Goldenberry: EDA Visual Programming in Orange

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

Nestor Rodriguez
Engineering School
District University of Bogota
Bogota, Colombia
nearodriguezg@correo.udistrital.edu.co

ABSTRACT

`Orange` is an open-source component-based software framework, featuring visual and scripting interfaces for many machine learning algorithms. Currently it does not support Estimation of Distribution Algorithms (`EDA`) or other methods for black-box optimization. Here we introduce `Goldenberry`, an `Orange` toolbox of `EDA` visual components for stochastic search-based optimization. Its main purpose is to provide an user-friendly workbench for researchers and practitioners, building upon the versatile visual front-end of `Orange`, and the powerful reuse and glue principles of component-based software development. Architecture of the toolbox and implementation details are given, including description and working examples for the components included in its first release: `cGA`, `UMDA`, `PBIL`, `TILDA`, $\texttt{UMDA}_c$, $\texttt{PBIL}_c$, `BMDA`, `CostFunctionBuilder` and `BlackBoxTester`.

`Goldenberry` is open-source and freely available at:
`http://goldenberry.codeplex.com`.

Categories and Subject Descriptors

D.2.6 [**Software**]: Software Engineering—*Programming Environments*; I.2.8 [**Computing Methodologies**]: Artificial Intelligence—*Problem Solving, Control Methods and Search*

Keywords

Component-based evolutionary software systems; EDAs

1. INTRODUCTION

Visual environments for machine learning (e.g. Clementine or SPSS Modeler [9], Weka [7], RapidMiner [12]) provide graphical workbenches to conduct user-friendly data analysis. Instead of scripting commands in an imperative computer language, users are able to graphically sketch processing units and interactions that are needed to run said analysis. `Orange` is one of such open-source visual frameworks, originally proposed for functional genomic analysis [4]. It has been progressively enriched with additional visual software components (*widgets*) for several machine learning tasks. It is known that many of these tasks can be casted as, or make use of, optimization problems in their underlying machinery (regression analysis, regularization, clustering, feature selection); however as far as we know, to this day `Orange` does not include a set of tools available to model explicit optimization problems within the context of machine learning analysis. The latter is precisely the motivation behind the software system we introduce in this paper: `Goldenberry`, a supplementary machine learning and evolutionary computation suite for `Orange`.

The software was built by taking advantage of three interesting features in `Orange`: (i) its versatile visual front-end; (ii) the powerful reuse and glue postulates of component-based software development in which it is based; and (iii) its conformity to the *open/closed* principle of object-oriented programming through an scripting interface to Python. We reasoned that such features would be advantageous for extending its functionality to the realm of optimization problems. Since our research group has been recently working on stochastic-search-based optimization, particularly in the field of Estimation of Distribution Algorithms (`EDA` [10, 15]), a decision was made to start off the project in its first stage by focusing on this kind of algorithms. As a result, the first release of `Goldenberry` comprises a set of `EDA` components (`cGA` [8], `UMDA` [13], `PBIL` [2], `TILDA` [17], $\texttt{PBIL}_c$ [19], $\texttt{UMDA}_c$ [6] and `BMDA` [14]) and a set of utility components (`CostFunctionBuilder` and `BlackBoxTester`) that we shall describe in the following sections. The second stage of the project is currently under development; it will comprise other black-box optimization techniques and also additional meta-heuristics and machine learning components.

The paper describes a general depiction of the software architecture, and provide working examples of its operation using standard benchmark and custom optimization problems. The latter was made possible by using the built-in cost functions or the free-text Python input mode from the `CostFunctionBuilder` component, which are convenient design features allowing this suite to be applied over a wide range of discrete and continuous optimization domains. `Goldenberry` is also open-source and is freely available at:
`http://goldenberry.codeplex.com`
For download instructions please refer to the Appendix.

2. THE SOFTWARE AT A GLANCE

`Goldenberry` was built as a suite of *widgets* for `Orange`. Widgets are visual elements that can be dragged onto the `Orange` visual programming board, also known as *canvas*.

GECCO'13 Companion, July 6–10, 2013, Amsterdam, The Netherlands.

Furthermore, widgets provide a set of input/output interfaces that encapsulate related services; many widgets collaborate to perform a given task. "Programming" in the canvas amounts to manually wiring up the appropriate interfaces between widgets. An example of a `Goldenberry` program to optimize a cost function using three different `EDA` widgets is shown in Figure 1. In that program, widgets are the components or processing units needed to perform the optimization task. Rather than depicting a *dataflow* between these components, connections represent provision and consumption of services encapsulated as objects, which are associated with each input/output interface. For instance, in this program the `cGA`, `PBIL` and `UMDA` widgets require a function to be optimized that is provided by the `CostFunctionBuilder` widget through three instances of the `CostFunction` object, one per `EDA`, each in charge of keeping statistics of the number of function evaluations and running times. The `EDA` components are responsible of setting up the parameters of their corresponding algorithm and of providing a ready-to-use `Optimizer` that is in turn, consumed by the `BlackBoxTester`. The latter is in fact responsible of orchestrating the execution of these `Optimizers`, that is, of running each `EDA` algorithm (whose parameters, including cost function, must have been already defined and provided in their output interfaces before execution begins) and also of collecting and visualizing the final results. Parameter settings and data presentation are embodied within the the graphical user interface of the widgets, as we shall depict in Section 5 and 6 (also Figure 6 to Figure 11). Intuitively, this programming paradigm is very much the same as building a hardware apparatus: the user first gather, connect and set-up the required units before switching-on the resulting device.

