Evoptool: an Extensible Toolkit for Evolutionary Optimization Algorithms Comparison

Gabriele Valentini, Luigi Malagò and Matteo Matteucci, *Member, IEEE*

***Abstract*— This paper presents *Evolutionary Optimization Tool* (Evoptool), an optimization toolkit that implements a set of meta-heuristics based on the Evolutionary Computation paradigm. Evoptool provides a common platform for the development and test of new algorithms, in order to facilitate the performance comparison activity. The toolkit offers a wide set of benchmark problems, from classical toy examples to complex tasks, and a collection of implementations of algorithms from the Genetic Algorithms and Estimation of Distribution Algorithms paradigms. Evoptool is flexible and easy to extend, also with algorithms based on other approaches that go beyond Evolutionary Computation.**

I. Introduction

Evolutionary Algorithms (EAs) [1] consist of a family of meta-heuristics used in black box optimization, based on the Darwinian paradigm of evolution. They include Genetic Algorithms, Estimation of Distributions Algorithms, Genetic Programming, Evolutionary Strategies, and other related techniques. In order to find candidate optimal solutions, EAs implement biologically inspired methods to evolve, from one generation to another, a population of individuals, that encode feasible solutions for an optimization problem.

Nowadays, several software packages that offer support to the development of an EA are available in different languages (C, C++, Java, Matlab). From the other side, as pointed out in [3], sometimes these packages suffer of a lack of flexibility that prevents them from being widely accepted and used. Moreover they are more oriented towards the implementation of specific algorithms, rather than the comparison of different strategies.

MATEDA [4] is a suite of scripts in Matlab that implement Estimation of Distributions Algorithms (EDAs). This package provides several methods of selection and sampling of the individuals in the population, as well as different model building methods, and a set of local search procedures for both single and multi-objective optimization problems. Other features of MATEDA include mechanisms for data analysis, data visualization, and function approximation. The purpose of the tool is to provide a fast way to develop and test an EDA, by combining the features and functions provided with new ones implemented by the user. This allows the user to evaluate a number of variants of the same algorithm before the final implementation.

GALib [5] is a library that stands out among the others for the use and implementation of GAs in C++. GAlib offers a set of built-in types for the representation of the individuals, including bit-strings, arrays, lists, and trees, it also allows the user to define ad-hoc chromosome-based representations. The library offers several variation operators and various population initialization methods. An interesting feature of this library is the support to the parallelization of the algorithms, from the other side, it lacks the implementation of other paradigms, since it is restricted to Genetic Algorithms.

EOlib, described in [3], is an object-oriented framework that provides a flexible set of classes for EAs. EOlib implements several individual representations, from bit-strings up to parse trees for Genetic Programming (GP), and multilayer perceptrons. The fitness domain is not restricted to built-in types since the library classes are templatized over the fitness type, and thus, it supports any kind of fitness representation. EOlib supplies several variation operators and different ways for combining them, for instance in sequential order, or by applying them only to subportions of the population.

As already pointed out, there are several general purpose libraries and software packages available that support the development of an Evolutionary Algorithm. All of them simplify and speed-up the implementation phase, with different levels of flexibility. The goals of *Evoptool* are basically different, since it addresses the needs of the research community in Evolutionary Computation, to have a common platform for performance comparisons of different algorithms and paradigms on a common set of benchmark problems.

In general, once a new algorithm is designed, it needs to be tested and its performance compared with other algorithms, possibly on the same set of problems. This phase is generally limited by the availability of the implementations of other algorithms, or by the lack of statistics on the same benchmarks. For these reasons, a performance comparison often includes few other algorithms, and is often restricted to algorithms coming from the same EC sub-community. Evoptool overcomes these limitations by offering a common platform for the development of evolutionary paradigms, the test over a set of shared benchmarks, and the analysis of their performance.

Evoptool provides a large set of benchmarks that include both toy problems and more complex tasks, and it is easy to extend with new optimization problems. At this stage it includes various algorithms from popular paradigms, such as GAs and EDAs. The Evoptool architecture is designed to

Gabriele Valentini is a master degree student at Dipartimento di Elettronica e Informazione (DEI), Politecnico di Milano, 20133 Milan, Italy (email: gabriele.valentini@mail.polimi.it).

Luigi Malagò is a Ph.D. student at Dipartimento di Elettronica e Informazione (DEI), Politecnico di Milano, 20133 Milan, Italy (phone: +39 02 2399 4031; email: malago@elet.polimi.it).

Matteo Matteucci is with the Dipartimento di Elettronica e Informazione (DEI), Politecnico di Milano, 20133 Milan, Italy (phone: +39 02 2399 3470; email: matteucci@elet.polimi.it).

facilitate the development process and to reduce the overhead related to the implementation of the benchmarks and other issues, such as statistics computation and visualization.

The rest of this paper is organized as follow: in Section II-A the goals are briefly introduced, Sections II-B and II-C present the algorithms and the benchmarks available in Evoptool, and Section II-D the statistics generated by the tool. In Section III we present Evoptool starting from its libraries (III-A), continuing with the Evoptool engine and the graphical user interface (III-B), and concluding with the algorithms and objective functions (III-C, III-D). Finally, Section IV concludes the paper and illustrates future lines of research.

