Deconstructing GAs into Visual Software Components

Leidy Garzón-Rodríguez
Engineering School
District University of Bogotá
Colombia
lpgarzonr@correo.udistrital.edu.co

Henry Alberto Diosa
Engineering School
District University of Bogotá
Colombia
hdiosa@udistrital.edu.co

Sergio Rojas-Galeano
Engineering School
District University of Bogotá
Colombia
srojas@udistrital.edu.co

ABSTRACT

We envisage Genetic Algorithms (GA) as search-based optimisation techniques encompassing independent bio-inspired operators and representations that are realizable as self-contained deployable computational units. In other words, we think of GAs as a set of software components conforming to a formally–defined evolution-oriented composition model. Furthermore, we imagine such components being assembled on a visual programming–free board, much like prefabricated electronic chips are wired up to build electronic devices. Here we introduce `Goldenberry-GA`, a toolbox of visual software components complying with these premises that has been built over the `Orange` framework for data mining. The paper describes at user-level the suite of new released components (`GeneticAlgorithm`, `InitialPopulation`, `SolutionRepresentation`, `Selection`, `Mutation`, `Crossover`), including working examples that demonstrate some advantages of the reuse and extension principles of its underlying component–based software architecture. It also explains the composition model specification of the toolbox and the software design patterns that were taken into account during its development. A qualitative comparative study with similar Evolutionary Computation frameworks is reported so as to highlight strengths and weaknesses of the toolbox, as well as to point out directions for future work.

`Goldenberry-GA` is open-source under the New BSD License. Downloading and installation guides are available at: `http://goldenberry-labs.org`

Categories and Subject Descriptors

D.2.11 [**Software**]: Software Engineering—*Software Architectures*; I.2.8 [**Computing Methodologies**]: Artificial Intelligence—*Evolutionary Algorithms*

Keywords

Component-based Software Development; Genetic Algorithms Software; Visual Programming.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

GECCO '15, July 11 - 15, 2015, Madrid, Spain
© 2015 ACM. ISBN 978-1-4503-3488-4/15/07…$15.00
DOI: http://dx.doi.org/10.1145/2739482.2768466

1. INTRODUCTION

Evolutionary computation (EC) techniques rely on the iterative application of stochastic–guided exploration/exploitation operators over a set of candidate solutions to an optimisation problem. These operators along with the encoding/decoding mappings can be seen as loosely-coupled computational modules designed according to a particular EC model. We believe such modules can be seen as self-contained software *components* with properly-specified communication *interfaces*, enabling users to quickly implement EC programs by piecing together compatible blocks, all the more useful if feasible within a graphical simulated board enabling their visual inspection and execution. This is evocative of how hardware devices are built by glueing prefabricated electronic chips through their connecting pins while overlooking the details of their encapsulated circuits.

Motivated on these goals, we have embarked on deconstructing an emblematic EC technique, the Genetic Algorithm (GA), into a collection of software components that we call `Goldenberry-GA`. To this end, we firstly designed the composition model, architecture and interfaces specifications; subsequently we built the suite of software components featuring a user-friendly visual programming environment, provided by the `Orange` data–mining software workbench [6]. Our ultimate purpose is to provide EC users with a new tool to benefit from the principles of component–based software development, both at novice or expert level through visual assemblage of existing components or object-oriented programming of new components and experiments.

The paper starts describing the suite GA visual software components through some illustrative working examples, then discusses briefly its underlying composition model and architecture, along with its extension capabilities, strengths and weaknesses compared to other existing GA software tools. It closes suggesting possible avenues of future development.

2. OVERVIEW OF THE TOOLBOX

2.1 Prior work

The toolbox is integrated within `Orange`, an open-source component-based framework for data mining [6], which was chosen as deployment environment because it features a user-friendly visual *canvas* for component's assemblage. As far as we know, to this date `Orange` does not include tools for solving optimisation tasks with GAs or other stochastic search-based algorithms, although a preliminary incarnation of `Goldenberry` focusing on Estimation of Distribution Algorithms (EDA) was previously released [16].

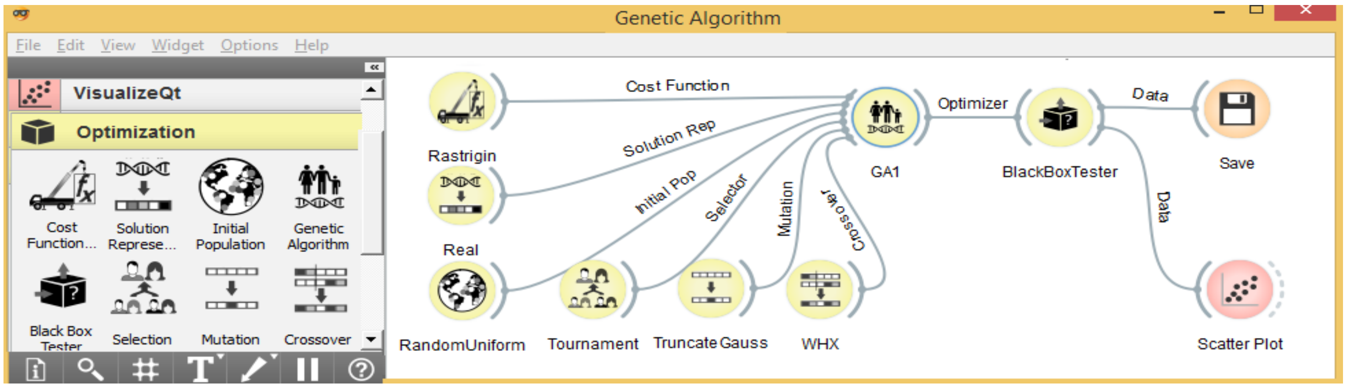


Figure 1: GA scheme assemblage using the visual Goldenberry-GA software components.

2.2 The suite of GA visual components

The analysis, design and development of the toolbox was carried out considering representations and genetic operators related to the simple GA [10, 12], which are summarised in Table 1. The resulting suite of components along with a visual program illustrating the assemblage of a GA, are shown in Figure 1. Some of their front-end configuration screens are shown in Figure 2. A brief description of the suite is given next. Technical and end–user specifications of the contracts components must comply during assemblage (sets of interfaces required and provided by each component) are available at: `http://goldenberry-labs.org/widgets`.