3. ARCHITECTURAL VIEW

3.1 General design considerations

The following software patterns were taken into account during the conception of `Goldenberry`:

- A *layering* pattern [3] was applied in order to decouple the algorithmic logic of the components from their user interface (widgets); thereby all the implemented algorithms can be used and integrated in plain Python scripts or using the command-line.

- A *template method* pattern [5] was used by identifying commonalities of the different `EDA` approaches. In this way a generic abstraction of an `EDA` procedure was defined; concrete details of implementation were deferred to each particular algorithm. A unified `EDA` framework was achieved.

- A *responsibility splitting* pattern [1] was utilised in `EDA` optimisers so as to separate the search metaheuristic from the probability estimation technique. Therefore probability models were made reusable among multiple `EDA`s (those included in this and future releases).

- A *dynamic binding* [1] together with an *interpreter* pattern [5] were applied in order to allow the user to provide customised cost functions at runtime, in the form of Python-like scripts.

The application of such patterns guided the resulting design of the software, as described in the remainder of this section.

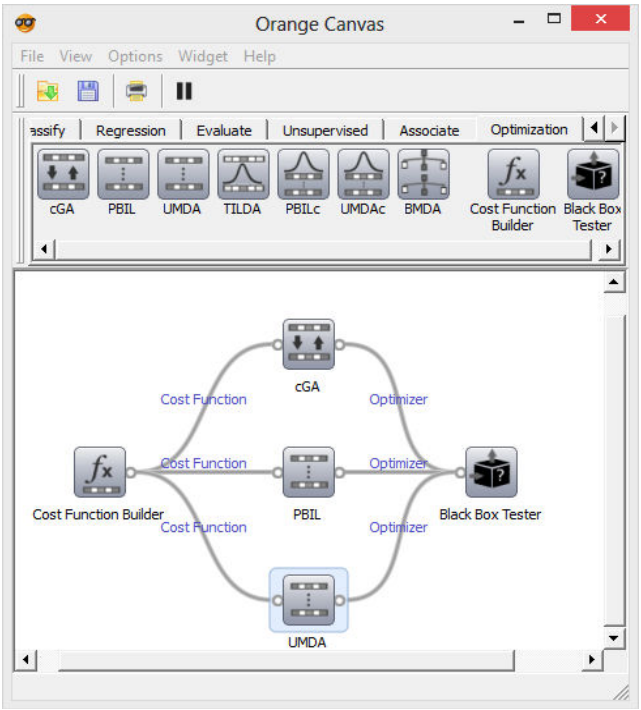


Figure 1: A `Goldenberry` program in the `Orange` canvas.

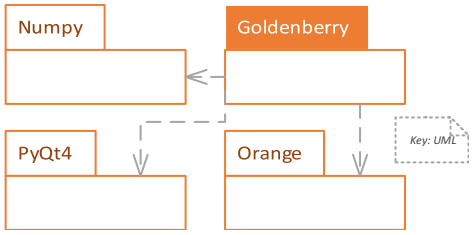


Figure 2: The context diagram of `Goldenberry`.

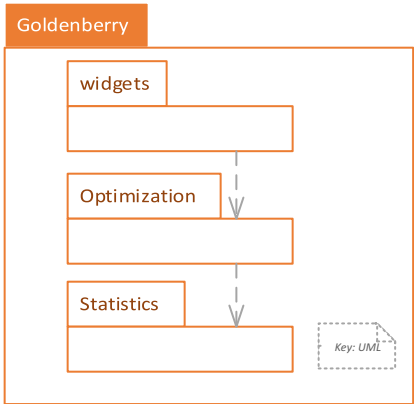


Figure 3: The three basic modules in `Goldenberry`.