II. Tool overview

The focus of Evoptool is on the algorithms, the benchmark problems, and the visualization of the statistics after each run. It is possible to execute it either from a graphical user interface, useful during the algorithm development phase, or from command line, better suited for the validation phase on large problems.

A. Analysis of algorithms performance

Even if Evoptool simplifies the implementation of the algorithms, is not intended as a general purpose library for EAs development as the ones mentioned in the previous section. The goal is to provide a platform where the performance comparison is simplified and standardized. In order to achieve this goal, we require a common optimization level among the algorithms, not to mislead the performance analysis with the running times. With optimization level, we refer to those low level optimizations such as multi-threading, compiling options, or the ones related to the machine architecture.

The Evoptool multi-threading engine supports the parallel executions of several algorithms for the comparison purpose, while each algorithm is implemented as a single thread.

The algorithm objects hierarchy, that will be discussed later, offers a single mechanism to access the fitness function, regardless of the particular benchmark problem. This function wraps the call to the benchmark function, and implements a fitness caching mechanism that allows to avoid the re-evaluation of the individuals inherited from the previous generation for which the fitness value has been already computed. The wrapper also counts the fitness calls, considering only the first evaluation for each individual.

The algorithm hierarchy offers a centralized way to compute simple population statistics, such as average fitness population, best and worst individuals, in a unique scan of the population, optimizing the computational costs.

Individuals in the population are represented with bitstring. The Evoptool underlying data structure for the representation exploits the vector class template of C++, which has a particular template specialization for the bool type, and is able to map each allele of an individual with a single bit in memory. This reduces the impact of dynamic memory allocation during the execution.

B. Algorithms

As stated before, Evoptool provides the implementation of a set of algorithms coming from several paradigms in Evolutionary Computation. Within the Genetic Algorithms field [6], we implemented some variants of the Simple Genetic Algorithm [7], [8]:

- *Simple Genetic Algorithm (SGA)*. It is the basic implementation of a genetic algorithm with truncation selection, bit-flip mutation, and single-point crossover.
- *SGA with binary tournament selection*. It is a variation of the SGA that differs for the selection operator that implements a binary tournament selection. Two individuals are picked up at random from the population, and their fitness is compared. The fittest individual is then selected for the new population.
- *SGA with uniform bitwise crossover*. Another variation of the SGA, it maintains the truncation selection and bit-flip mutation, but implements a uniform bitwise crossover. This variation operator compares the corresponding bits of the two individuals and swap them with a fixed crossover probability.

Most of the algorithms implemented inside Evoptool are EDAs [2]. The ones with an underlying univariate probabilistic model include:

- *Population Based Incremental Learning (PBIL)* [9]. It maintains a vector of marginal probabilities which is used to sample a new population at each generation. Then, after a selection of the population, the marginal probabilities are computed again and the vector is updated according to a learning rule.
- *Univariate Marginal Distribution Algorithm (UMDA)* [10]. It can be considered as a particular instance of PBIL, with the learning parameter equal to 1. New individuals are generated using marginal probabilities estimated from the previous population after selection.
- *Compact Genetic Algorithm (cGA)* [11]. It is a space-efficient variation of PBIL that maintains a probability vector, but rather than generating a whole population, only two individuals are sampled at each iteration. The probability vector is then updated according to a learning rule which considers both individuals. Only the probabilities related to those variables of the best individual which assume a value different from the respective variables of the other individual are updated.
- *Univariate Distribution Estimation Using Markov Random Field (Univariate DEUM)* [12]. It employs an univariate Markov Random Field (MRF) to model the relationship between individuals and their fitness. MRF parameters are estimated with maximum likelihood, then the model is used to sample a new population by employing a Gibbs or Metropolis sampler.

The Estimation of Distribution Algorithms implemented in our toolkit are not limited to those with univariate model. In Evoptool we also implemented some bivariate EDAs, which

employ statistical models able to detect interactions between couples of variables, such as:

- *Mutual Information Maximizing Input Clustering (MIMIC)* [13]. It employs a directed conditional chain model, and learns it with a greedy heuristic based on the Shannon's Information Entropy, in order to minimize the Kullback-Leibler Divergence between the model and the true distribution.
- *Combining Optimizers with Mutual Information Trees (COMIT)* [14]. This algorithm employs a tree-structured Bayesian Network rather than a single directed chain as in MIMIC. A maximum weight spanning tree is used to learn the structure of the tree, guided by an entropy-based information principle. Once the structure is defined, individuals are sampled according to the Bayesian Network, and then a fast search algorithm is employed to further improve the fitness of the generated individuals. Inside Evoptool there are two different implementations: one with a hill-climber fast search, and the other with a PBIL based fast search, as proposed in [14].
- *Bivariate Chain DEUM (Bivariate DEUM)* [12]. It differs from the univariate version only for the structure of the underlying probabilistic model that takes the form of a bivariate undirected chain.
- *Ising Model DEUM (Ising DEUM)* [12]. It is a DEUM like algorithm specifically designed to match the structure of a two-dimensional square lattice. This algorithm is well suited for benchmarks with an underlying lattice model, as for the Spin Glass problem.