`GeneticAlgorithm`. The core component controlling the execution of a GA. It defines six input interfaces allowing assemblage with `CostFunctionBuilder`, `InitialPopulation`, `SolutionRepresentation`, `Selection`, `Mutation` and `Crossover`. Additionally, it provides a user interface for configuration of running parameters such as operator application probabilities, number of evaluations, population size and replacement strategy, as well as visualisation of single–run results (see Figure 2, middle–top). On the other hand, this component exposes an output interface involving services allowing the execution, statistics gathering and best solution retrieval from the GA. We remark that the only required visual connection is to the `CostFunctionBuilder` component. If no other input connection is detected, the GA runs with a default configuration (binary coding, uniform random initial population, tournament selection, one bit mutation with $p_m = 0.01$, one point crossover with $p_c = 0.6$, 50 candidates, 80 generations and generational replacement). Besides, notice that the execution's flow of the GA is independent of the assemblage order with other components; in fact, it only relies on the availability of such connections in an asynchronous mode (when the `GeneticAlgorithm` is run, every other connected component sends a signal indicating their available services at the time of execution).

Table 1: Simple GA representations/operations

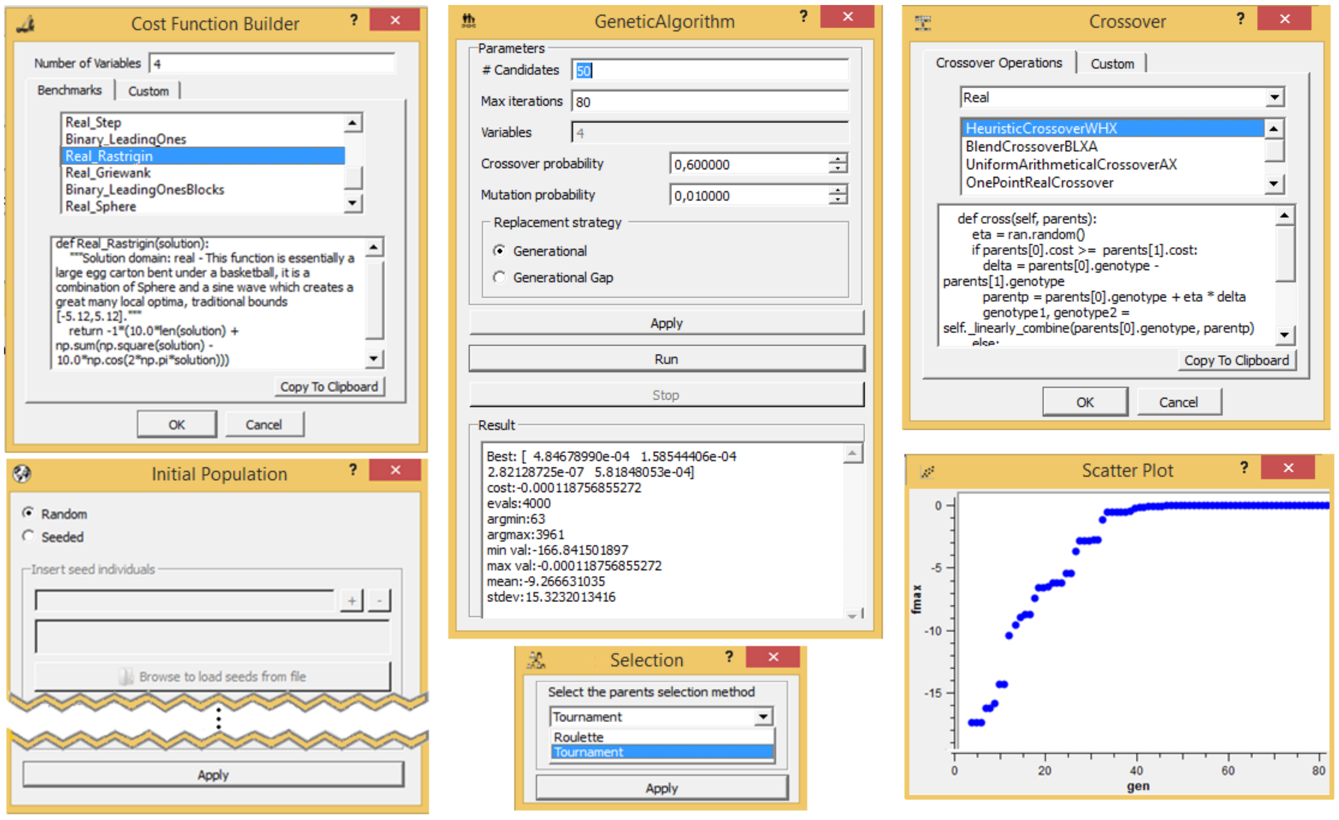
Solution encoding	Binary, Real
Initial population	Random, Seeded, Biased
Replacement strategy	Generational, Generational gap
Selection	Tournament, Roulette wheel
Crossover	Binary: One point, Two points Real: AX, BLX-α, BLX-α-β, WHX Custom
Mutation	Binary: Flip one bit, Multiple bits Real: Gaussian, Truncated Gaussian Custom

`CostFunctionBuilder`. The component allows the definition of the function to be optimised (fitness function). Its user interface includes a built–in library of standard benchmark functions and a text box for code–injection of custom functions written as a Python script, including a convenient copy–paste option on these scripts (see Figure 2, left–top). Its output interface can be connected to the `GeneticAlgorithm` or any other optimiser component. Since other optimisers may not need a solution representation mapping, the number of problem variables must be defined here.

`SolutionRepresentation`. This component is used to define the encoding of the candidate solutions into chromosomes as well as the genotype–phenotype mapping. The available representations are: direct real mapping, real scaled to the unit interval, direct binary mapping and grey–code mapping. Its output interface complies with the corresponding `GeneticAlgorithm` input interface. The final length of the chromosome would depend on the number of variables defined in `CostFunctionBuilder` and the chosen mapping.

`InitialPopulation`. This component specifies several strategies for generation of the initial population, including: random values (uniformly distributed over the range defined for the genes), seeded (particular instances of candidate solutions are inserted into the initial population) and biased (preferred subspaces represented by prototype individuals defined with specific means and variances). Its configuration user interface is shown in Figure 2, left–bottom. Its output interface can be provided to the `GeneticAlgorithm`.