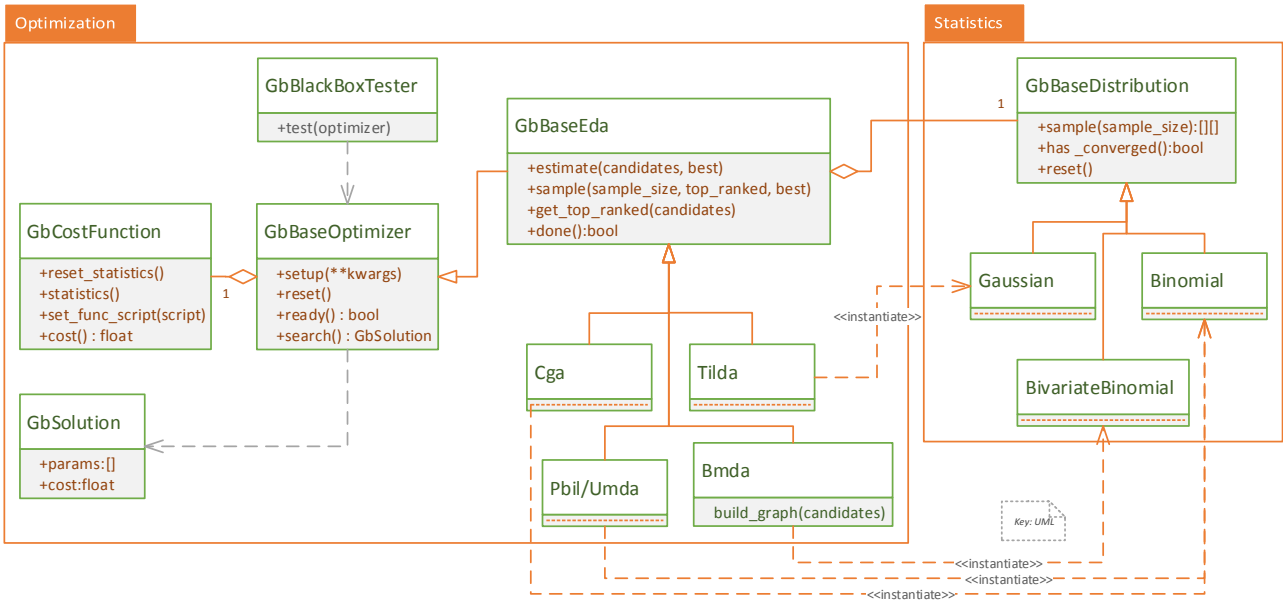


Figure 4: Decomposition view of `Goldenberry` modules.

3.2 Architecture packages

Widgets are actually visual wrappers for software components written in the Phyton scripting feature provided by `Orange`. Implementation of any widget or suite of widgets require the use of the `Orange` core library, the `PyQt4` library for user-interface designing tools and optionally, the `NumPy` library for additional scientific computation support (`PyQt4` and `NumPy` are standard libraries of the Python programming language). Therefore, the top-level organization of the `Goldenberry` suite of widgets is illustrated in the context diagram of Figure 2.

The internal architecture of the current release is shown in the module diagram of Figure 3. Three modules were developed. The `widgets` modules are the visual wrappers for the suite of `Goldenberry` components. The `Optimization` module includes objects and routines needed to implement the `EDA` components; this is the core module of the suite. Finally, `Statistics` is a utility module to allow modelling of some probability distributions used by the `EDA` algorithms.

4. STRUCTURAL VIEW

A decomposition view of the `Optimization` and `Statistics` modules including object classes and dependencies, is shown in the class diagram of Figure 4. It is expected that the modular design proposed here, would allow for new or customized `EDA` algorithms, probability distributions or other stochastic-search optimizers, to be added to the toolbox as extensions of such architecture. This would be one of the advantages of observing the software patterns earlier mentioned in Section 3.1.

4.1 Base classes

A core class `GbBaseOptimizer` was designed as an abstract class representing any black-box optimization algorithm (see Figure 4). The optimizer uses a `GbSolution` object, which holds the values of the parameters or variables in an arbitrary solution to a given problem (the vector `params[]`); it also holds its associated cost. The optimizer additionally encapsulates a `setup()` method to initialize its running parameters (e.g. problem size or number of variables, maximum number of evaluations, etc.) and the `reset()` method to set up the optimizer for a new run. The key method `search()`, defines the actual optimization algorithm which returns a best found solution; `ready()` is a checkpoint method to validate readiness of the optimizer to start the search.

A given candidate solution is evaluated with the `cost()` method from the `GbCostFunction` class (in the evolutionary computation literature this would be equivalent to computing the *fitness* of a candidate). The routine to evaluate the function is defined via the `set_func_script()` method. Furthermore, this class also includes a method to keep track of `statistics()`, such as the number of cost function evaluations or max/min/mean cost values; the `reset_statistics()` method clears up the statistics working memory.

4.2 EDA classes

In the current release `Goldenberry` provides implementation of only `EDA`-type optimizers. Other black-box optimization techniques would be added in the near future. Hence, a specialized abstract class `GbBaseEda` was derived from `GbBaseOptimizer` (see Figure 4). This class defines the two distinctive methods of any `EDA`: `estimate()` as the mechanism that builds up its probabilistic model, and `sample()` as the algorithm to generate new candidate solutions from that model. Two additional methods were designed: `get_top_ranked()` selects the promising candidates from where the probability model estimation is updated; and `done()`, which checks for convergence of the estimated model. These are the methods that would be iteratively executed during a run of the `search()` method from the parent class.

A particular implementation of the `GbBaseEda` class aforementioned, determines a type of `EDA` algorithm that would be used as optimizer. `Goldenberry` features a number of concrete `EDA` implementations, including univariate algorithms

such as `cGA`, `TILDA`, `PBIL` and `UMDA` (discrete and continuos-domain variants for the last two), and also a bivariate algorithm, the `BMDA`. The latter extends the base class with an additional method `build_graph()` to model pairwise dependencies between problem variables. This assortment of algorithms was chosen so as to incorporate techniques using both univariate and bivariate probability distribution approaches. In the first release of the software we emphasised in univariate versions due to their algorithmic simplicity; nonetheless, future releases will build upon these algorithms and contemplate higher-order `EDA`s using tree-based, Bayes and dependency networks techniques.

For illustration purposes, we shall now outline some of the Python scripts implementing these classes. Firstly, let us recall the *Population-Based Incremental Algorithm* (`PBIL`) [2]. The aim of this algorithm is to discover a real-valued probability vector from which a population of competent binary candidate solutions can be sampled. The algorithm starts-off with a random-valued vector; then, the vector is iteratively sampled in order to refine the model using the most promising solutions from the sample and an *incremental learning* re-estimation rule, until the vector converges to a fixed distribution. Our rendering of this procedure is shown in Algorithm 1, were a vectorized operation mode was assumed.