Besides EDAs with univariate and bivariate models, the set of algorithms provided with Evoptool includes a multivariate EDA which makes use of Bayesian Networks to model higher order interactions among variables:

- *Simple Bayesian Optimization Algorithm (sBOA)* [15]. This algorithm adopts a Bayesian Network to model the fitness structure. A greedy algorithm is employed to search the space of possible networks using the *Bayesian-Dirichlet* metric to measure the network quality. At each generation, a new Bayesian Network is constructed, marginal and conditional probabilities are computed from a selection of the current population, and than new individuals are generated by direct sampling.

The collection of algorithms provided with Evoptool allows a straightforward preliminary analysis of the behaviour of a new algorithm, by comparing its performance with respect to the others. The user may rely on Evoptool for an easy computation of the statistics, which is transparent to the implementation of the new algorithm.

C. Benchmarks

Evoptool is designed to support the study of algorithms for combinatorial or discrete optimization. In particular we focused on boolean function optimization, by implementing a set of benchmark functions which accept as argument a string of bits and returns a real value. In the following, let $x = x_1 \dots x_n$ be a string of bits with n elements, with $x_i \in \{0, 1\}$.

Evoptool provides a set of benchmark problems with different difficulties, varying from classical toy problems to more complex benchmark. This facilitates the evaluation of the behaviour of the algorithms on different types of problems, so that almost all efforts can be spent on the development of the algorithm.

A first set of benchmarks is composed of simple toy problems without any variable dependency that can be encoded by linear functions.

- *One Max* [16]. A simple problem where the fitness value is given by the count of alleles of the individual taking value one. It has a unique optimal solution and every allele has the same relevance in the evaluation of the function.
$$f(x) = \sum_{i=1}^{n} x_i$$
- *Sum Value.* In this toy problem the optimal solution is represented by a string of all ones. The fitness value is computed as the sum of the products between alleles and their position in the string. As in the previous function there are no variable dependencies, but in contrast the relevance of each allele is different and it grows linearly.
$$f(x) = \sum_{i=1}^{n} i x_i$$
- *Binary Value* [17]. Very similar to the Sum Value benchmark, it gives an increasing exponential relevance to each allele.
$$f(x) = \sum_{i=1}^{n} 2^{i-1} x_i$$

Another simple problem that does not admit local minima is:

- *One-Zero Max.* It is a variation of One Max, where the fitness of an individual is represented by the maximum between the count of zeros and the count of ones in the string of bits. The relevant property of this problem is the existence of two optimal solutions: one with all alleles taking value one, and the other with all zeros.
$$f(x) = \max \left\{ \sum_{i=1}^{n} x_i, \sum_{i=1}^{n} (1 - x_i) \right\}$$

Another group of functions consists of problems designed to mislead classical Genetic Algorithms. Their structure defines dependencies among variables of different orders (bivariate and multivariate).

- *Alternated Bits* [18]. Also called 1D Checkerboard, it introduces dependencies between couples of adjacent variables defining a chain structure. The problem takes into account the value of a variable relative to that of its neighbours in the chain, in particular, higher fitness is achieved when adjacent variables take different values.

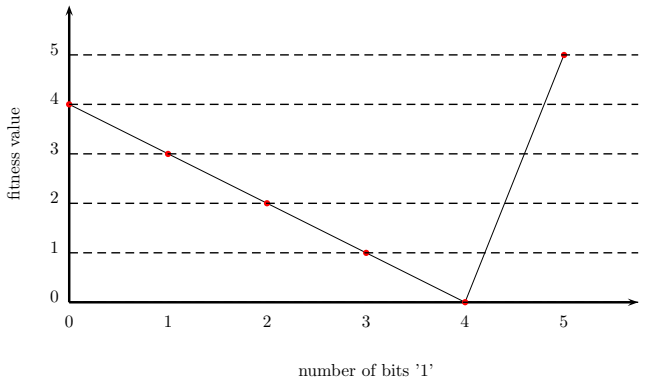

Fig. 1. This figure shows the landscape of the Trap-5 benchmark problem.

- *F3-Deceptive, Overlapping and Bipolar* [19]. This deceptive function is composed of separable adjacent pattern of length three which have one global optimum for all ones and a deceptive local optimum for all zeros. The final fitness value is composed by the sum of the contributions of each deceptive function defined on three consecutive bits. The Bipolar variant is based on blocks of length six, where there are two global optima (all ones and all zeros) and a deceptive local optimum (three ones and three zeros). Overlapping versions introduce overlaps among building blocks.
- *Trap-5* [19]. This benchmark is similar to the previous deceptive functions, and differs for the length of the pattern. The fitness landscape of this benchmark can be seen in Fig. 1. As the previous function, it has a global optimum for all alleles taking value one, and a deceptive local optimum for all zeros.
- *Four Peaks and Six Peaks.* These benchmarks have been specifically designed for EDAs [9], [13]. They have both several global and local optimal solutions, and their difficulty can be choosen by varying the size of the basins of attraction of the local maxima, (through the value of the T parameter). They are defined as:

$$f(x,T) = \max\{\text{tail}(0,x), \text{head}(1,x)\} + R(x,T)$$

where $\text{head}(1,x)$ is the number of adjacent bits taking value one starting from the head of the bitstring, $\text{tail}(0,x)$ is the number of adjacent bits taking value zero starting from the tail of the bitstring, and $R(x,T)$ is the reward function. The reward functions R_4 for Four Peaks, and R_6 for Six Peaks, are defined as follow:

$$R_4(x,T) = \begin{cases} n \text{ if } (\text{tail}(0,x) > T \wedge \text{head}(1,x) > T) \\ 0 \text{ otherwise} \end{cases}$$

$$R_6(x,T) = \begin{cases} n \text{ if } (\text{tail}(0,x) > T \wedge \text{head}(1,x) > T) \\ \quad \vee\ (\text{tail}(1,x) > T \wedge \text{head}(0,x) > T) \\ 0 \text{ otherwise} \end{cases}$$

where T defines the number of adjacent bits taking the same value.

Others similar benchmarks implemented in Evoptool are *Liepins Vose Fully Deceptive* [22] and *Quadratic functions* [24].

Finally, more complex problems that have been included are:

- *MAX 3-SAT.* The Maximum Satisfiability problem is described in [21]. Given a collection of boolean clauses with three literals, the task in this benchmark is to find the instance which maximizes the number of satisfied clauses. The problem is of extreme interest, since many real-world problems can be reduced to MAXSAT, and it is known to be NP-hard.
- *Ising Spin Glass* [20]. The general Ising Spin Glass problem can be described as an energy function minimization problem, defined over a set of spin variables $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_n)$, given a set of coupling constants h and J. The energy function takes the form:

$$H(\sigma) = -\sum_{i=1}^{n} h_i \sigma_i - \sum_{(i,j)\in L} J_{ij}\sigma_i\sigma_j$$

  Each spin variable can be either $+1$ or -1, and the set L determines the lattice structure for the n sites. Given a set of coupling constants, the task in the Ising Spin Glass problem is to find the value for each spin that minimizes the total energy H.
- *Texture Restoration* [23]. The task in this problem it to restore a corrupted monochromatic image. This is achieved by the minimization of an energy function defined as:

$$H(x \mid y) = -\beta \sum_{(i,j)\in L} x_i x_j - \frac{1}{2}\ln\frac{1-p}{p}\sum_{i=1}^{n} x_i y_i$$

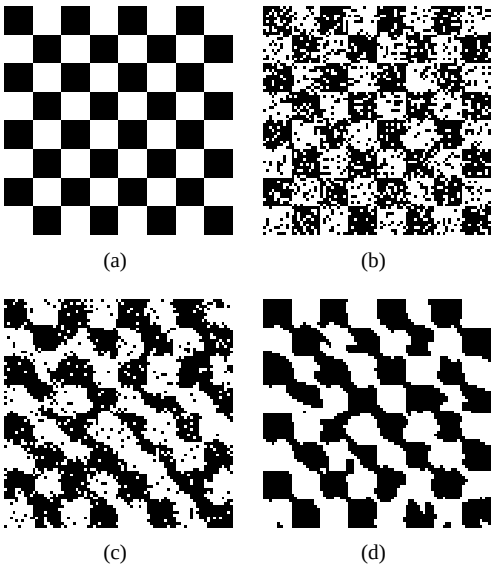

Fig. 2. This figure shows the images associated to the best individual, after 2400 generations, for a Texture Restoration benchmark problem over an image of 80x80 pixels. Algorithms: SGA with population size 500 and mutation rate 0.004, UMDA with population size 300. (a) original image, (b) corrupted image, (c) SGA best individual, and (d) UMDA best individual.

where $y = y_1 \dots y_n$ encodes the value of bits of the corrupted image, β is the smoothness weight, and p the probability of a change of colour of a pixel. This benchmark is not commonly adopted in the Evolutionary Computation literature, but it is interesting for its natural underlying lattice structure, given by pixels adjacencies, and expressed in L. An example of the images produced by Evoptool during the execution of this benchmark is showed in Fig. 2.

It is easy to extend the current set of benchmarks with little effort, thanks to the Evoptool architecture. A description of the steps required to do that will be presented later.

D. Statistics

Evoptool has been designed to collect statistics at each generation of every algorithm execution. In particular it records the fitness value of the best and worst individual in the population, the average population fitness, and the number of calls of the fitness function. Those data can be later combined to produce automatically the following graphics:

- *Average population fitness over generations*
- *Average population fitness over fitness calls*
- *Best and worst individual fitness over generations*
- *Image associated to the best individual* (only for computer vision benchmarks)

Another interesting feature is given by the *multi executions* option, used to run in parallel multiple instances of the same algorithm, and to get more reliable estimations of the performance trends.