`Selection`. The component establishing the selection operator. The available options are roulette wheel and stochastic tournament (code–injection of custom selection techniques is intended for future versions). Its configuration user interface is shown in Figure 2, middle–bottom. Its output interface can be connected to the `GeneticAlgorithm`.



Number of Variables 4

```
def Real_Rastrigin(solution):
    """Solution domain: real - This function is essentially a
large egg carton bent under a basketball, it is a
combination of Sphere and a sine wave which creates a
great many local optima, traditional bounds
[-5.12,5.12]."""
   return -1*(10.0*len(solution) +
np.sum(np.square(solution) -
10.0*np.cos(2*np.pi*solution)))
```

[✓ Random] Seeded

Insert seed individuals

Parameters
Candidates 50
Max iterations 80
Variables 4
Crossover probability 0,600000
Mutation probability 0,010000
Replacement strategy: [✓ Generational] Generational Gap

Result

```
Best: [  4.84678990e-04   1.58544406e-04
2.82128725e-07   5.81848053e-04]
cost:-0.000118756855272
evals:4000
argmin:63
argmax:3961
min val:-166.841501897
max val:-0.000118756855272
mean:-9.266631035
stdev:15.3232013416
```

Real

```
    def cross(self, parents):
        eta = ran.random()
        if parents[0].cost >=  parents[1].cost:
            delta = parents[0].genotype -
parents[1].genotype
            parentp = parents[0].genotype + eta * delta
            genotype1, genotype2 =
self._linearly_combine(parents[0].genotype, parentp)
        else:
```

Select the parents selection method: Tournament

Figure 2: User interfaces for some of the `Goldenberry-GA` components in Figure 1. Clock-wise from bottom left: `InitialPopulation`, `CostFunctionBuilder`, `GeneticAlgorithm`, `Crossover`, `ScatterPlot`, `Selection`.

`Crossover`. The component defining the settings for the recombination operator. Depending on the solution representation, its user interface provides a library of built–in widely–known operators, such as one point and two point crossover for binary chromosomes, or heuristic crossover (WHX), blend crossover (BLX-α and BLX-α-β) and uniform arithmetical (AX) crossover for real–coded chromosomes (see Figure 2, right–top). Likewise the `CostFunctionBuilder`, this component also enables Python script code–injection for further customisation of the crossover operation. Again, its output interface can be provided to the `GeneticAlgorithm`.

`Mutation`. A component similar to the previous one, but focusing on the definition of the mutation operator. The available operators for binary genotypes are: one bit and multiple bit flip mutation. In the case of real–coded genotypes it provides the truncated Gaussian mutation [15]. Additionally, Python script code–injection is also enabled.

`BlackBoxTester`. This component is intended to visualise and compare results between multiple executions of several `GeneticAlgorithms` or other optimisers, which must be provided as input interfaces. The user front–end collects the results in a table format summarising the statistics of the multiple repetitions or the details of individual runs. These tables can also be exported as an output interface to the `Orange`'s `Save` component.

2.3 Working examples

This section illustrates the use of the toolbox in solving some typical benchmark optimization problems by assembling GAs combining different genetic operators. We defined an experimental protocol similar to [3], as follows.

Problems. Two continuous–domain problems were chosen, Rastrigin's (F_1) and Griewank's (F_2), with dimensionality $d = 4$. We report normalized fitnesses, $F_i = \dfrac{1}{1-f_i}$, where:

$$f_1(\mathbf{x}) = -\left(10d + \sum_{k=1}^{d} x_k^2 - 10\cos\left(2\pi x_k\right)\right);\ \mathbf{x} \in [-5.12, 5.12]^d$$

$$f_2(\mathbf{x}) = -\left(1 + \sum_{k=1}^{d} \frac{x_k^2}{4000} - \prod_{k=1}^{d} \cos\left(\frac{x_k}{\sqrt{k}}\right)\right);\ \mathbf{x} \in [-600, 600]^d$$

Since these are originally minimisation problems, we maximise their negations, as suggested in [12].

GAs. Four GAs obtained as the combination of two selection (Roulette Wheel and Stochastic Tournament) and two crossover operators (WHX and BLX-α-β), were tested:

$$GA_1 \equiv (\text{RW+WHX}); \quad GA_2 \equiv (\text{RW+BLX-}\alpha\text{-}\beta);$$
$$GA_3 \equiv (\text{ST+WHX}); \quad GA_4 \equiv (\text{ST+BLX-}\alpha\text{-}\beta).$$

GA parameters. Population size: 50 individuals; termination criterion: 80 generations per 100 repetitions; replacement strategy: generational; $p_c = 0.6$ and $p_m = 0.01$.

Chromosome representation. Direct real mapping with 4 genes (4 dimensions or variables).

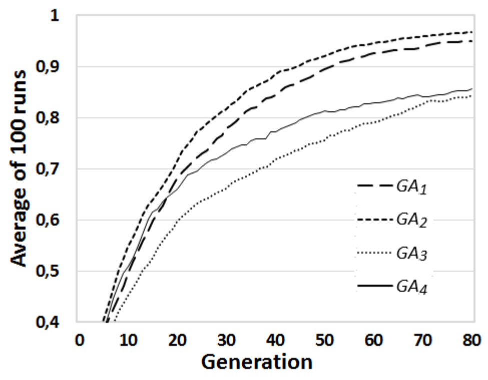


Figure 3: Average GA performance in solving F_1.

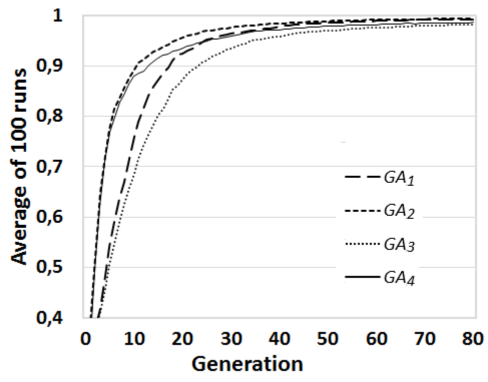


Figure 4: Average GA performance in solving F_2.