Algorithm 1 `PBIL`

Requires: Cost function $\mathsf{fitness}(\cdot)$, binomial distribution model $\mathcal{B}(\boldsymbol{\theta})$ with parameters $\boldsymbol{\theta} \in \mathbb{R}^d$, learning rate $0 \leq \eta \leq 1$
Outputs: Solution $\mathbf{s} \in \mathbb{R}^d$
Initialize $\boldsymbol{\theta}$ and $\mathbf{s}$ with random values in $[0,1]^d$
repeat until convergence
Sample n candidates from model: $P \sim \mathcal{B}(\boldsymbol{\theta})$
Assess fitness of candidate population: $\mathbf{f} = \mathsf{fitness}(P)$
Choose m candidates: $S = \{\mathbf{x}_i \in P : f_i \in \text{ top-}m\text{-ranked}\}$
Re-estimate model: $\boldsymbol{\theta} = (1-\eta)\boldsymbol{\theta} + \eta \frac{1}{m} \sum_i \mathbf{x}_i, \quad \mathbf{x}_i \in S$
Update solution: $\mathbf{s} = \mathsf{argmax}(\mathsf{fitness}(\mathbf{s}), \mathsf{fitness}(S^{top}))$

In `Goldenberry`, the `PBIL` widget defines a class `Pbil` which overrides the `initialization()` and `estimate()` methods according to the previous algorithm. Thus, `PBIL` is fully implemented as the following class script (notice that the `average` vectorized auxiliary operation from the `NumPy` library (`np`) is used):

```
class Pbil(GbBaseEda):
 def initialize(self):
   self.distr=Binomial(self.var_size)

 def estimate(self, top_ranked, best):
   self.distr.p=self.distr.p*(1-self.learning_rate)
      + self.learning_rate * np.average(top_ranked)
```

The small size of the script is due to the fact that many algorithmic details have been inherited from the parent class, `GbBaseEda`, because they are common to most of other `EDA`s. For example, the parent class defines the initialization of parameters such as `sample_size`, `pop_size` (size of the population), `selection_rate` (percentage of top-ranked selected candidates), and `max_evals` (limit on the number of cost function evaluations allowed on an entire search). Other common features such as the `sample()` method (which delegates this task to the respective probability model), the `get_top_ranked()` method to select the most promising candidates from the sample, and the `search()` method itself, are defined in this abstract class. Its script is partially shown below.

```
class GbBaseEda(GbBaseOptimizer):
...
 def sample(self, sample_size, top_ranked, best):
    return self.distr.sample(sample_size)

 def get_top_ranked(self, candidates):
    fits = self.cost_func(candidates)
    index = np.argsort(fits)
     [:(self.cand_size*self.selection_rate/100):-1]
    return candidates[index],
     bSolution(candidates[index[0]],fits[index[0]])
...
 def search(self):
     if not self.ready():
      raise Exception("Optimizer not ready.")
     best = GbSolution(None, float('-Inf'))
     top_ranked = None
     while not self.done():
         candidates=self.sample(self.sample_size,
                                top_ranked, best)
         top_ranked, winner =
                  self.get_top_ranked(candidates)
         self.estimate(top_ranked, best)
         if best.cost < winner.cost:
             best = winner
         self.iters += 1
     return best

 @abc.abstractmethod
 def estimate(self, candidates, best):
     raise NotImplementedError()
```

It can be seen in the previous code that the implementation of `estimate()`, the estimation of distribution method, is deferred to the specific `EDA` class, as it was the case of the `PBIL` component.

Now let us complete the illustration by mentioning another well-known `EDA`, the *Compact Genetic Algorithm* (`cGA`) [8]. The algorithm is similar in fashion to `PBIL`, the main difference being that it operates in a *compact* mode, that is, instead of estimating the distribution from a batch of many candidates (population), the algorithm works by sampling two candidates at a time and using them to incrementally build the estimation. The pseudo-code of this procedure is thus shown in Algorithm 2, which again is our rendition of the original, written in vectorized operation mode.

Algorithm 2 `cGA`

Requires: Cost function $\mathsf{fitness}(\cdot)$, binomial distribution model $\mathcal{B}(\boldsymbol{\theta})$ with parameters $\boldsymbol{\theta} \in \mathbb{R}^d$
Outputs: Solution $\mathbf{s} \in \mathbb{R}^d$
Initialize $\boldsymbol{\theta}$ and $\mathbf{s}$ with random values in $[0,1]$
repeat until convergence
repeat n **times**
Sample 2 candidates from model: $\{\mathbf{x}_1, \mathbf{x}_2\} \sim \mathcal{B}(\boldsymbol{\theta})$
Rank them: $\{\mathbf{w}, \mathbf{l}\} = \mathsf{compete}(\mathsf{fitness}(\mathbf{x}_1), \mathsf{fitness}(\mathbf{x}_2))$
Re-estimate model with winner and loser: $\boldsymbol{\theta} = \boldsymbol{\theta} + \frac{1}{n}(\mathbf{w} - \mathbf{l})$
Update solution: $\mathbf{s} = \mathsf{argmax}(\mathsf{fitness}(\mathbf{s}), \mathsf{fitness}(\mathbf{w}))$

