International Journal on Artificial Intelligence Tools

ADAPTIVE LEARNING SEARCH, A NEW TOOL TO HELP COMPREHENDING METAHEURISTICS

JOHANN DRÉO*
JEAN-PHILIPPE AUMASSON†
WALID TFAILI*
PATRICK SIARRY*

* *Université Paris XII Val-de-Marne*
Laboratoire Images, Signaux et Systèmes Intelligents (LISSI, EA 3956)
61, avenue du Général de Gaulle, 94010 Créteil, France
{*dreo, siarry, tfaili*}*@univ-paris12.fr*
† *Security and Cryptography Laboratory, EPFL, Switzerland*

The majority of the algorithms used to solve hard optimization problems today are population metaheuristics. These methods are often presented under a purely algorithmic angle, while insisting on the metaphors which led to their design. We propose in this article to regard population metaheuristics as methods making evolution a probabilistic sampling of the objective function, either explicitly, implicitly, or directly, via processes of learning, diversification, and intensification. We present a synthesis of some metaheuristics and their functioning seen under this angle, called Adaptive Learning Search. We discuss how to design metaheuristics following this approach, and propose an implementation with our Open Metaheuristics framework, along with concrete examples.

Keywords: metaheuristics; hard optimization; evolutionary algorithm; ant colony algorithm; simulated annealing; estimation of distribution; optimization framework; software; performance assessment.

1. Introduction

Optimization problems appear in many fields, as various as identification problems, supervised learning of neural networks, shortest path problems, *etc.* Metaheuristics[18] are a family of stochastic optimization algorithms, often applied to hard combinatorial problems for which no more efficient method is known. They have the advantage of being generic methods, thus do not require a complex tuning for each problem, and can be used as a kind of "black boxes". Recall that, generally, optimization algorithms search for a *point* into the *search space*, so as to optimize (*i.e.*, minimize or maximize) the *objective function* (also called fitness or goal function). Metaheuristics are often divided into two sets:

(1) Algorithms handling a *single point*, making it evolve towards a solution.
(2) Algorithms handling a *population*, *i.e.*, a finite set of points, and computing a new population at each iteration.

An essential observation is that the population of the second category is a *sampling* of the objective function. Although those classes are not disjoint (an algorithm can belong to both classes, according to the point of view), we only consider population metaheuristics, which are simply referred as *metaheuristics* hereafter.

In order to analyze metaheuristics, several formalizations were proposed[40,41,10], following the adaptive memory programming[52] concept, which aims at unifying the population metaheuristics by stressing a common process. Following our previous observation, we present a new model which considers the population as a probabilistic sampling, and illustrate our approach with a software specially developed, Open Metaheuristics[15]. It provides a common framework to all metaheuristics, and so facilitates their implementation and comparison. This contrasts with general frameworks[55] such as HotFrame[21], Templar[33] or HSF [14] where metaheuristics are simply algorithms without a specific structure, and with specific frameworks, such as Evolving-Objects[34], where only one specific metaheuristic structure is considered.

We introduce the preliminary material in Section 2, then we present our model Adaptive Learning Search in Section 3. Section 4 focuses on the implementation, presenting the software Open Metaheuristics along with several examples. Section 5 is the conclusion.

2. Fundamental concepts

Metaheuristics share a certain number of properties. An essential one is that they handle a sampling of the objective function, via common processes. The framework of adaptive memory programming aims at stressing those processes.

The probabilistic sampling should ideally pick the best solutions with higher probability. However, in an optimization problem, the effective goal is not to sample the objective function, but to find the distribution's optimum. Thus, sampling must concentrate on the areas of interest, while converging gradually towards the optimum by means of algorithms. From the point of view of sampling, this convergence is carried out by a progressive fall of dispersion in these areas.

2.1. *Adaptive memory programming*

Adaptive Memory Programming (AMP) is a common framework to metaheuristics[52], described in Algorithm 1. It stresses out the concepts of *memory*, *intensification*, and *diversification*. In the literature of evolutionary algorithms, these two last notions are often replaced by the words *exploitation* and *exploration*, which have a similar meaning.

We briefly detail each element of the AMP framework:

- *Memory* stands for the information collected by the algorithm on the objective function distribution. It can be represented either as a simple set of points, or as more complex structures, like pheromone tracks in ant colony algorithms. Memory can be defined as global (compared to the problem as a whole) or

Algorithm 1 AMP framework.

(1) Memorization of a set of solutions, or of a data structure containing the characteristics of the solutions produced by the search.
(2) Construction of a temporary solution based on the data memorized.
(3) Improvement of the solution by an algorithm of local search.
(4) Memorization of the new solution, or of the data structure associated.

inter-individual (a solution relative to another one).

- *Intensification* exploits the information obtained, in order to improve the current solutions. This is typically a local search algorithm (for instance with the Nelder-Mead algorithm[43] or a taboo search).
- *Diversification* aims at collecting new information, by exploring the search space.

The three components presented are not always clearly distinct, and are strongly interdependent in an algorithm. An example of metaheuristic that fits well the AMP model is the method GRASP[47].

2.2. *Sampling and AMP*

In the majority of metaheuristics, the sampling of the objective function is probabilistic (diversification). Ideally, this sampling should be performed with respect to an approximation of the distribution of the points, so as to locate an area of interest, and then converge towards the optimum (intensification).

Most of the metaheuristics do not have any *a priori* information on the distribution, thus *implicitely* learn it by diversification and intensification, such as ant colony algorithms, and "classical" metaheuristics. Conversely, some methods use an approximation of the distribution, and are called *explicit* methods (see[3]). For example, Estimation of Distribution Algorithms[37] (EDA) are explicit methods. We also distinguish the *direct* methods, using directly the objective function, like the simulated annealing.

2.3. *General scopes*

We assisted to several attempts of structuration in the scope of distribution sampling. For instance, Monmarché *et al.* proposed the model Probabilistic Search Metaheuristic [40,41] (PSM), based on the comparison of the algorithms PBIL[2,4], BSC [50], and the ant system algorithm[10]. The general principle of a PSM method is presented in Algorithm 2. Notice the relation of this approach with the estimation of distribution algorithms. However, the PSM approach is limited to the use of probability vectors, while specifying an essential update rule for these vectors.