	10th generation ($\mu \pm \sigma \times 10^{-1}$)		80th generation ($\mu \pm \sigma \times 10^{-1}$)	
	F_1	F_2	F_1	F_2
GA_1	0.518±1.327	0.788±1.243	0.949±0.938	0.992±0.095
GA_2	0.565±1.197	0.905±0.543	0.967±0.632	0.993±0.068
GA_3	0.466±1.453	0.716±1.488	0.844±1.353	0.982±0.132
GA_4	0.527±1.276	0.885±0.562	0.858±1.055	0.986±0.101

Table 2: Average GA statistics over 100 repetitions.

Figure 1 shows the visual scheme that was assembled for solving problem F_1 with GA_1; similar assemblages were made for GA_2, GA_3 and GA_4, and also for problem F_2 (schemes available at: http://goldenberry-labs.org/benchmarks). The results were collected using the BlackBoxTester component and exported to a tabular data file using the Save component, so as to conduct a graphical analysis with a special–purpose plotting tool. Observe that a quick inspection of the results was also possible by glueing the output to the ScatterPlot component (see Figure 2, bottom–right). Worthy of attention is the smooth integration of Goldenberry-GA with the three aforementioned third–party components.

GAs average performance is reported in Figure 3 and 4, demonstrating their effectiveness in solving these problems (GA_2 performing slightly better). Besides, instantaneous statistics of the GAs at the 10th and 80th generations are reported in Table 2, corroborating the average successful convergences towards the optimum (again, GA_2 exhibits faster convergence rates).

3. SOFTWARE ARCHITECTURE

Goldenberry-GA was conceived using the software architectural style of independent components [2], a design concept where software is built as a set of self–contained computational units that communicate to each other through messages, in order to request and provide services (in our case, pertaining to a GA system). Such messages are sent and received through ports or *interfaces* that are negotiated between the components within the local execution environment (in our case, the environment is provided by Orange).

The *descriptive architecture* was obtained using the Unified Process and Unified Modeling Language adapted to component–based software development [5]. The initial result of the inception stage in the process was the Goldenberry-GA *prescriptive architecture*, which defined the platform–independent conceptual components of the GA prior to development (the interested reader is referred to [14] for details of such prescriptive architecture).

3.1 Descriptive architecture

The descriptive architecture was built in the elaboration stage, when design and technological decisions are taken (e.g. the choice of the object-oriented paradigm for component development, Python as the programming language, Orange as the execution environment, reusing some existing components) on the basis of the aforementioned prescriptive model. As a result we obtained the component–based *descriptive* architecture that is shown in Figure 5. Both prescriptive and descriptive architecture models are available at http://goldenberry-labs.org/models.

The architecture comprises two layers. On the one hand, the business logical layer corresponds to the inner components implementing the actual computations carried out by the GA (tagged with the "Mgr" suffix). As it can be seen in the figure, the core inner component GbGAMgr coordinates the execution of the GA. The surrounding components provide output interfaces for each service required by the GA when it is executed. Five inner components were designed for this purpose: GbSolutionRepMgr, GbInitPopMgr, GbSelectorMgr, GbCrossoverMgr and GbMutatorMgr.

On the other hand, the user–interaction layer consists of visual components (tagged with the "Widget" suffix) that are deployable over the Orange graphical canvas. They serve as visual companions to the inner components in the business layer. Each widget exposes the interfaces or ports they provide for assemblage with other widgets in the canvas. Additionally they expose the external interfaces that realize dialogue windows allowing users to setup parameters and visualise the output of particular services (ISolutionRepWidget, IGAWidget, IBlackBoxWidget, IInitPopWidget, ISelectorWidget, ICrossoverWidget, IMutatorWidget and ICostFunctionWidget). Examples of realizations of such user interface windows are shown in Figure 2.

Each visual component specifies a formal contract of its interfaces, enabling its connection with other components. However, some restrictions such as the admission of genetic operators by a particular chromosome representation must be ensured by the user, as currently the software verifies contract agreement only at the abstract level of the interface specification; if the actual realizations are incompatible, a runtime error would occur.

Lastly, notice that GbCostFunctionWidget and GbBlackBoxWidget were provisioned from the previous release [16].