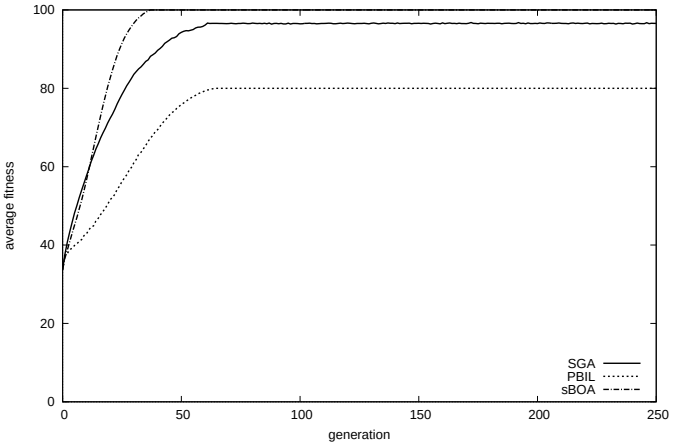

Fig. 3. Average population fitness trend over generations for a Trap-5 benchmark problem of size 100. Algorithms: SGA with population size 500 and mutation rate 0.001, PBIL with population size 1000 and learning rate 0.3, and sBOA with population size 5000 and max incoming edges 2.

In order to facilitate the visualization of the statistics, and the comparison of the behaviour of the algorithms over different benchmark problems, the range of values assumed by each objective function provided in Evoptool is normalized over the interval $[0, 100]$.

The data collected at runtime are used to generate graphics from which it is possible to obtain a sample and preliminary analysis of the algorithm performance over different benchmarks. Some examples of the graphics produced are presented in Fig. 2, 3 and 4.

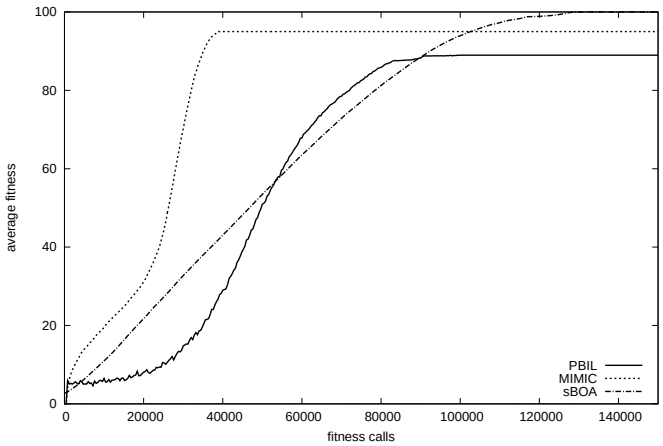

Fig. 4. Average population fitness trend over number of fitness calls for an Ising Spin Glass benchmark problem of size 225 (15x15). Algorithms: PBIL with population size 400 and learning rate 0.3, MIMIC with population size 500 and offspring size 250, and sBOA with population size 5000 and max incoming edges 4.

III. Technical description

Evoptool is an open-source object-oriented framework written in C++, based on several well known libraries and binaries such as *gtkmm*, *opencv*, *gsl*, *libxml++*, and *gnuplot*. Evoptool is distributed under the GNU General Public License, and its code is available for the download on the AIRWiki webpage [27].

A. Modules and features

Evoptool consists of a set of modules that implement the core functionalities of the framework, and in particular supports the implementation of new algorithms and benchmark problems. The main modules are:

- *ga*. It includes the classes that inherit from the abstract class *GA*. This library contains the implementation of the whole set of Genetic Algorithms provided with Evoptool.
- *eda*. The module consists of the abstract class *EDA* and the implementations of the Estimation of Distributions Algorithms that comes with the toolkit.
- *opt-pbl*. This library contains the set of benchmarks. Each class inherits from the abstract class *ObjectiveFunction*, that defines the interface for a benchmark function.
- *gui*. It is the core module. It includes the Evoptool engine and the graphical user interface. It consists of a collection of customized widgets organized in the *AlgorithmDecorator* hierarchy, that allow the integration of new algorithms with the GUI, and other widgets for the visualization of the statistics. The module also

includes the implementation of the *XMLParser* class, used to parse the configuration file necessary for the command line execution.

- *common*. This library offers several general purpose classes and methods. It contains the abstract classes that define the objective functions and the algorithms hierarchies, and for the basic chromosome representation *BinaryInstance*. It also includes an implementation of the Singular Valued Decomposition (SVD) algorithm [26], for the resolution of a system of linear equations. Other features support the conversion between bit-strings and monochromatic images, and the management of permutation masks. Finally, the module includes the *Solver* class, that serves as a wrapper for the execution of the algorithms in a thread safe environment. The Solver is flexible enough to allow the execution of both algorithms implemented in Evoptool and extern binaries[1], and collect and store data for the visualization of the statistics.

Others modules can be easily added to extend the functionalities of Evoptool, for example for algorithms that do not belong to the GAs or EDAs paradigms. In order to support the development of complex algorithms based on probabilistic graphical models, we are planning to implement a library supporting the operations on graphs and trees.