The EDA's were presented as evolutionary algorithms, with an explicit diversification[42]. They are undoubtedly the algorithms closest to a general scope.

Algorithm 2 The scope of the *PSM* method.

Initialize a probability vector $p_0(x)$
Until stopping criteria:
  Build m individuals $x_1^l, \ldots, x_m^l$ using $p_l(x)$
  Evaluate $f\left(x_1^l\right), \ldots, f\left(x_m^l\right)$
  Rebuild a probability vector $p_{l+1}(x)$ while considering $x_1^l, \ldots, x_m^l$ and $f\left(x_1^l\right), \ldots, f\left(x_m^l\right)$
End

The Iterated Density Evolutionary Algorithms[6,7,8] (IDEA) are a generalization of those, presented in Algorithm 3.

Algorithm 3 The *IDEA* approach.

Initialize a population P_0 of n points
Until stopping criteria:
  Memorize the worst point θ
  Search an appropriate distribution $D_i(X)$ from the population P_{i-1}
  Build a population O_i of m points according to $D_i(X)$, with $\forall O_i^j \in O_i$: $f\left(O_i^j\right) < f(\theta)$
  Create a population P_i from a part of P_{i-1} and a part of O_i
  Evaluate P_i
End

IDEA uses a more general diversification than PSM, while not being limited to a probability vector as model, but specifying that the search for the best probability distribution forms an integral part of the algorithm. However, the fall of dispersion is carried out by selecting the best individuals, no precision on the use of different intensification principles is given.

3. Adaptive learning search

In this section we present Adaptive Learning Search (ALS), a new framework for considering the structure of metaheuristics, based on the AMP approach.

3.1. *Main processes*

Instead of considering only a memorization process, we propose to consider a *learning* phase. Indeed, the memory concept is quite static and passive; in a sampling approach, it suggests that the sample is simply stored, and that the metaheuristic only takes into account the previous iteration, without considering the whole opti-

mization process. We emphasize on the fact that the memorized data is not only a raw input, but provides *information* on the distribution, and thus on the solutions.

Thereby, we propose to consider three terms to describe the main steps in a population metaheuristic: learning, diversification and intensification, with respect to a sampling either explicit, implicit, or direct. An ALS algorithm is thus organized as presented in Algorithm 4.

Algorithm 4 ALS algorithm.

Initialize a sample;
Iterate until stopping criteria:

Sampling: either explicit, implicit or direct,
Learning: the algorithm extracts information from the sample,
Diversification: it searches for new solutions,
Intensification: it searches to improve the existing sample,
Replace the previous sample with the new one.

End

3.2. *Examples*

We present several famous metaheuristics under the scope of ALS.

3.2.1. *Simulated annealing*

The simulated annealing[36,9] was created from the analogy between a physical process (the annealing) and an optimization problem. As a metaheuristic, it is based on works simulating the evolution of a solid towards its minimal energetic state[39,28].

The classic description of simulated annealing presents it as a probabilistic algorithm, where a point evolves in the search space. The method uses the Metropolis algorithm, recalled in Algorithm 5, inducing a markovian process[1,35]. The simulated annealing, in its usual version ("homogeneous"), calls this method at each iteration.

It is possible to see the simulated annealing as a population algorithm. Indeed, the Metropolis algorithm directly samples the objective function using a degenerated parametric Boltzmann distribution (of parameter T). Hence, one of the essential parameters is the temperature decrease, for which many laws were proposed[53]. There also exists some versions of the simulated annealing more centred on the handling of a points population [30,56,38,31].

Here, the Metropolis method represents the diversification (coupled with the learning), while the temperature decrease is controlling the intensification process. Note that other methods than Metropolis' may be used[11,45].

Algorithm 6 presents a synthesis of the simulated annealing. The learning step is not present in basic versions, but many existing variants have tried to link the tem-

Algorithm 5 Sampling with the Metropolis method.

Initialize a starting point x_0 and a *temperature* T
For $i = 1$ to n:
  Until x_i accepted
    If $f(x_i) \leq f(x_{i-1})$: accept x_i
    If $f(x_i) > f(x_{i-1})$: accept x_i with a probability $e^{-\frac{f(x_i)-f(x_{i-1})}{T}}$
  End
End

perature to certain characteristics of the sampling obtained through the Metropolis method[20,44,12]. Finally, the simulated annealing is mainly characterized by its direct sampling of the objective function.

Algorithm 6 ALS model for the simulated annealing.

Sampling: direct.
Learning: relational mechanisms between the set of points and the temperature.
Diversification: sampling of the objective function through the Metropolis method.
Intensification: temperature decrease.

3.2.2. *Evolutionary algorithms*

Evolutionary algorithms[19] are inspired from the biological process of the adaptation of alive beings to their environment. The analogy between an optimization problem and this biological phenomenon has been formalized by several approaches[29,22,46], leading for example to the famous family of genetic algorithms[25]. The term *population* metaheuristics fits particularly well; following the metaphor, the successive populations are called *generations*. A new generation is computed in three stages, detailed below.

(1) Selection: improves the reproduction ability of the best adapted individuals.
(2) Crossover: produces one or two new individuals from their two parents, while recombining their characteristics.
(3) Mutation: randomly modifies the characteristics of an individual.

One clearly identifies the third step with the diversification stage, while the first one stands for the intensification. We interpret the crossover as a learning from the previous information (*i.e.* from the ancesters). Several methods[50,27,26,5] were designed for the diversification operators, which emphasize the implicit process of distribution sampling.

The ALS modelling of this generic scheme is presented in Algorithm 7.

Algorithm 7 ALS model for evolutionary algorithms.

Sampling: implicit.
Learning: crossover.
Diversification: mutation.
Intensification: selection.

3.2.3. *Estimation of distribution algorithms*

Estimation of Distribution Algorithms (EDA) were first created as an alternative to evolutionary algorithms[42]: the main difference is that crossover and mutation steps are replaced by the choice of random individuals with respect to an estimated distribution obtained from the previous populations. The general process is presented in Algorithm 8.

Algorithm 8 Estimation of distribution algorithm.