The cGA, as implemented in full in Goldenberry, is shown in the script below.

```
class Cga(GbBaseEda):
   def initialize(self):
       self.distr=Binomial(self.var_size)
       self.learning_rate=1.0/float(self.pop_size)
       self.sample_size=2

   def estimate(self, (winner, loser), best):
       self.distr.p =
         np.minimum(np.ones((1, self.var_size)),
           np.maximum(np.zeros((1, self.var_size)),
           self.distr.p +
  (winner.params-loser.params)*self.learning_rate))

   def get_top_ranked(self, candidates):
       costs = self.cost_func(candidates)
       maxindx = np.argmax(costs)
       winner = GbSolution(candidates[maxindx],
                           costs[maxindx])
       loser = GbSolution(candidates[not maxindx],
                          costs[not maxindx])
       return  (winner, loser), winner
```

The algorithm is initialized with the following settings: a binomial univariate probability model the size of the number of variables; learning rate of $1/n$ (where n is the population size); and sample size of two, because of its *compact* style. The `estimate()` method is overridden to account for a compact learning rule, using two competitors; here again, the auxiliary `NumPy` library is used (`np`). The `get_top_ranked()` is overridden accordingly, to return the winner and loser of the contest as the top-ranked set.

4.3 Statistics classes

The classes comprising this module were designed to account for the mechanisms needed to modeling probability distributions. A `GbBaseDistribution` is defined encapsulating a `sample()` method, a `has_converged()` validation method, and a `reset()` method for randomly initializing the parameters of the probability model (see Figure 4). Currently four probability distributions are implemented: `Binomial`, `Gaussian`, `TruncatedGaussian` and `BivariateBinomial`. Most of the functionality of these classes were implemented using standard `NumPy` routines such as generation of normally independent distibuted random vectors, and element-wise vector operators (details omitted).

5. COMPONENTS VIEW

Figure 5 shows the components view of the toolbox. For illustration purposes only two `EDA` components are shown: `cGA` for discrete domains and `TILDA` for continuous-valued domains (we recall the other implemented components are: `PBIL`, `UMDA`, $\texttt{PBIL}_c$, $\texttt{UMDA}_c$, `BMDA`).

5.1 The EDA components

Any of these components internally consists of a collaboration between an `EDA` algorithm and a probability model (e.g. `cGA` uses a binomial distribution, `TILDA` uses a Gaussian distribution, and so on). Each `EDA` component exposes two interfaces. The first interface requires a `GbCostFunction` object that is able to evaluate the cost function on a given candidate solution, and also is able to trace statistics of the total number of evaluations made by the `EDA`. The second interface provides a `GbBaseOptimizer` object, initialized and ready to run a stochastic search to optimize the cost function.

These components are at the end of the day deployed as widgets in the `Orange` canvas, under a new toolbar named "Optimization" (see again Figure 1). The user interface of these widgets consists of a setup/results window. In there, parameters settings are applied to the `EDA` component and optimization results are displayed in an output text box after execution of the algorithm. An example of the user interface for the `cGA` widget, applied to solve the classical `OneMax` problem with 100 variables, using a population size of 30, and 1000 maximum number of evaluations, is shown in Figure 6.

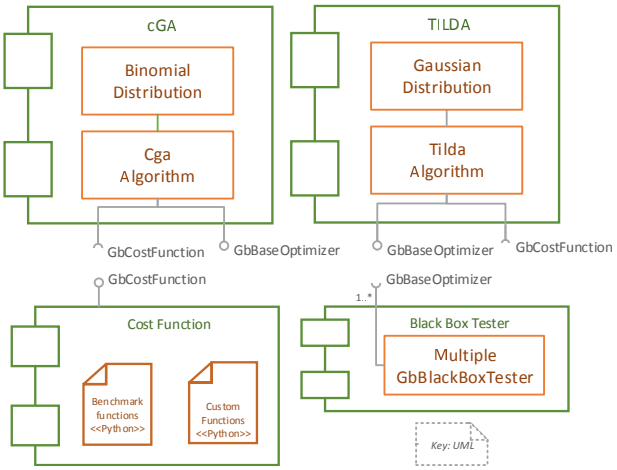


Figure 5: The components diagram of `Goldenberry`.

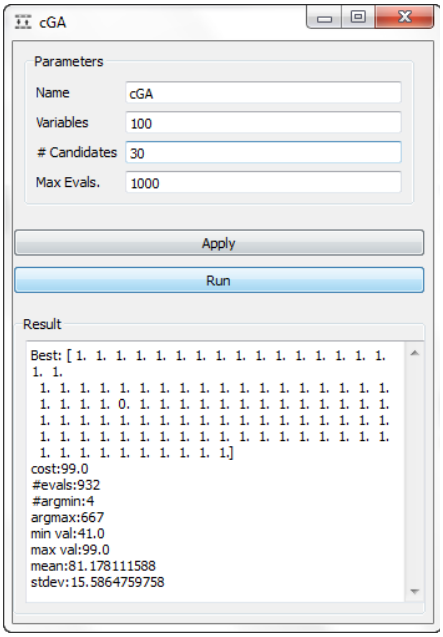


Figure 6: User interface of the `cGA` widget.