The directory named *evoptool-file* includes a collection of folders for the files related to the executions of the algorithms. These folders are:

- *configuration-examples*. It contains some examples of the XML Evoptool configuration file.
- *glade*. It contains the XML file generated with *Glade*, required by *gtkmm* to build and load the Evoptool GUI.
- *gnuplot*. It includes the gnuplot configuration files generated by Evoptool in order to produce the graphics through system calls to the *gnuplot* binary file.
- *optimization-problem*. This folder contains those files used by the *ObjectiveFunction* classes. For instance, it includes the files with the Spin Glass coupling constants, the images for the Texture Restoration benchmark, and the instances of MAXSAT generated by *SATLIB* [25].
- *temp*. It is a folder for the temporary files generated by Evoptool, and includes the data files for statistics and the graphics.
- *wrapped*. This folder includes the source files of the wrapped algorithms which have not be re-implemented from scratch but that are included in Evoptool. At this stage, it contains a patched version of Simple BOA[2] written by Pelikan.

B. Evoptool engine and graphical user interface

Evoptool is designed to be easily extendible, both in terms of new algorithms and benchmark problems, and also of the statistics computed and visualized. However, in order to allow the parallel execution of several algorithms, and to manage the collection of their statistics, it has an articulated underlying architecture.

The core of the toolkit is the Evoptool engine, implemented inside the *Evoptool* class. The engine is designed to address all the low level tasks related to the execution of the algorithms. It is implemented as a multi-thread process, where each thread is dedicated to a particular task. This set of threads communicates through signals, exploiting the functionality of the *sigc++* library. The suite of thread is composed by:

- *initializerThread*. This thread performs all the operations necessary to initialize a test execution, from cleaning the temporary folders, to loading the *TestParameters* structure.
- *runnerThread*. This thread manages the tests execution, by running a new *solverThread* for each of the selected algorithms, and by waiting for their termination.
- *finalizerThread*. This thread is created by the *runnerThread* at the end of the test execution. It allows to export data in a *tgz* archive, and it manages the computation of the final statistics.
- *stopperThread*. The *stopperThread* is created when the user stops the execution of a test before the maximum number of generations is reached. It simply changes the value of a run condition variable, causing all the *solverThreads* to terminate before the next generation is created.
- *solverThread*. This thread is a simple wrapper for the *Solver* object previously described.

The engine also implements a suite of low level functions, for instance, there are methods to create the *gnuplot* configuration files and invoke it through a system call, handle the memory allocation of the *TestParameters* structure, and manage the single Random Number Generator. Others features offered by this class are the implementation of the signal handlers, a logger for the tool execution, and the operation needed to export the data collected.

It is possible to run Evoptool by command line by typing:

```
./evoptool --nogui testConfiguration.xml
```

In order to execute Evoptool as a command line process a test configuration file must be provided as argument. This file is written in XML format, and it is parsed by the *XMLParser* object, that will initialize a *TestParameters* structure.

During the development phase of an algorithm, it is useful to evaluate its preliminary behaviour by testing it over benchmarks with a limited problem size. In this way, it is possible to fix bugs, and to apply more consistent adjustments to the algorithm. In order to support this need, Evoptool can also be executed with a graphical user interface, so that it is possible to analyse the statistics at runtime. The GUI can be run by typing `./evoptool` without any further parameter. The GUI is designed to be user-friendly. In order to run a test, the user must select a benchmark, set the test parameters,

[1] It is possible to use a wrapper object in order to include external algorithm implementations, as for the Simple Bayesian Algorithm, for which only a patch for the original source code is provided.

[2] Available online at http://www.cs.umsl.edu/pelikan/software.html

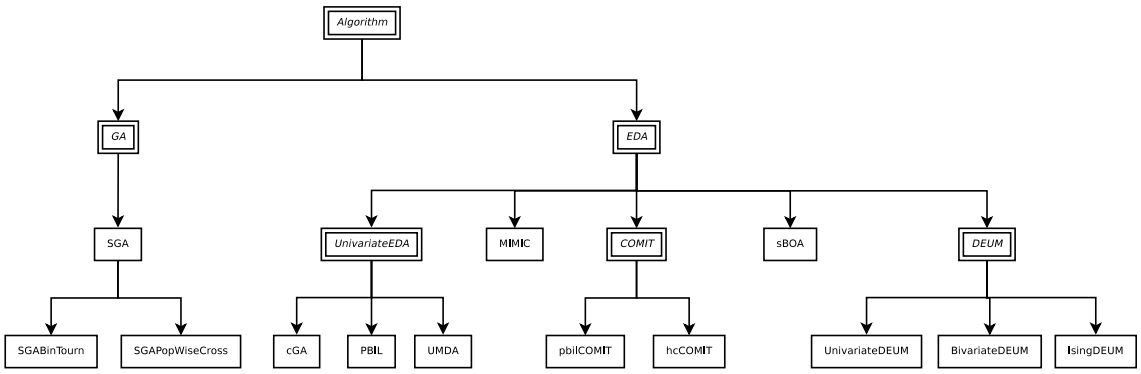

Fig. 5. Algorithms class diagram. Classes represented by a double squared box are abstract class.

select the algorithms and their parameters, and, finally, start the execution.