$D_0 \leftarrow$ Randomly generate M individuals.
$i = 0$
While stopping criteria:
  $i = i + 1$
  $D_{i-1}^{S_e} \leftarrow$ Select $N \leq M$ individuals in D_{i-1} using the selection method.
  $p_i(x) = p\left(x \mid D_{i-1}^{S_e}\right) \leftarrow$ Estimate the probability distribution of the selected individuals.
  $D_i \leftarrow$ Sample M individuals from $p_i(x)$
End

The main difficulty is how to estimate the distribution; the algorithms used for this are based on an evaluation of the dependency of the variables, and can belong to three different categories:

(1) Models without any dependency: the probability distribution is factorized from univariant independent distributions, over each dimension. That choice has the defect not to be realistic in case of hard optimization, where a dependency between variables is often the rule.
(2) Models with *bivariant* dependency: the probability distribution is factorized from bivariant distributions. In this case, the learning of distribution can be extended to the notion of *structure*.
(3) Models with *multiple* dependencies: the factorization of the probability distribution is obtained from statistics with an order *higher* than two.

For continuous problems, the distribution model is often based on a normal distribution.

Some important variants were proposed, using for example "data clustering" for multimodal optimization, parallel variants for discrete problems (see[37]). Convergence theorems were also formulated, in particular with modeling by Markov chains, or dynamic systems.

We model the EDA algorithms in the ALS scope in Algorithm 9.

Algorithm 9 ALS model for estimation of distribution algorithms.
Using an *explicit* sampling.
Learning: extraction of the parameters of an explicit distribution;
Diversification: sampling of the distribution;
Intensification: selection.

3.3. *Implementing metaheuristics*

The ALS concept can be used as a tool for implementing several metaheuristics in a common framework. This approach permits to reuse common code and facilitates implementation, by focusing on the original parts of an algorithm. Thus we have chosen to separate the implementation of the ALS concept from the implementation of the metaheuristics theirselves.

In order to maximize the factorization of the code, an object oriented language is suitable. Some modelling languages, such as the UML language[49], also help to describe the sketch concepts handled. Additionally, *design patterns* provide generic solutions for common problems in software design, they became popular in the 90's[24], and are now commonly used for most of projects implemented with oriented object languages, like Java or C++.

In order to implement the ALS approach, we can use a common pattern, namely the Template Method pattern. This method is a quite simple behavioral pattern, it only consists in defining the skeleton of an algorithm, in terms of sketch methods for common operations, while allowing subclasses to define particular steps of the algorithm. It can be used for the metaheuristics classes (see Figure 1), which need to share basic attributes or methods (sample, output, *etc.*) but will implement differently common methods (mainly diversification, intensification and learning) that will be called by the main procedure. Additionally, this pattern permits to define methods for some steps of the algorithm that are specific to a given metaheuristic. One of the advantages of this pattern is to force the use of a common interface for all metaheuristics, thus facilitating their manipulation[23].

Implementing metaheuristics through an ALS approach can facilitate comprehension and comparison of several algorithms. Indeed, as they are structured over the three main steps and a sample, one can compare the evolution of this sample

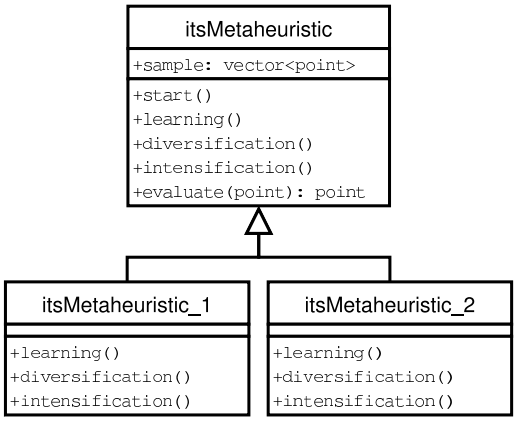


Fig. 1. UML diagram of the template pattern, used to implement a metaheuristic as an ALS.

through these steps, and not only through iterations.

One of the advantages of implementing different metaheuristics in a common framework is to reduce the implementation differences, thus facilitating the comparison of two algorithms.

4. Implementation in the Open Metaheuristics framework

We implemented the ALS approach in a framework called Open Metaheuristics[15] (oMetah), a project under the LGPL licence.

4.1. *Algorithms, problems and communication layers*

In order to separate the implementation of metaheuristics from the implementation of problems, we have designed the project around three components:

- metaheuristics,
- problems,
- communication layers.

As shown on Figure 2, metaheuristics and problems communicate through a client/server abstraction module. This permits to easily interface implemented problems and algorithms, but also external problems with existing metaheuristics. Indeed, such an approach can help interfacing oMetah objects with another software. For example, if a problem is only available as a command line binary software, one can use a communication protocol using a temporary *file* where results are written, a network protocol can also be used. Such an approach is used in several general frameworks, but in a static link way, whereas we prefer a client/server way, more generalized and flexible.

In order to handle the different object types, we use an abstraction of a set of objects (see Figure 3). Thanks to this class and to the abstract factory pattern,

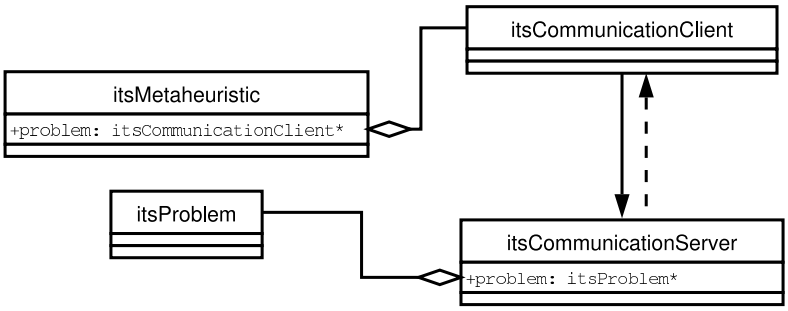


Fig. 2. UML diagram of relationships between metaheuristics and problems in oMetah.