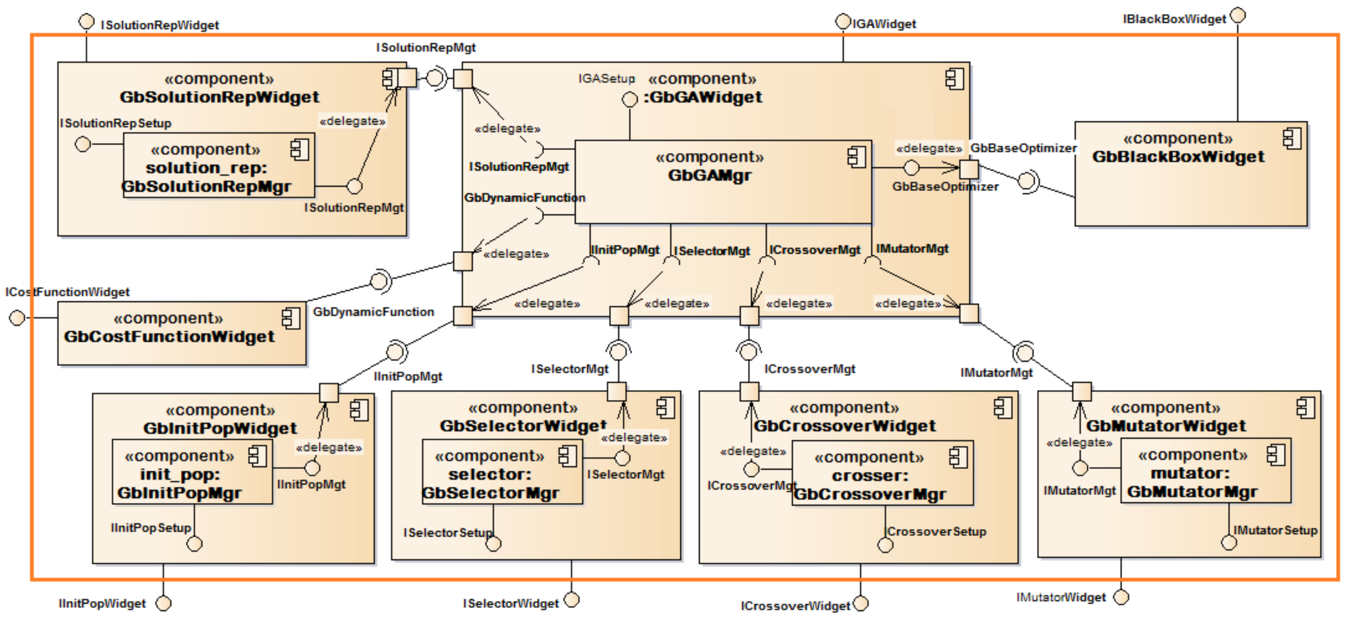

«component» GbSolutionRepWidget
«component» solution_rep: GbSolutionRepMgr
«component» :GbGAWidget
«component» GbGAMgr
«component» GbBlackBoxWidget
«component» GbCostFunctionWidget
«component» GbInitPopWidget
«component» init_pop: GbInitPopMgr
«component» GbSelectorWidget
«component» selector: GbSelectorMgr
«component» GbCrossoverWidget
«component» crosser: GbCrossoverMgr
«component» GbMutatorWidget
«component» mutator: GbMutatorMgr
«delegate»
ISolutionRepWidget
ISolutionRepMgt
IGAWidget
IBlackBoxWidget
ICostFunctionWidget
IInitPopWidget
ISelectorWidget
ICrossoverWidget
IMutatorWidget
IGASetup
GbBaseOptimizer
GbDynamicFunction
IInitPopMgt
ISelectorMgt
ICrossoverMgt
IMutatorMgt
ISolutionRepSetup
IInitPopSetup
ISelectorSetup
CrossoverSetup
IMutatorSetup

Figure 5: The descriptive architecture of `Goldenberry-GA`.

The widget/business component duality also determines two layers of user interaction. Firstly, the widget layer is intended for novice users preferring a programming–free workbench where swift GA realizations can be done with minimal effort by just visually connecting pre-fabricated components. Secondly, the business components layer allows expert users for rapid prototyping of new intuitions or computational techniques embodied within the existing or new derived components using scripting and programming skills. Both layers admit extension mechanisms for customisation purposes, as we shall explain in later sections.

3.2 Design patterns

During elaboration of the `Goldenberry-GA` architecture, the following software design patterns were applied [7].

Factory method. The *factory method* pattern was applied for the design of the `GBSolutionRepMgr` component, as it is shown in the class diagram of Figure 6. An abstract class `Encoder` encapsulates the two methods required for genotype to phenotype mapping. Specific mapping techniques are implemented as realisations of this class, for example `DirectMappingScaledUnitInterval`, `GreyCode-Mapping` and so on. The factory class is `EncoderFactory` which contains the method `get_encoder()` that instantiates a concrete object of any of these encoders. In this way, the `SolutionRep` class can delegate into the factory the calls to the actual mapping operations, depending on the encoder chosen by the user. This approach enables `Goldenberry-GA` to easily extend the set of available solution representations: the user just needs to add a Phyton class implementing the data structures and algorithms for the new genotype–phenotype mapping as long as it implements the `Encoder` abstract class. Then, in its next execution, the `GbSolutionRepMgr` component will recognise it and make it available in its corresponding widget. This pattern was also applied to implement the `GbInitPopMgr` component.

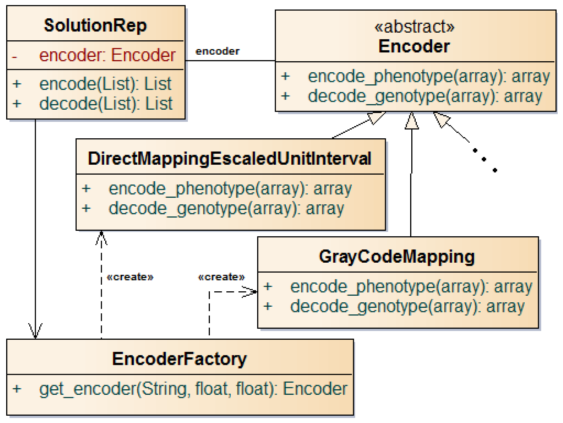

SolutionRep
- encoder: Encoder
+ encode(List): List
+ decode(List): List
encoder
«abstract» Encoder
+ encode_phenotype(array): array
+ decode_genotype(array): array
DirectMappingEscaledUnitInterval
+ encode_phenotype(array): array
+ decode_genotype(array): array
GrayCodeMapping
+ encode_phenotype(array): array
+ decode_genotype(array): array
«create»
«create»
EncoderFactory
+ get_encoder(String, float, float): Encoder

Figure 6: Excerpt of a model applying the factory method pattern to the encoding of solutions.

Strategy. The *strategy* pattern was applied for the design of the `GbMutatorMgr` component, as it is shown in the class diagram of Figure 7. In this case, the pattern allows the component to change dynamically its mutation strategy during execution of the GA. Here an abstract class `Mutator-Strategy` encapsulates the `mutate()` method that would be implemented as concrete realizations of this class, for example `OneBitBinaryMutation`, `TruncateGaussMutation` and so on. The strategy class is `MutatorContext` which is able to instantiate during runtime a particular mutator strategy, depending on the choice stored as a `String` in the `mutation_method` attribute of the `Mutator` class. When the `GbGAMgr` uses the `IMutatorMgt` interface, the latter delegates into the `MutatorContext` the instantiation of the mutator strategy according to the name assigned to the `mutation_method` attribute at that point. The pattern was also applied to the `GbCrossoverMgr` and `GbSelectorMgr` components so as to allow changing or adapting the different genetic strategies dynamically during execution of a GA.