The class underlying the user interface is named *EvoptoolGUI*, and it implements the functions necessary to collect the test parameters and to communicate them to the engine. The interface can be viewed as a decorator attached to the Evoptool engine, that extends its functionalities.

The graphical user interface is dynamic. The major part of the widgets are created at run-time, as for the benchmark command bar, updated once the benchmark is selected, or for the statistics panel that is managed in different ways according to the selected benchmark. Finally, since the processing of the statistics can be expensive, in order to keep up with the refreshing rate at which the graphics are generated, only those statistics that are currently showed by the GUI are computed.

C. Algorithms hierarchy

The classes which implement the algorithms are organized in a hierarchy (Fig. 5), as the object-oriented programming paradigm suggests, in order to take advantage of inheritance and polymorphism. When a new algorithm is implemented, it is possible to inherit from those classes that already implement some of the desired methods. For instance, the DEUM algorithms inherit from an abstract class *DEUM* and implement only the functions related to the statistical model employed by the specific algorithm. Another example is represented by the *SGABinTourn* and the *SGAPopWiseCross* classes that inherit from the *SGA* class and overwrite, respectively, the selection operator and the crossover operator.

The algorithm hierarchy is headed by the abstract class *Algorithm* that defines some virtual methods that must be implemented by the concrete classes. These methods include the *run* procedure, which defines the single generation step for the algorithm, and other functions related to the mechanism for fitness caching.

The abstract classes *GA* and *EDA* inherit from the class *Algorithm*. These classes define virtual functions that are characteristic for those paradigms, such as *selection*, *crossover* and *mutation* for *GA*, and *selection*, *estimation* and *sampling* for *EDA*. If one of those functions is not necessary for a particular algorithm, the user can simply omits it in the *run* procedure.

In order to extend Evoptool with a new algorithm, there are few simple steps to follow:

1) Choose an algorithm class, either abstract or concrete, from which to inherit and implement the behaviour of the the new algorithm.
2) Create the algorithm decorator for the GUI and add it to *EvoptoolGUI* class.
3) Handle the new algorithm in the *XMLParser* class.

The major part of the code is related to the implementation of the algorithm behaviour, which means that Evoptool framework does not increase significantly the amount of work necessary for the integration of a new algorithm.

D. Objective functions hierarchy

Benchmarks within Evoptool are implemented as classes, in contrast to other libraries which implement the optimization problems as functions. This choice favours a more elegant design that simplifies the implementation of complex benchmark that need ad-hoc structures or articulated evaluation procedures, as for the *TextureRestoration* objective function. Also the optimization problems are organized in a hierarchy. The reason for this choice is also related to the way that the Evoptool engine manages the fitness evaluation and the caching mechanism.

The abstract class from which the whole set of benchmark classes inherits is named *ObjectiveFunction*, and it defines the the virtual function f that represents the fitness function.

In order to add a new benchmark to Evoptool, there are few steps to follow:

1) Implement the benchmark within a class that inherits from *ObjectiveFunction* and implements the method *f*.
2) Integrate the new benchmark within *EvoptoolGUI*.
3) Handle the new function in the *XMLParser* class.

In order to reduce the effort required by the implementation, is possible to skip the second step, and exclude the new implemented benchmark from the set of problems supported by the graphical user interface.

IV. Conclusions

In this paper, we described Evoptool, a flexible and easy to extend evolutionary optimization tool designed to facilitate the comparison of the performance of different algorithms. Evoptool is an open-source software package, which is public and freely available under the *GNU General Public License* (GNU GPL). Further documentation together with a link to the source files can be found on the AIRWiki webpage [27].

In the near future we plan to extend both the set of algorithms and the benchmark problems provided, trying to extend the capabilities of Evoptool to facilitate the analysis and the comparison of meta-heuristics coming from the Evolutionary Computation field. Other future works include an extension of the current set of libraries supporting manipulations of probabilistic graphical models, and the definition of a new set of statistics related to the model structure for the algorithms that make use of probabilistic models, as for EDAs. Also it would be interesting to support non-binary benchmarks by extending the individual representations to strings with finite alphabet or by converting the problem to binary function by a proper reparameterization.

References

[1] T. Bäck, "Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms", Oxford University Press, USA, 1996.

[2] P. Larrañaga and J. A. Lozano, "Estimation of Distribution Algorithms: A New Tool for Evolutionary Computation", Kluwer Academic Publisher, 2002.

[3] M. Keijzer, J. J. Merelo, G. Romero and M. Shoenauer, "Evolving Objects: a general purpose evolutionary computation library", *Lecture notes in computer science*, pp. 231-244, Springer, 2002.

[4] R. Santana, C. Echegoyen, A. Mendiburu, C. Bielza, J. A. Lozano, P. Larranaga, R. Armananzas and S. Shakya, "MATEDA: A suite of EDA programs in Matlab", *Technical Report EHU-KZAA-IK-2/09, Department of Computer Science and Artificial Intelligence*, 2009.