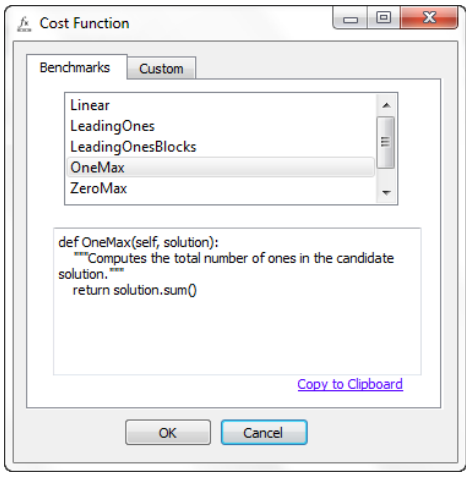
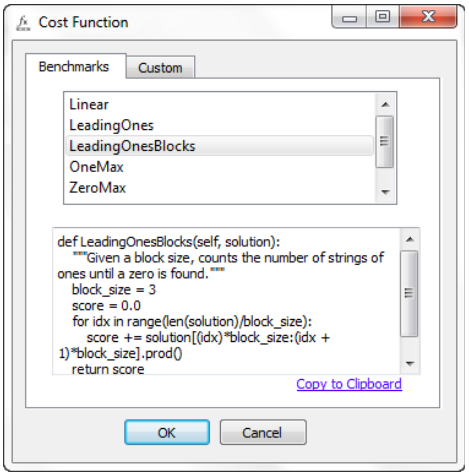
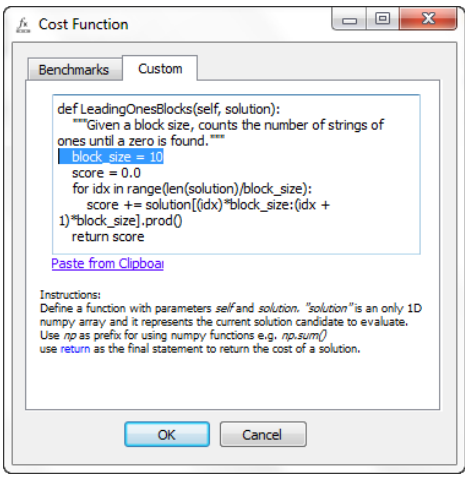


Figure 7: The benchmark input mode of the `CostFunctionBuilder` component. Here the well-known `Onemax` problem is chosen.



(a)



(b)

Figure 8: How to customize a built-in bechmark function: (a) A benchmark function is chosen and its code is copied into the clipboard; (b) the code is pasted in the "Custom" tab and edited as required.

5.2 The Cost Function Builder component

This component enables the user to define the cost function for the optimization problem. It provides two input modes to setup such a function. In the first mode, a set of ready-to-use built-in benchmark functions are listed to the user as it is illustrated in Figure 7. This mode appears on the "Benchmarks" tab of the `CostFunctionBuilder` user interface. The user chooses a function name, and then the Python script implementing the function is shown in the underlying text box; the box is read-only, so the code can not be edited but can be copied into the clipboard. The benchmark functions were taken from those suggested in [11].

The second mode consists of a free-text input box for writing up any customized cost function in the Python language. This mode appears on the "Custom" tab of the `CostFunctionBuilder` user interface. Custom functions must be written complying with the following Python-style signature:

```
def yourcustomfunctionname(self, solution):
...
return computedcostofsolution
```

An arbitrary routine to evaluate the given `solution` (a `NumPy` 1D vector array with the values of that solution to the problem variables) must be defined in order to compute its associated cost. For example, if the user wants to define a customized version of the benchmarks built-int functions, he or she may copy to the clipboard its original code, then paste it in the "Custom" tab and make the necessary adjustments (see Figure 8). Alternatively the user may write up his cost function code from scratch, to meet his particular problem needs. We remark that this a is non-intrusive input mode, meaning that the user-written-code is bind to the `Goldenberry` components in run time; no additional intervention has to be done in the source code of the software. We anticipate this feature would extend the usability of the `EDA` components to a wide range of discrete and continuous optimization problem domains.

5.3 The Black-Box Tester component

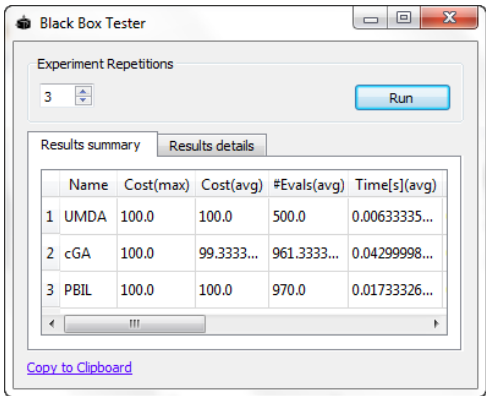
This component was designed to allow the user to run and compare execution of different algorithms or algorithm configurations over the same cost function, in one experiment with several repetitions. The number of repetitions is set as an input parameter for this component. Another interesting functionality of the `BlackBoxTester` is that it is able to collect outputs of all the optimizer components provided as inputs, and display summarized statistics, and also details of the different runs and repetitions. The user interface of the widget associated to this component will be used to display results for the working experiments reported in the next section.