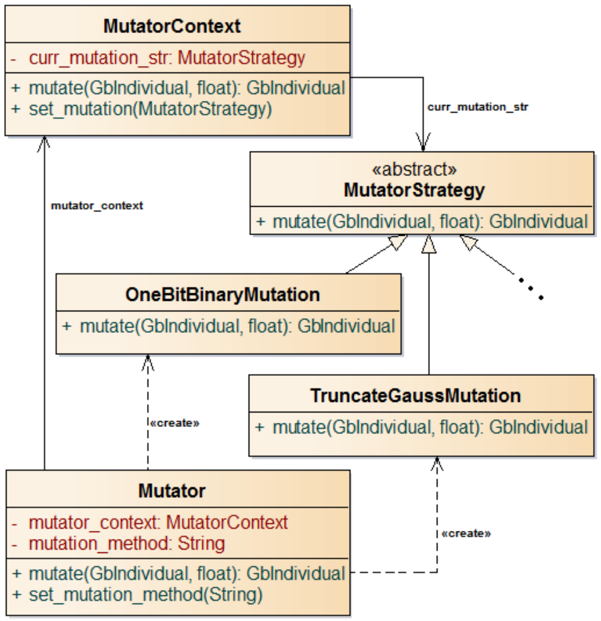

MutatorContext
- curr_mutation_str: MutatorStrategy
+ mutate(GbIndividual, float): GbIndividual
+ set_mutation(MutatorStrategy)
curr_mutation_str
mutator_context
«abstract»
MutatorStrategy
+ mutate(GbIndividual, float): GbIndividual
OneBitBinaryMutation
+ mutate(GbIndividual, float): GbIndividual
TruncateGaussMutation
+ mutate(GbIndividual, float): GbIndividual
«create»
«create»
Mutator
- mutator_context: MutatorContext
- mutation_method: String
+ mutate(GbIndividual, float): GbIndividual
+ set_mutation_method(String)

Figure 7: Excerpt of a model applying the strategy pattern to dynamically set the mutation operator.

As an aside comment, the complete set of structural and behavioural models applying these patterns for each component are available at http://goldenberry-labs.org/models.

3.3 Architectural advantages

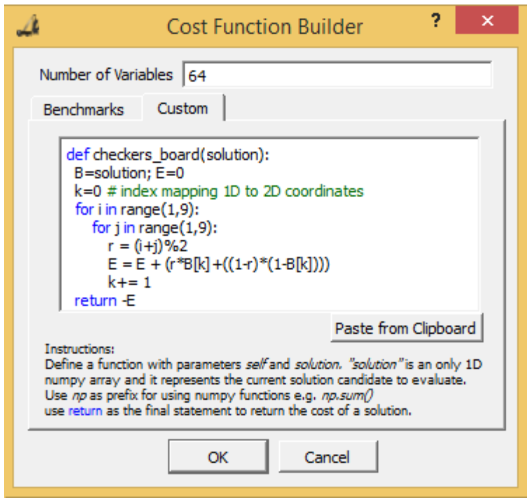
Owing to its underlying architecture, `Goldenberry-GA` exhibits two important software quality attributes [9]:

Maintainability. This attribute is verified easily from the point of view of *repairability*, because any faulty component can be fixed in an isolated way without compromising the remainder components of the GA system. Similarly from an *evolvability* perspective, any obsolete component can be replaced with another fresh component implementing updated technologies, as long as it complies with the contractual interface model of the old one. For example, it would be possible to build a new compatible `GbCrossOverMgr` component using a distributed computing model, provided that it exposes an `ICrossOverMgt` and `ICrossOverSetUp` interfaces and its contractual specification.

Reusability. Since each component is a detachable independent computational unit, it can be assembled into other metaheuristics schemes complying with the `Goldenberry-GA` compositional model. As a matter of fact, some components such as `CostFunctionBuilder` or `BlackBoxTester` can be reused indistinctly in GAs or in EDAs. Accordingly, we anticipate that several `Goldenberry` components would be reusable in other EC methods such as memetic algorithms or local search metaheuristics.

4. EXTENSION CAPABILITIES

The toolbox can be adapted to particular needs in three different levels of extensibility. The basic level allows end-users to define custom cost functions for the GA to optimise. The intermediate level allows GA researchers to try out new operators and representations that can be specialised from



Number of Variables 64

```
def checkers_board(solution):
 B=solution; E=0
 k=0 # index mapping 1D to 2D coordinates
 for i in range(1,9):
    for j in range(1,9):
       r = (i+j)%2
       E = E + (r*B[k]+((1-r)*(1-B[k])))
       k+= 1
 return -E
```

Instructions:
Define a function with parameters *self* and *solution*. *"solution"* is an only 1D numpy array and it represents the current solution candidate to evaluate.
Use *np* as prefix for using numpy functions e.g. *np.sum()*
use return as the final statement to return the cost of a solution.

Figure 8: Example of customising a cost function.

the component collection comprised in the toolbox. The advanced level addresses the implementation of new stochastic search components based on the `Goldenberry-GA` software architecture. Each level of extensibility is explained next.

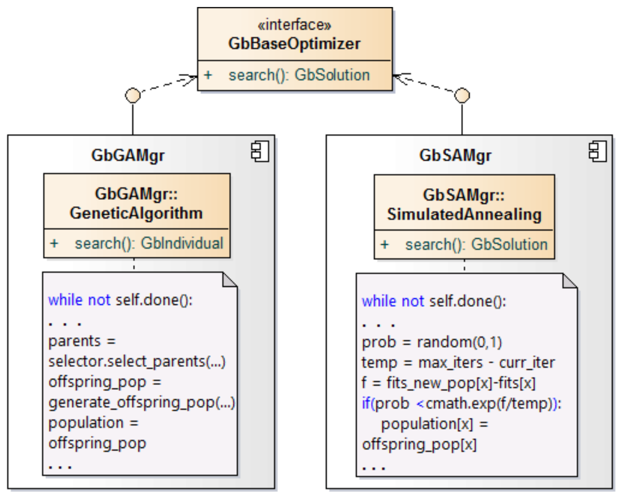
4.1 Cost function customisation

This is the core functionality of the `CostFunctionBuilder` component. Apart from including a predefined library of benchmark problems (*Sphere, Rastrigin, Rosenbrock, Schwefel, Step, Griewank, OneMax, ZeroMax, LeadingOnes*), the user is able to inject a Python function script specifying the computation of the fitness (cost) of a candidate `solution` given as an input parameter to the script. The function should comply with the mapping and domain that would have been previously defined in the `SolutionRepresentation` widget. The easiest way of writing the function is to make a copy of one of the scripts in the library of problems, and then tailor it to the custom cost function. For example, suppose we want the GA to find the optimal configuration of 64 white and black tiles on a 2D board according to the following energy function (which is actually a parity-function defining the energy of a standard checkers board):

$$E(B) = \sum_{i=1}^{8}\sum_{j=1}^{8} r_{ij}b_{ij} + (1 - r_{ij})(1 - b_{ij}),$$

where board $B = \{b_{ij},\ 1 \le i, j \le 8\}, b_{ij} \in \{0\,(\text{white}), 1\,(\text{black})\}$ y r_{ij} is the cell parity, $r_{ij} = (i + j) \mod 2$. The corresponding Python script is illustrated in Figure 8, where the 2D board is mapped to the 1D vector `solution`, and the final cost is reported as the negation of E, because again, instead of maximising we seek to minimise the function.