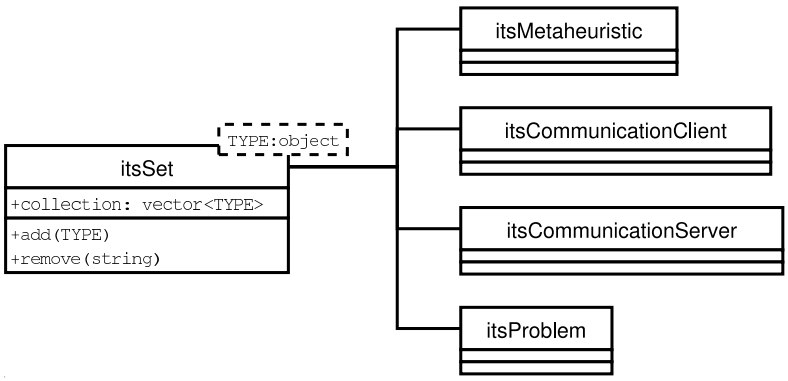


Fig. 3. UML diagram of the set class.

it is possible to manipulate several objects sharing a common interface. This class can be used to connect several metaheuristics with several problems through several communication protocols.

4.2. *The metaheuristic base class*

Each new metaheuristic must inherit from a base class, which contains tools to build it and manages its interface. The most important method in this base class is the `start` function, depicted in Algorithm 10, which is used to start the optimization process. Basically, it iterates through the three steps, and output the sample, until a stopping criterion is reached. Virtual methods are provided for additional processes at the beginning or at the end of the optimization.

4.3. *Examples of implementations*

Implementing a search algorithm in oMetah is rather simple: defining the three steps (learning, diversification and intensification) is sufficient. Currently, several

Algorithm 10 Implementation of the start method for the metaheuristic base class.

```
void itsMetaheuristic::start() {
  // an initialization step before performing the optimization
  initialization();
  // while no stopping criterion reached
  while( !isStoppingCriteria() ) {
    learning();  // Learning phase
    outputSample();
    diversification();  // Diversification phase
    outputSample();
    intensification();  // Intensification phase
    outputSample();
   // one more iteration
    iterationsCurrent++;
  }
  // an ending step, if necessary
  end();
}
```

algorithms have been implemented:

- random and exhaustive search,
- Nelder-Mead search,
- genetic algorithms,
- ant colony algorithms,
- estimation of distribution algorithms,
- simulated annealing.

We present three examples of implementations in the following sections.

4.3.1. *Random search*

As a very simple illustration of implementation, we give in Algorithm 11 the implementation of a pure random search, using an uniform distribution. This example demonstrates the definition of the three steps, the class used for points (solution vector and solution value) and some of the common methods, such as the call for the objective function.

4.3.2. *Estimation of distribution*

As a more complex illustration, Algorithm 12 implements a simple EDA, using a multi-normal probability density function (PDF) as a learning basis. We comment the three steps below.

(1) The aim of the learning step is to extract the parameters (namely the mean vector and the variance-covariance matrix) of a multi-normal PDF from the sample.

Algorithm 11 Implementation of a simple random search with oMetah.

```
void itsRandom::learning() { // No learning. }

void itsRandom::diversification() {
// draw each point in an hyper cube
 for( unsigned int i=0; i < getSampleSize(); i++) {
  itsPoint p;
  p.setSolution(
   randomUniform(
    getProblem()->boundsMinima(),
    getProblem()->boundsMaxima()
   )
  );
  setSample[i]( evaluate(p) ); // call for the problem
 }
}

void itsRandom::intensification() { // No intensification. }
```

(2) At the diversification step, one draws a new sample according to the parameters.
(3) The intensification step selects the best points.

4.3.3. *Interface to other frameworks*

The high-level structure of Open Metaheuristics permits algorithms written with other frameworks to be embedded. As an example, we provide an interface to the Evolving-Objects (EO) framework[34]. This framework is dedicated to evolutionary computing, and uses a component-based approach. Algorithm 13 shows the skeleton of the interface, implemented as a wrapper, to the canonic genetic algorithm (referred as SGA in EO). The attributes types starting with `eo` are specific to the EO framework. The two methods `sampleToPop` and `popToSample` are used to convert the oMetah sample to the EO "population", and conversely.

4.4. *Manipulating objects*

The objects (metaheuristics, problems and communication layers) in oMetah can be easily manipulated through the abstract factory pattern. Abstract factory belongs to the class of creation design patterns, which deal with object creation in an object oriented language. It provides an easy way to create objects sharing common properties – in fact they will have the same parent class – but one has not to explicitly specify which classes. In a simple way, one will define one abstract factory, as an instance of an abstract class, that will be used to create instances of derived classes, via their own factories[23].

For example, an abstract factory for the problems would allow to create instances of specific metaheuristics, so as to, for example, add them to a specific set, by using

Algorithm 12 Implementation of an estimation of distribution algorithm.

```
void itsEDA::learning(){
  // mean vector
  this->parameterNormalMean = mean( this->getSample() );
  // variance covariance matrix
  this->parameterNormalVarCovar = varianceCovariance( this->getSample() );
}
void itsEDA::diversification() {
  // draw each point in a multi-normal PDF
  for( unsigned int i=0; i < getSampleSize(); i++) {
    itsPoint p;
   // draw according to parameters
    vector<double> sol = randomNormalMulti(this->parameterNormalMean,
this->parameterNormalVarCovar);
   // uniform PDF used if out of bounds
    for(unsigned int i=0; i < this->getProblem()->getDimension(); i++) {
      if( sol[i] < this->getProblem()->boundsMinima()[i] ||
          sol[i] > this->getProblem()->boundsMaxima()[i] ) {
        sol = randomUniform( this->getProblem()->boundsMinima(),
this->getProblem()->boundsMaxima() );
      }
    }
    p.setSolution(sol);
    sample[i] = evaluate(p); // call for the problem
  }
}
void itsEDA::intensification(){
  // Select the best points
  setSample( selectOnValues( getSample(), selectNumber ) );
}
```

the same interface for all. One only has to define a subclass of the metaheuristic factory for each metaheuristic that we program (see Figure 4). As a result, one has not to specify the type of objects wanted, it even remains unknown, and the programmer can easily change it by modifying the factory creating method.