6. GOLDENBERRY AT WORK

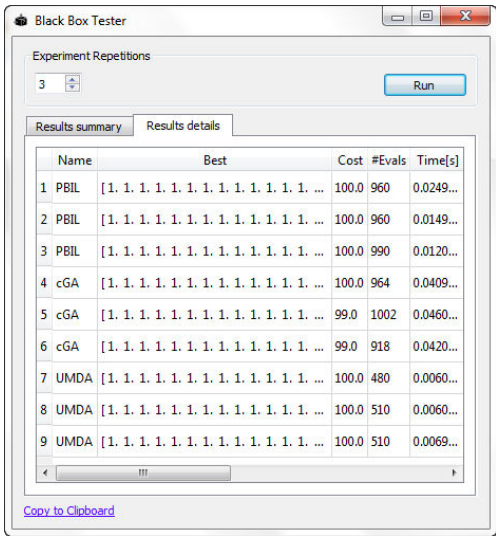
In this section we show how to use the developed components to solve optimization problems defined as minimization of a cost function. We carried out experiments using benchmark and customized cost functions. Other uses in machine learning can be also envisioned (see for example the discussion in Section 7).

The results of these experiments are shown next. It is worth to remark that previous to deployment, additional val-

idation for the algorithmic machinery of each of the `Goldenberry` components was carried-out using a carefully designed set of unit tests (omitted because of space limitations). The program shown in Figure 1 was used in these experiments: A `CostFunctionBuilder` component is instantiated to select the optimization problem, as explained before; three `EDA` components were tested (`cGA`, `PBIL` and `UMDA`) using as input the selected `CostFunction`; finally, a `BlackBoxTester` executes and collects the outputs of the `EDA` components and shows summarised and detailed results over a number of repetitions of the experiment.



(a)



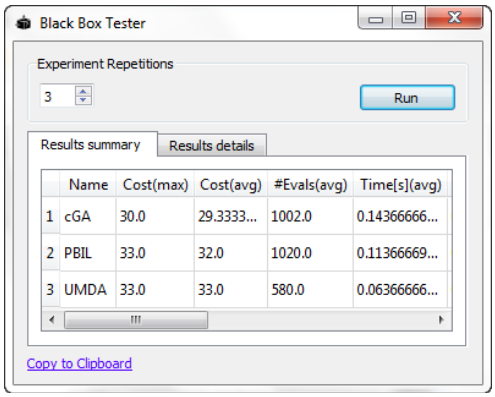
(b)

Figure 9: `Onemax` experiment results: (a) Summary over all repetitions (partial view); (b) Details per repetition (partial view).

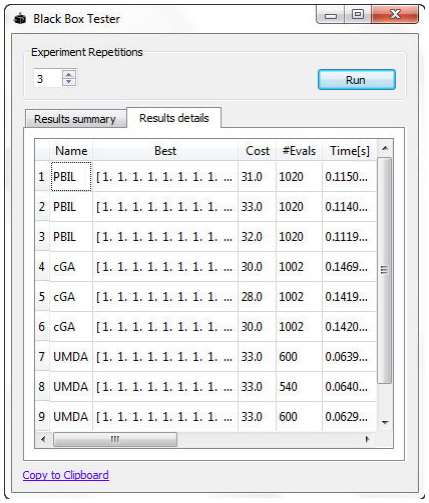
6.1 Benchmark function optimization

Two problems were chosen from the built-in benchmark library of functions: `Onemax` and `LeadingOnesBlock`. Results for the `Onemax` experiment are reported in Figure 9. Figure 9(a) shows the following aggregate results per `EDA` algorithm: maximun cost found in all experiment repetitions; average cost over all repetitions; average number of cost function evaluations per repetition; and average CPU time per repetition (in seconds). Other statistics can be displayed by scrolling the table bar to the right. Notice that the obtained tabulated results can be exported to spreadsheet software tools for further analysis by using the "Copy to Clipboard" option located underneath the output table. Figure 9(b) shows a detailed view of the results obtained in the experiments. Here each row holds the outputs per repetition for each input `EDA` component. The summarized results are statistics of these individual records. The only column not used in the summarized report is the actual vector representation of the best solution found in each run (for `Onemax` the expected solution is an all-ones vector). The cost of this solution is shown in the next column to its right.

For the `LeadingOnesBlock` experiment, results are similarly self-explained, as reported in Figure 10.



(a)



(b)

Figure 10: `LeadingOnesBlock` experiment results: (a) Summary (partial view); (a) Details (partial view).

6.2 Customized function optimization

In this experiment we used the same `Goldenberry` program of Figure 1 to carry out optimization of the customized cost function defined in Figure 8, that is, the same `LeadingOnesBlock` problem this time with a `block_size` value of 10. Similarly to the other experiments, results are shown in Figure 11.

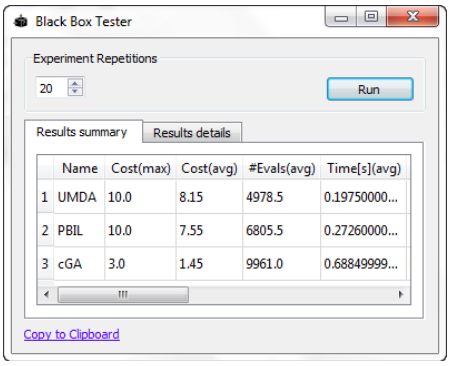


Figure 11: Results of custom function experiment.

7. CONCLUSIONS

It is our belief that user-friendly, open-source visual tools may have a big potential benefit in tasks carried out daily by data mining analysts. The `Orange` platform is a fantastic effort complying with these premises by combining a powerful visual programming approach with the reuse and glue principles of component-based software. The `Goldenberry` initiative is a modest contribution intended to extend the application domain of `Orange` to the field of black-box and metaheuristics optimization. In this first release we developed a number of `EDA`-based software components featuring a non-intrusive, runtime-binding interface to allow users to define tailor-made discrete and continuous optimization problems.