There are two interesting observations worth to mention here. The first one is that little knowledge of programming are needed to define the custom cost function; just basic notions of algorithmic structures (`if-else` conditionals, `while` or `for` loops, arithmetic operators) and its syntax in Python which is conventional. The second observation is that the `CostFunctionBuilder` component provides an output interface that can be connected not only to the `GeneticAlgorithm` but also to other optimisers, such as the components `cGA`, `UMDA` or `TILDA` from the EDA release [16].



```
while not self.done():
. . .
parents =
selector.select_parents(...)
offspring_pop =
generate_offspring_pop(...)
population =
offspring_pop
. . .
```

```
while not self.done():
. . .
prob = random(0,1)
temp = max_iters - curr_iter
f = fits_new_pop[x]-fits[x]
if(prob <cmath.exp(f/temp)):
    population[x] =
offspring_pop[x]
. . .
```

Figure 9: Excerpt of a model extending a GA (left) to a Simulated Annealing component (right).

4.2 GA component specialisation

As mentioned before, this level of extensibility caters for GA researchers experimenting with tailor-made versions of the standard operators or encoders or conceiving new ones. Two choices are available for this end. First, the user may add a new operator to the predefined libraries in the `Crossover` or `Mutation` components, by using the same Python-script code–injection described earlier for the custom fitness function. The second choice is to resort to the aforementioned strategy pattern, by adding concrete genetic operator objects implementing new algorithms or data structures.

4.3 Brand–new EC components

This is the highest abstract level of extensibility, providing researches with the possibility of deriving components for known or novel EC or stochastic-based search techniques, using the `Goldenberry-GA` architecture as a framework of reference. To exemplify this point, let us assume that we want to implement a new component for the `SimulatedAnnealing` metaheuristic [11]. To this end, we would have to extend the class model of the existing `GbGAMgr` component in the way shown in Figure 9. As it can be seen, both `GbGAMgr` and `GbSAMgr` realise the `GbBaseOptimiser` class with a particular implementation of the `search()` method which would be dependant on the corresponding metaheuristic. The complete implementation and examples of visual schemes using this new `SimulatedAnnealing` component are available at http://goldenberry-labs.org/sa.

5. COMPARISON WITH OTHER TOOLS

In this section we attempt to highlight the main differences with some of the established GA and other metaheuristic optimisation frameworks, also known as MOFs. When unambiguous, we will refer to the combined releases of GA and EDA components as simply `Goldenberry`. The comparative study was carried out considering ECJ, ParadisEO, EvA2, FOM, JCLEC, Opt4j, EasyLocal, MALLBA and HeuristicLab, following the six criteria and scoring check–lists defined in [13]:

C1:	Metaheuristics techniques
C2:	Adaptation to the problem and its structure
C3:	Advanced characteristics
C4:	Global optimisation process support
C5:	Design, implementation and licensing
C6:	Documentation and support

The results of the evaluation of `Goldenberry` are summarised in Figure 10 along with the scoring of the other MOFs as originally reported in [13]. The scores in each criterion are scaled to the unit interval and summed up to a maximum of 6; `Goldenberry` obtained a overall score of 2. The details of our evaluation in each criteria are available at http://goldenberry-labs.org/mof-eval.

It can be seen that the toolbox is particularly strong in criteria C5 and C4 as expected, since these are related to software engineering best practices aspects (C5), and graphical user interface, interoperability and experiment design (C4). In these respects `Goldenberry` compares favourably with ECJ, ParadisEO, JCLEC and Opt4j (C5), and with JCLEC, OAT and HeuristicLab (C4). Now regarding C2, `Goldenberry` scores unsurprisingly adequate, taking into account that this criteria is strongly related to the GA aspects it was designed for, including solution encoding, genetic operators and fitness function customisation; in this sense the toolbox is on par with HeuristicLab, EVA2 and ParadisEO.

By contrast, it is evident that the main weaknesses are related to C1 and C6. Concerning C1, `Goldenberry` ranks lower than the other MOFs. There is ample room to improve here, since in its current release it only involves GA and EDA techniques. In this respect we expect to take advantage of the extensibility mechanisms described earlier. On the other hand, C6 is also another avenue of improving as the toolbox is in its early stage of community awareness and dissemination. Lastly, `Goldenberry` currently lacks supporting of advanced characteristics such as hybridisation and distributed computing; thus, its performs poor in criterion C3, where other tools excel, particularly ParadisEO, FOM or MALLBA.

An additional distinction worth to remark is that as far as we understand, none of the other evaluated MOFs devoted to GAs was built from a visually-enabled component architecture faithful to the component-based software development principles [5]; in fact, most of them adhere to an object-oriented or structured programming paradigm. Some initiatives such as OSGiLiath [8], MALLBA [1] and Para-

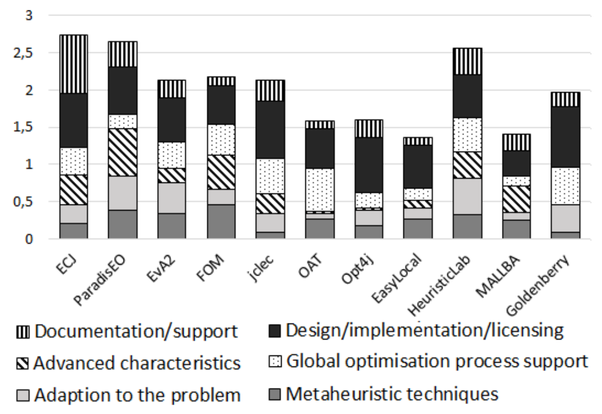


Figure 10: `Goldenberry` vs. other MOFs

disEO [4] in spite of being component-based libraries, only support scripting mechanisms for component assemblage. Furthermore, although some tools feature visual programming environments (e.g. the *operator graph* of HeuristicLab [17]) allowing users to connect computational *modules*, ultimately such programs represent *dataflows* with no predefined assumptions or guarantees on the intermediate computational steps involved throughout. Rather than a dataflow, the visual assemblage in `Goldenberry` represent provision and consumption of services precisely specified as contracts that each component is liable to comply, constituting a guarantee of reusability in other EC schemes conforming with said composition specification.