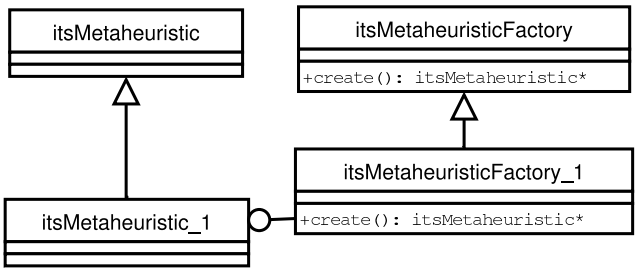


Fig. 4. UML diagram of the abstract factory pattern, used to manipulate metaheuristics implementations.

Algorithm 13 Skeleton of the header of an interface to the EO framework.

```
// oMetah uses real parameters
typedef eoReal<eoMinimizingFitness> EOType;
class itsEOInterface : public itsMetaheuristic {
// EO data structures
eoPop<EOType> * eoOffspring;
eoPop<EOType> * eoPopulation;
// EO operators and their parameters
eoSelectOne<EOType> * eoSelect;  // selection
eoQuadOp<EOType> * eoCross; // crossover
float eoCrossRate;
eoMonOp<EOType> * eoMutate;  // mutation
float eoMutateRate;
void learning() {
  sampleToPop();
  /* use the crossover operator for a finite set of point pairs... */
  popToSample();
}
void diversification() {
  sampleToPop();
  /* use the mutation operator on each point... */
  popToSample();
}
void intensification() {
  sampleToPop();
  /* use the replacement operator... */
  /* use the selection operator... */
  popToSample();
  evaluate();
}
// The constructor takes instances of EO operators as arguments
itsEOInterface( eoPop< EOType > * _pop, eoSelectOne<EOType> * _select,
eoQuadOp<EOType> * _cross, float _cross_rate, eoMonOp<EOType> * _mutate,
float _mutate_rate )
: itsMetaheuristic(), eoPopulation( _pop ), eoSelect( _select ), eoCross(
_cross ), eoCrossRate( _cross_rate), eoMutate( _mutate ), eoMutateRate(
_mutate_rate )
{ }
// Convert the eoPop of EOType individuals to the oMetah sample
void popToSample()  { /* ... */ }
// Convert the oMetah sample to the eoPop of EOType
void sampleToPop()  { /* ... */ }
};
```

4.5. *Using the library*

4.5.1. *Results output*

In order to facilitate the study of metaheuristics behavior, one needs to know as much information as possible. An access to the state of a metaheuristic at each

iteration is thus crucial. In the ALS approach, this state is illustrated by the sample, modified at each step of an iteration.

To achieve this goal, a structured output format is useful. We proposed an XML (*eXtensible Markup Language*) file format, gathering all information of an optimization session (see Figure 5). The advantages of such an approach is that one can easily extract information from the files. As XML is extensible, one can also add a new representation of information. For example, one can add a specific structure to represent a solution (like a tree or a more complex representation) or add the values of the metaheuristics parameters at each iteration (when the algorithm is dynamic, for example). In a XML document, this unpredicted addition will not disturb the possibility of extracting other information.

```
<optimization>
 <iteration id="0">
  <step class="start">
   <sample>
    <point><values>0.705915</values><solution>0.840188</solution></point>
    <point><values>0.155538</values><solution>0.394383</solution></point>
   </sample>
  </step>
  <evaluations>4</evaluations>
 </iteration>
 <iteration id="1">
  <step class="learning">
   <sample>
    <point><values>0.705915</values><solution>0.840188</solution></point>
    <point><values>0.155538</values><solution>0.394383</solution></point>
   </sample>
  </step>
  <step class="diversification">
   <sample>
    <point><values>0.0771588</values><solution>0.277775</solution></point>
   </sample>
  </step>
  <step class="intensification">
   <sample>
    <point><values>0.0348113</values><solution>0.186578</solution></point>
   </sample>
  </step>
  <evaluations>7</evaluations>
 </iteration>
 <optimum>
  <point><values>0.0348113</values><solution>0.186578</solution></point>
 </optimum>
</optimization>
```

Fig. 5. Simple example of the oMetah output format for the optimization results. The algorithm used is a simple EDA, using a sample of 2 points and optimizing the well known Sphere continuous problem, in one dimension.

Open Metaheuristics also proposes a set of tags to include information about the

algorithm and the problem used to generate the results. This information, included in the header of the XML file, are especially useful for verifying and reproducing experiments.

4.5.2. *Tests automation*

We worked out a set of tools (called oMetah Lab) for the tests automation. It permits to run metaheuristics, to manage the data storage and to synthetize results.

This test suite is written in the Python programming language, and consists of two components: the `test` and the `stats` modules. The `test` module aims at running a given metaheuristic on a given problem, several times. The role of the `stats` module is to read the XML data, outputs graphics and tables, and to synthetize the results in a report. An example of automated script for the comparison of two metaheuristics is given in Algorithm 14. The `stats` module permits to choose between a set of plugins, each one handling a different output.

Algorithm 14 Example of a test session for oMetah Lab. The script runs 10 times an EDA and a random algorithm, on the Rosenbrock problem with 2 variables, and outputs all the available graphics in a HTML report.

```
import ometahtest
import ometahstats
path = './ometah'
# Number of runs to calculate
runs = 10
u = ometahtest.Test(path,'-s 10 -i 10 -e 100 -p Rosenbrock -d 2 -m
CEDA',runs)
u.start()
v = ometahtest.Test(path,'-s 10 -i 10 -e 100 -p Rosenbrock -d 2 -m
RA',runs)
v.start()
# calculate all graphics and generate a HTML report
ometahstats.process(paths, 'all','html')
```

4.5.3. *Graphical output*

Knowing the state of the sample permits to calculate the distribution of the points at each iteration or step, as shown on Figure 6, or the distributions of optimums, as shown in Figure 7. With the knowledge of these distributions, one can also rigorously compare them through a non-parametric statistical significance test, like a Mann-Whitney-Wilcoxon one, as recommended by Taillard[51]. One can also plot the optimums found in the solution space, as shown on Figure 8.

