples of successful utilization in autonomous robotics and automatic design in fluid dynamics. As future work, it is necessary to design GUIs over the already existing utilities to facilitate its use. A graphical tool for building configuration files will also be developed and further optimization of some parts of the libraries is on the way too.

ACKNOWLEDGMENT

This work was supported by Xunta de Galicia (09DPI012166PR) and European Reg. Development Funds.

REFERENCES