It is also important to note that `Goldenberry-GA` was not originated as a new framework, but as an add-on to the existing `Orange` platform [6], with a view to extend its application domain to the field of EC optimisation. In this regard it is anticipated that the suite of `Goldenberry` components can be integrated as function optimisers in other higher-level data mining analysis. We hope with this tool to promote cross-fertilizations between these two fields, from the researcher and practitioner point of view.

Nevertheless, we are aware that in its current release, `Goldenberry` is work in progress towards a fully developed and robust EC framework. The comparison study has pointed out some of the current strengths that can be improved in new versions. And more importantly yet, the evaluation has also highlighted flaws worthy of attention in future work, in order to improve its encouraging overall scoring of 2/6.

6. CONCLUSION AND FUTURE WORK

Two interesting features of `Goldenberry-GA` are worth to emphasize. Firstly, its programming–free graphical interface is advantageous for practitioners looking for a friendly visual environment where GA experiments can be easily being arranged, executed and visualised with virtually no coding needed, only wiring up prefabricated components. Secondly, the toolbox is also suitable for skilled users researching new genetic operators or algorithms than can be extended or composed programmatically from the suite of existing components. The latter is possible because the toolbox complies with the openness and reuse principles of component–based software development and design patterns. The application of these principles was demonstrated by the integration with the `Orange` framework (including incorporation into its canvas specification and reusing of existing data–processing components such as `ScatterPlot` and `Save`) and the extension of the proposed architecture to implement other stochastic search algorithms (`SimulatedAnnealing`).

In the same direction, being a new tool aimed at EC enthusiasts and researchers, it is expected that the release of `Goldenberry-GA` into the `Orange` data-mining platform may promote the development of a community market of evolutionary or stochastic population–based or bio–inspired components building upon the proposed descriptive architecture. This is in fact accordant with the ultimate goal of the component–based software paradigm, in the sense of making easier the creation of high–quality EC programs by reusing or adapting off–the–shelf components supplied by specialised software factories or laboratories. In this direction, an obvious immediate target would be the development of new components for combinatorial representations and operators, multi–objective fitness function builders and other metaheuristics [12]. It is also alluring to address the weaknesses and to improve on the strengths discussed in previous sections. Clearly, enabling code injection in the `Selection` component and automatic verification of agreement between genetic representations and operations using the *mediator* design pattern [7], are a priority.

As a closing remark, equally interesting would be to explore the benefits of using other programming paradigms in the implementation of the component–based architecture. In its current release `Goldenberry-GA` uses the object–oriented paradigm in Python, but for critical–mission tasks or time–efficiency constraints, the declarative and concurrent approaches for component implementation might be advantageous. Notice that such an endeavour would require no changes on the component prescriptive architecture which is independent of language or programming technology.

Acknowledgements. We thank the anonymous reviewers for their thoughtful observations and suggestions.

7. REFERENCES

[1] E. Alba, G. Luque, J. Garcia;Nieto, G. Ordonez, and G. Leguizamon. MALLBA, a software library to design efficient optimisation algorithms. In *Int. J. Innov. Comput. Appl.*, pages 74–85, 2007.

[2] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Pearson Education, 2012.

[3] D. Broglio Carvalho, J. C. Nunes Bittencourt, and T. D'Martin Maia. The Simple Genetic Algorithm performance: A comparative study on the operators combination. In *INFOCOMP 2011, The First International Conference on Advanced Communications and Computation*, 2011.

[4] S. Cahon, N. Melab, and E.-G. Talbi. ParadisEO: A framework for the reusable design of parallel and distributed metaheuristics. *Journal of Heuristics*, 10(3):357–380, 2004.

[5] J. Cheesman and J. Daniels. *UML Components: A Simple Process for Specifying Component-Based Software.* Component Software Series. Addison-Wesley, Londres, 2000.

[6] J. Demšar, T. Curk, A. Erjavec, Črt Gorup, T. Hočevar, M. Milutinovič, M. Možina, M. Polajnar, M. Toplak, A. Starič, M. Štajdohar, L. Umek, L. Žagar, J. Žbontar, M. Žitnik, and B. Zupan. Orange: Data mining toolbox in Python. *Journal of Machine Learning Research*, 14:2349–2353, 2013.

[7] R. J. Gamma Erich, R. Helm and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[8] P. Garcia-Sanchez, J. Gonzalez, P. Castillo, M. Arenas, and J. Merelo-Guervos. Service oriented evolutionary algorithms. *Soft Computing*, 17(6):1059–1075, 2013.

[9] C. Ghezzi, M. Jazaheri, and D. Mandrioli. *Software Qualities and Principles*. Chapman and Hall/CRC, 2004.

[10] D. E. Golberg. *Genetic algorithms in search, optimization, and machine learning.* Addion wesley, 1989.

[11] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[12] S. Luke. *Essentials of Metaheuristics*. Lulu, 2nd. edition, 2013.

[13] J. A. Parejo, A. Ruiz-Cortés, S. Lozano, and P. Fernandez. Metaheuristic optimization frameworks: a survey and benchmarking. *Soft Computing*, 16(3):527–561, 2012.

[14] L. G. Rodriguez, H. A. Diosa, and S. Rojas-Galeano. Towards a component-based software architecture for Genetic Algorithms. In *Proceedings of the 9th Colombian Computing Conference (9CCC)*, pages 1–6. IEEE, 2014.

[15] S. Rojas-Galeano and N. Rodriguez. A memory efficient and continuous-valued compact EDA for large scale problems. In *Proceedings of GECCO '12*, pages 281–288. ACM, 2012.

[16] S. Rojas-Galeano and N. Rodriguez. Goldenberry: EDA visual programming in Orange. In *Proceedings of GECCO '13*, pages 1325–1332. ACM, 2013.

[17] S. Wagner, G. Kronberger, A. Beham, M. Kommenda, A. Scheibenpflug, E. Pitzer, S. Vonolfen, M. Kofler, S. Winkler, V. Dorfer, and M. Affenzeller. *Advanced Methods and Applications in Computational Intelligence*, volume 6, chapter Architecture and Design of the HeuristicLab Optimization Environment, pages 197–261. Springer, 2014.