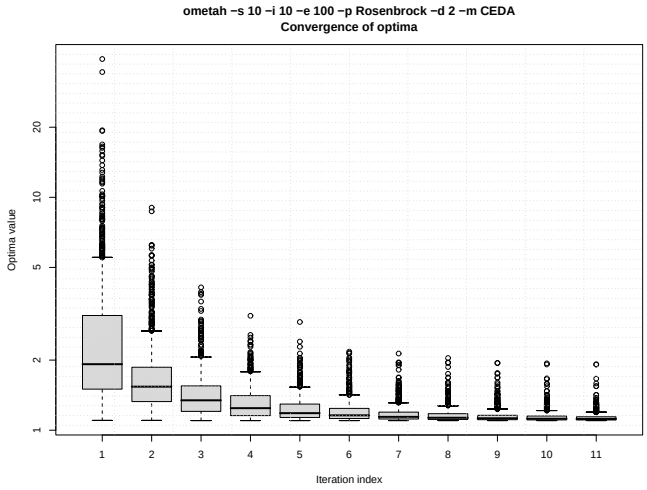


Fig. 6. Distribution of the sample on the values space for an Estimation of Distribution Algorithm[6] optimizing the Rosenbrock objective function[13] with two dimensions. The distributions are obtained for each iteration, at the intensification step, through 50 runs, with a sample of 10 points, limited by 100 evaluations of the problem. Box plots are showing the median, quartiles, minimum and maximum, the whiskers size is limited at 1.5 times the inter-quartiles distance.

4.5.4. *Application*

Open Metaheuristics has been successfully used to solve an optimization problem in medical imaging[17,16]: we employed optimization techniques for registration of retinal angiograms (registration is an important tool for solving many medical image analysis problems). Many common minimization strategies have been applied to image registration problems[48,32], but metaheuristics have shown interesting results, especially for high resolution difficult registration problems.

The key feature, in this application, was the ability to implement the problem in a very simple manner, using the template pattern. Indeed, the separation between metaheuristics, communication layer and problems permits to focus on the implementation of a single class, all the other processes being managed by the framework.

The implementation of the imaging part was tackled with the CImg[54] template library. Algorithm 15 shows the skeleton of the problem header. The main method is the virtual `objectiveFunction`, which takes a point as an argument, and returns the same point, with its value updated. The mandatory attributes, fixing the problem bounds, are set up by the constructor.

Once the problem implemented, it is easy to optimize it with the available metaheuristics, and benefit of the automated data processing. Figure 9 shows examples of graphics directly produced by Ometah. The source code is available online, in the *registration* module of the Open Metaheuristic project[15], and the details of the problem are presented in two publications[17,16].

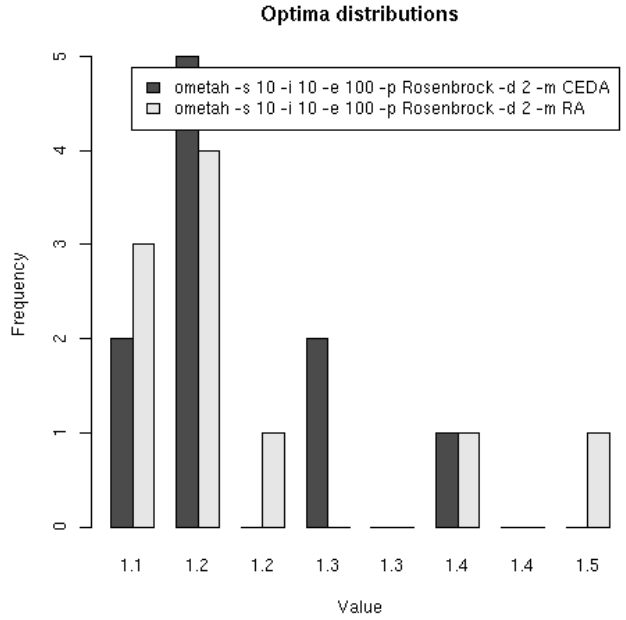


Fig. 7. Distribution of the sample on the values space for an Estimation of Distribution Algorithm (CEDA) and a Random Search (RA) optimizing the Rosenbrock objective function with two dimensions. The distributions are obtained for each iteration, at the intensification step, through 50 runs, with a sample of 10 points, limited by 100 evaluations of the problem.

Algorithm 15 Skeleton of a registration problem using the CImg imaging library

```
// The class inherits from the oMetah problem base class
class itsRegistration : public itsProblem
{
public:
// The first image, a CImg type
CImg<unsigned char> img1;
// The second image, to register
CImg<unsigned char> img2;
// The objective function computes the similarity
itsPoint objectiveFunction(itsPoint point);
// Constructor
itsRegistration();
}
```

5. Conclusion

We have shown that population metaheuristics can be viewed as algorithms handling a probabilistic sampling of a probability distribution, representing the objective

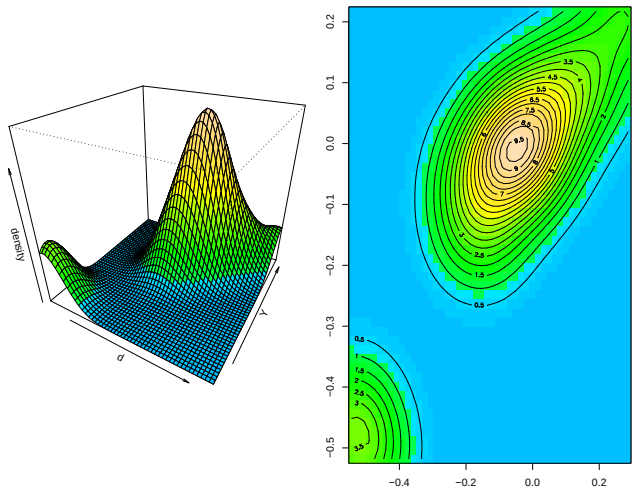


Fig. 8. Distribution on the solutions space of the optima found by a simple Random Search on the Rosenbrock problem with two dimensions. The optima are obtained through 50 runs, with a sample of 10 points, limited by 100 evaluations of the problem. The density is estimated with a two-dimensional kernel density method, with an axis-aligned bivariate normal kernel.