[5] M. Wall, "GAlib: A C++ Library of Genetic Algorithm Components", documentation available online at http://lancet.mit.edu/ga, 1995.

[6] M. Mitchell, "An introduction to genetic algorithms", MIT Press, Cambridge, MA, USA, 1998.

[7] D. E. Goldberg, "Simple genetic algorithms and the minimal, deceptive problem", *Genetic algorithms and simulated annealing*, vol. 74, Morgan Kaufman Publisher, 1987.

[8] T. Jansen and I. Wegener, "The analysis of evolutionary algorithms a proof that crossover really can help", *Algorithmica*, vol. 34, num.1, pp. 47–66, Springer, 2008.

[9] S. Baluja and R. Caruana, "Removing the Genetics from the Standard Genetic Algorithm", *Machine learning: proceedings of the Twelfth International Conference on Machine Learning*, Tahoe City, California, 1995.

[10] H. Mühlenbein and G. Paass, "From recombination of genes to the estimation of distributions: I. binary parameters", *Lecture notes in computer science*, vol. 1141, pp. 178–187, Springer, 1996.

[11] G. R. Harik, F. G. Lobo and D. E. Goldberg, "The compact genetic algorithm", *The 1998 IEEE International Conference on Evolutionary Computation, ICEC'98*, pp. 523–528, 1998.

[12] S. K. Shakya and J. McCall, "Optimization by estimation of distribution with DEUM framework based on Markov random fields", *International Journal of Automation and Computing*, vol. 3, num. 3, pp. 262–272, Spinger, 2007.

[13] J. S. De Bonet, C. L. Isbell and P. Viola, "MIMIC: Finding optima by estimating probability densities", *Advances in Neural Information Processing Systems*, pp. 424–430, MIT Press, Cambridge, MA, 1997.

[14] S. Baluya and S. Davies, "Combining multiple optimization runs with optimal dependency trees", *Tech Report CMU-CS-97-157, Computer Science Department, Carnegie Mellon University*, 1997.

[15] M. Pelikan, D. E. Goldberg and E. Cantu-Paz, "BOA: The Bayesian optimization algorithm", *Proceedings of the Genetic and Evolutionary Computation Conference GECCO-99*, vol. 1, pp. 525–532, Morgan Kaufmann Publishers, San Francisco, CA, 1999.

[16] C. Höhn and C. R. Reeves, "The crossover landscape for the one-max problem", *Proceedings of the 2nd Nordic Workshop on Genetic Algorithms*, vol.605, pp. 27–22, J. Alander Ed, 1996.

[17] S. Droste, T. Jansen and I. Wegener, "On the analysis of the (1+ 1) evolutionary algorithm", *Theoretical Computer Science journal*, vol. 276, num. 1-2, pp. 51–81, Elsevier, 2002.

[18] A. E. I. Brownlee, "Multivariate Markov networks for fitness modelling in an estimation of distribution algorithm", *ph.D. Thesis, The Robert Gordon University*, 2009.

[19] C. Sangkavichitr and P. Chongstitvattana, "Dictionary based estimation of distribution algorithms", *International Symposium on Communications and Information Technologies, ISCIT'07*, pp. 364–369, 2007.

[20] S. K. Shakya, J. McCall and D. F. Brown, "Solving the Ising Spin Glass Problem using a Bivariate EDA based on Markov Random Fields", *IEEE Congress on Evolutionary Computation, CEC 2006*, pp. 908–915, 2006.

[21] D. S. Johnson, "Approximation algorithms for combinatorial problems", *In STOC '73: Proceedings on the fifth annual ACM symposium on Theory of computing*, Texas, United States, pp. 38–49, ACM Press, 1973.

[22] G. E. Liepins and M. D. Vose, "Deceptiveness and genetic algorithm dynamics", *In G. J. E. Rawlins (Ed.), Foundations of genetic algorithms*, pp. 141–151, San Mateo, Morgan Kaufmann, 1990.

[23] G. Winkler, "Image analysis, random fields and dynamic Monte Carlo methods", Springer, 1995.

[24] M. Pelikan and H. Muehlenbein, "Marginal distributions in evolutionary algorithms", *Proceedings of the International Conference on Genetic Algorithms Mendel*, vol. 98, pp. 90–95, 1998.

[25] H. H. Hoos and T. Stützle, "SATLIB: An Online Resource for Research on SAT", *In: I.P.Gent, H.v.Maaren, T.Walsh, editors, SAT 2000*, pp.283-292, IOS Press, 2000.

[26] G. H. Golub and C. Reinsch, "Singular value decomposition and least squares solutions", *Numerische Mathematik*, vol. 14, num. 5, pp.403–420, Springer, 1970.

[27] The Evoptool Team, "Evoptool: Evolutionary Optimization Tool", Artificial Intelligence and Robotics Laboratory, Department of Electronics and Information, Politecnico di Milano, available online at `http://airwiki.elet.polimi.it/mediawiki/index.php/Evoptool:_Evolutionary_Optimization_Tool`, 2010.