As it was mentioned in the introduction, it is anticipated that these components can be used independently as function optimizers, as it is reported in this paper, or as part of higher-level data mining machines. Let us illustrate the point with the task of feature selection. The aim there is to select an optimal subset of relevant variables for a classification problem; the relevant found subset can be further analysed by experts for specific purposes (in biology for example, features may represent over-expressed gene activity due to an illness condition). In the so-called *wrapper* scheme of the problem [18] the classifier is enabled to incorporate the selection mechanism during the learning stage: a weighted kernel classifier, for example, may use an `UMDA` component to define relevance coefficients for the variables and then use them as input for the kernel machine (that is the approach taken in [16]). Currently these type of mechanisms are hidden in the current implementation of classifier or clustering components of `Orange`. Thus, by providing interfaces to black-box optimizers we hope to extend applicability to this type of tasks, giving researchers a flexible workbench to design new component-based machine learning techniques. Novel components complying with the design principles described in this paper will be needed though (e.g. component-based genetic algorithms, kernel machines, etc.), and that would be matter of the next `Goldenberry` release.

8. ACKNOWLEDGEMENTS

We would like to acknowledge Henry Diosa, Leidy Garzón and Harry Sanchez, members of the Arquisoft Research Group from the District University of Bogota, for the design and implementation of the `BlackBoxTester` component. We also thank the anonymous reviewers for their valuable comments which allowed us to greatly improve readability of the paper.

9. REFERENCES

[1] F. Bachmann, L. Bass, and R. Nord. Modifiability tactics. Technical Report CMU/SEI-2007-TR-002, Software Engineering Institute, Carnegie Mellon University, 2007.

[2] S. Baluja and R. Caruana. Removing the genetics from the standard genetic algorithm. Technical Report CMU-CS-95-141, Carnegie-Mellon University, 1995.

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. Wiley, volume 1 edition, Aug. 1996.

[4] T. Curk, J. Demsar, Q. Xu, G. Leban, U. Petrovic, I. Bratko, G. Shaulsky, and B. Zupan. Microarray data mining with visual programming. *Bioinformatics*, 21(3):396–398, 2005.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, Nov. 1994.

[6] C. Gonzáles, J. A. Lozano, and P. Larrañaga. Mathematical modelling of UMDA$_c$ algorithm with tournament selection. *International Journal of Approximate Reasoning*, 31(3):313–340, 2002.

[7] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. Witten. The WEKA data mining software. *SIGKDD Explor. Newsl.*, 11(1):10–18, 2009.

[8] G. R. Harik and F. G. Lobo. The compact genetic algorithm. *IEEE Transactions on Evolutionary Computation*, 3:523–528, 1999.

[9] IBM. *IBM SPSS® Algorithms Guide*. 2012.

[10] P. Larrañaga and J. A. Lozano, editors. *Estimation of Distribution Algorithms: A New Tool for Evolutionary Computation*. Springer, 2001.

[11] S. Luke. *Essentials of Metaheuristics*. Lulu, 2009. Available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/.

[12] I. Mierswa, M. Wurst, R. Klinkenberg, M. Scholz, and T. Euler. Yale: Rapid prototyping for complex data mining tasks. In L. Ungar, M. Craven, D. Gunopulos, and T. Eliassi-Rad, editors, *Proceedings of the 12th ACM SIGKDD*, pages 935–940, NY, USA, August 2006. ACM.

[13] H. Mühlenbein. The equation for response to selection and its use for prediction. *Evolutionary Computation*, 5(3):303–346, 1997.

[14] M. Pelikan and H. Müehlenbein. The bivariate marginal distribution algorithm. In R. Roy, T. Furuhashi, and P. Chawdhry, editors, *Advances in Soft Computing*. Springer London, 1999.

[15] M. Pelikan, K. Sastry, and E. Cantú-Paz, editors. *Scalable Optimization via Probabilistic Modeling: From Algorithms to Applications*. Springer-Verlag, NJ, USA, 2006.

[16] S. Rojas-Galeano, E. Hsieh, D. Agranoff, S. Krishna, and D. Fernandez-Reyes. Estimation of relevant variables on high-dimensional biological patterns using iterated weighted kernel functions. *PLoS ONE*, 3(3), 2008.

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

[18] Y. Saeys, I. n. Inza, and P. Larrañaga. A review of feature selection techniques in bioinformatics. *Bioinformatics*, 23(19):2507–2517, Sept. 2007.

[19] M. Sebag and A. Ducoulombier. Extending Population-Based incremental learning to continuous search spaces. *Lecture Notes in Computer Science*, 1498, 1998.

Appendix. Download and installation

`Goldenberry` is hosted publicly under a GPL license in Codeplex (`http://goldenberry.codeplex.com`). To install the software follow these steps:

1. Download and install Orange 2.6.1 from: `http://orange.biolab.si/download/`.
2. Get the latest `Goldenberry` release from the `Downloads` tab in `http://goldenberry.codeplex.com` (the download link is located to the left side of the screen).
3. Follow the installation instructions in the release notes just below the download link.
4. Open the `Orange` application and look for the "Optimization" toolbar in the canvas (as shown in Figure 1).