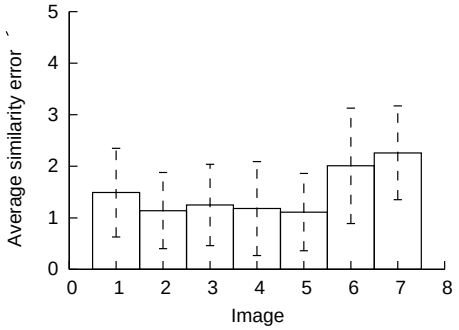
function of an optimization problem. These algorithms are iteratively manipulating the sample thanks to three processes: learning, intensification and diversification. These metaheuristics can thus be viewed as adaptive learning search algorithms.

The ALS approach can be used to implement metaheuristics, thus facilitating their design and analysis. Indeed, implementing a metaheuristic in the ALS framework consists in implementing the three main steps. As this simplifies the implementation, the metaheuristics can be more easily compared and studied.

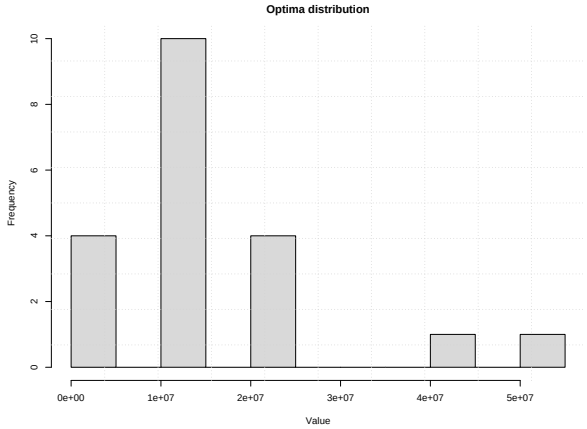
The ALS framework is implemented in the Open Metaheuristics library. This project proposes a XML format for structuring the output of metaheuristics, in order to facilitate the comparison between these optimization algorithms. Moreover, the link between metaheuristics and problems can be handled by several communication protocols, permitting to connect external problems and/or optimizers with internal ones.

References

1. E. H. L. Aarts and P. J. M. Van Laarhoven. Statistical cooling : a general approach to combinatorial optimisation problems. *Philips Journal of Research*, 40:193–226, 1985.
2. S. Baluja. Population-based Incremental Learning: A Method for Integrating Genetic Search Based Function Optimization and Competitive Learning. Technical Report CMU-CS-94-163, Carnegie Mellon University, 1994.
3. S. Baluja. Genetic Algorithms and Explicit Search Statistics. *Advances in Neural Information Processing Systems*, 9:319–325, 1997.
4. S. Baluja and R. Caruana. Removing the Genetics from the Standard Genetic Algo-



(a) Mean and standard deviations of the similarity error (giving the quality of the registration of two images), obtained with an ant colony algorithm, hybridicized with a Nelder-Mead search, on several images



(b) Distribution of the optimums values for a single image, obtained with an estimation of distribution algorithm.

Fig. 9. Example of results obtained with the implementation of the registration problem optimized by metaheuristics in the oMetah framework.

rithm. In A. Prieditis and S. Russel, editors, *International Conference on Machine Learning*, pages 38–46, Lake Tahoe, California, 1995. Morgan Kaufmann.

5. S. Baluja and S. Davies. Fast probabilistic modeling for combinatorial optimization. In *Fifteenth National Conference on Artificial Intelligence, Tenth Conference on Innovative Applications of Artificial Intelligence*, Madison, Wisconsin, 1998.
6. P. A. N. Bosman and D. Thierens. An algorithmic framework for density estimation based evolutionary algorithm. Technical Report UU-CS-1999-46, Utrech University, 1999.
7. P.A.N. Bosman and D. Thierens. Continuous iterated density estimation evolutionary algorithms within the IDEA framework. In M. Muehlenbein and A.O. Rodriguez, editors, *Proceedings of the Optimization by Building and Using Probabilistic Mod-*

els OBUPM Workshop at the Genetic and Evolutionary Computation Conference GECCO-2000, pages 197–200, San Francisco, California, 2000. Morgan Kauffmann.
8. P.A.N. Bosman and D. Thierens. IDEAs based on the normal kernels probability density function. Technical Report UU-CS-2000-11, Utrecht University, 2000.
9. V. Cerny. Thermodynamical approach to the traveling salesman problem : an efficient simulation algorithm. *J. of Optimization Theory and Applications*, 45(1):41–51, 1985.
10. A. Colorni, M. Dorigo, and V. Maniezzo. Distributed Optimization by Ant Colonies. In F. Varela and P. Bourgine, editors, *Proceedings of ECAL'91 - First European Conference on Artificial Life*, pages 134–142, Paris, France, 1992. Elsevier Publishing.
11. M. Creutz. Microcanonical Monte Carlo simulation. *Physical Review Letters*, 50(19):1411–1414, May 1983.
12. P. M. C. De Oliveira. Broad Histogram : An Overview. *arxiv :cond-mat/0003300v1*, 2000.
13. K. Deb and P. N. Suganthan. Special Session on Real-Parameter Optimization. In *IEEE Congress on Evolutionary Computation*, September 2005.
14. R. Dorne and C. Voudouris. *Metaheuristics: computer decision-making*, chapter HSF: the iOpt's framework to easily design metaheuristic methods, pages 237–256. Kluwer Academic Publishers, Norwell, MA, USA, 2004.
15. J. Dréo, J.-Ph. Aumasson, and W. Tfaili. Open Metaheuristics. http://ometah.berlios.de, 2005.
16. J. Dréo, J.-C. Nunes, and P. Siarry. Robust rigid registration of retinal angiograms through optimization. *Computerized Medical Imaging and Graphics*, to appear, 2007. DOI: 10.1016/j.compmedimag.2006.07.004.
17. J. Dréo, J.C. Nunes, P. Truchetet, and P. Siarry. Retinal angiogram registration by estimation of distribution algorithm. In *6th IFAC Symposium on Modelling and Control in Biomedical Systems (IFAC 2006)*, 2006.
18. J. Dréo, A. Pétrowski, P. Siarry, and É. D. Taillard. *Metaheuristics for hard optimization*. Springer, 2006.
19. J. E. Eiben, A. E. Smith. *Introduction to Evolutionary Computing*. Springer Verlag, 2003.
20. A. M. Ferrenberg and R. H. Swendsen. Optimized Monte Carlo Data Analysis. *Physical Review Letters*, 63:1195, 1989.
21. A. Fink and S. Voß. Generic metaheuristics application to industrial engineering problems. *Computers and Industrial Engineering*, 37(1-2):281–284, 1999.
22. L. J. Fogel, A. J. Owens, and M. J. Walsh. *Artifical Intelligence through Simulated Evolution*. Wiley, 1966.
23. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison Wesley Professional, 1994.
24. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Abstraction and reuse of object-oriented design. *Lecture Notes in Computer Science*, 707:406–431, 1993.
25. D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine learning*. Addison-Wesley, 1989.
26. G. Harik. Linkage learning in via probabilistic modeling in the EcGA. Technical Report 99010, IlliGAL, 1999.
27. G. Harik, F. G. Lobo, and D. E. Goldberg. The compact genetic algorithm. In *IEEE Conference on Evolutionary Computation*, pages 523–528, 1998.
28. W. K. Hastings. Monte Carlo sampling method using Markov chains and their applications. *Biometrika*, 57, 1970.

29. J. H. Holland. Outline for logical theory of adaptive systems. *J. Assoc. Comput. Mach.*, 3:297–314, 1962.
30. K. Hukushima and K Nemoto. Exchange Monte Carlo method and application to spin glass simulations. *Journal of the Physical Society of Japan*, 65:1604–1608, 1996.
31. Y. Iba. Population Annealing: An approach to finite-temperature calculation. In *Joint Workshop of Hayashibara Foundation and SMAPIP*. Hayashibara Forum, 2003.
32. M. Jenkinson and S. Smith. A global optimisation method for robust affine registration of brain images. *Medical Image Analysis*, 5:143–156, 2001.
33. M. Jones. *An Object-Oriented Framework for the Implementation of Search Techniques*. PhD thesis, University of East Anglia, 2000.
34. Maarten Keijzer, J. J. Merelo, G. Romero, and Marc Schoenauer. Evolving objects: A general purpose evolutionary computation library. In *Artificial Evolution*, pages 231–244, 2001.
35. J. Kertesz and I. Kondor, editors. *Advances in Computer Simulation*, chapter Introduction To Monte Carlo Algorithms. Springer-Verlag, 1998.
36. S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.
37. P. Larrañaga and J.A. Lozano, editors. *Estimation of Distribution Algorithms, A New Tool for Evolutionary Computation*. Genetic Algorithms and Evolutionary Computation. Kluwer Academic Publishers, 2002.
38. F. Liang and W. H. Wong. Evolutionary Monte Carlo: Application to C_p Model Sampling and Change Point Theorem. *Statistica Sinica*, 10, 2000.
39. N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *Journal of Chemical Physics*, 21:1087–1092, 1953.
40. N. Monmarché, E. Ramat, G. Dromel, M. Slimane, and G. Venturini. On the similarities between AS, BSC and PBIL: toward the birth of a new meta-heuristics. E3i 215, Université de Tours, 1999.
41. N. Monmarché, N. Ramat, L. Desbarat, and G. Venturini. Probabilistic search with genetic algorithms and ant colonies. In A.S. Wu, editor, *Proceedings of the 2000 Genetic and Evolutionary Computation Conference Workshop*, pages 209–211, 2000.
42. H. Mühlenbein and G. Paaß. From recombination of genes to the estimation of distributions I. Binary parameters. *Lecture Notes in Computer Science 1411: Parallel Problem Solving from Nature*, PPSN IV:178–187, 1996.
43. J. A. Nelder and R. Mead. A simplex method for function minimization. *Computer Journal*, 7:308–313, 1965.
44. M. E. J. Newman and R. G. Palmer. Error estimation in the histogram Monte Carlo method. *arxiv:cond-mat/98043006*, 1998.
45. Y Okamoto and U. H. E. Hansmann. Thermodynamics of helix-coil transitions studied by multicanonical algorithms. *Journal Physical Chemistry*, 99:11276–11287, 1995.
46. I. Rechenberg. *Cybernetic Solution Path of an Experimental Problem*. Royal Aircraft Establishment Library Translation, 1965.
47. M.G.C. Resende. Greedy randomized adaptive search procedures (GRASP). Technical Report TR 98.41.1, AT&T Labs-Research, 2000.
48. N. Ritter, R. Owens, J. Cooper, R. H. Eikelboom, and P. P. V. Saarloos. Registration of stereo and temporal images of the retina. *IEEE Trans. On Medical Imaging*, 18:404–418, 1999.
49. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.
50. G. Syswerda. Simulated Crossover in Genetic Algorithms. In L. D. Whitley, editor,

Second workshop on Foundations of Genetic Algorithms, pages 239–255, San Mateo, California, 1993. Morgan Kaufmann.
51. É. D. Taillard. A statistical test for comparing success rates. In *Metaheuristic international conference MIC'03*, Kyoto, Japan, August 2003.
52. É. D. Taillard, L. M. Gambardella, M. Gendreau, and J.-Y. Potvin. Adaptive Memory Programming: A Unified View of Meta-Heuristics. *European Journal of Operational Research*, 135(1):1–16, 1998.
53. E. Triki, Y. Collette, and P. Siarry. A theoretical study on the behavior of simulated annealing leading to a new cooling schedule. *European Journal of Operational Research*, 166:77–92, 2005.
54. D. Tschumperlé. Cimg library. Available at `http://cimg.sourceforge.net/`, 2005.
55. S. Voß and D. L. Woodruff. *Optimization Software Class Libraries*. OR/CS Interfaces Series. Kluwer, Dordrecht, 2002.
56. O. Wendt and W. König. Cooperative Simulated Annealing: How Much Cooperation is Enough ? Technical Report 97-19, Institute of Information Systems, Goethe University, Frankfurt, 1997.