University of Castilla-La Mancha

A publication of the
Department of Computing Systems

LiO: an easy and flexible library of metaheuristics

by

Juan Luis Mateo and Luis de la Ossa

Technical Report #**DIAB-06-04-1** (Ver. 2) November 2007

DEPARTAMENTO DE SISTEMAS INFORMTICOS
ESCUELA POLITCNICA SUPERIOR
UNIVERSIDAD DE CASTILLA-LA MANCHA
Campus Universitario s/n
Albacete - 02071 - Spain
Phone +34.967.599200, Fax +34.967.599224

LiO: an easy and flexible library of metaheuristics

Juan Luis Mateo Luis de la Ossa

November 27, 2006

Contents

Chapter 1

Introduction to LiO

This book describe LiO, that stands for Library of Optimization, a new tool in the field of search algorithms and combinatorial optimization. This library has been developed by SIMD group at the Computing Systems Department in University of Castilla-La Mancha.

In this field there are already several others tools, but LiO has a new approach and intend to serve researchers, teachers, students and newbies, each one on his level to work with metaheuristics. The main aim set at the beginning of this project was that the result should be very easy to learn and to use. Besides, another aim was generality, that is, LiO should be used in many different scenarios.

That lead us to define three profiles of use:

- The use of LiO as support for teaching. Anybody could configure and run this kind of algorithms without experience.
- Provide a set of algorithms a resources of common use in order to let researchers use it without effort and allowing him to save time.
- The capability to adapt itself to new specific needs like adding new algorithms or resources, changing the existing ones, or working with other data types more complexes.

Another important characteristic is that LiO has been coded in Java. This gives it portability and makes possible run LiO over almost all platforms nowadays.

Part I

Using LiO as a tool

Chapter 2

The LiO GUI

The LiO graphical user interface is the easiest way of deal with metaheuristics. Anybody can execute several algorithms to solve distint problems, modify every parameter and see the results in various views. The main window of this gui is shown in figure 2.1. In this figure it can been seen the different areas which LiO gui is made of.

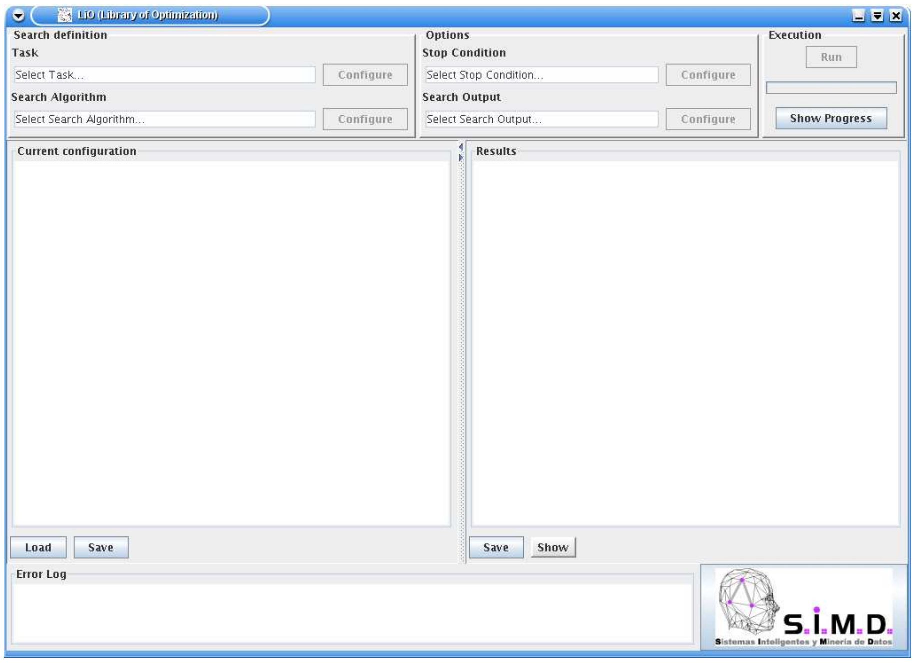


Figure 2.1: LiO grafical user interface main window

First of all, at the upper left corner is the *Search definition* area. On this area it can be selected which problem it's going to be solved and which search algorithm it'll be used. Moreover, when both, task and algorithm, have been chosen, their parameters can be modified using *Configure* buttons. This configuration, task and algorithm objects and its parameters recursively, will be shown in the *Current configuration* area. With the buttons underneath it can be saved to a file, to use it again after, and be loaded a saved configuration. A configuration file stores all configuration parameters, what algorithm is used, what task and more. In chapter 3 it will be seen more details about configuration file.

To the right of *Search definition* area is the *Options* area. Here it can be changed the amount computation the search algorithm will do at maximum if optimum value it is not found (*Stop Condition*). It can be set maximum number of evaluations or how much time at maximum, among other options. It can also be changed the information shown as result of a execution (*Search Output*). In this case there is an interesting option, called *tag*, by mean of that it can be set a text which will be printed with the output to identify each execution in an easier way. All execution results will be shown in the *Results* text area in reverse order, that is, last execution will be at top. The content of *Results* text area can be saved to a file with the *Save* button, and with *Show* button it can be chosen between three charts. The first one shows evaluations by iterations, the second one shows fitness evolution by iterations and the last one shows the convergence velocity defined as

$$C(t) = \log \sqrt{\frac{f_{max}(0)}{f_{max}(t)}},$$

where $f_{max}(i)$ is the best fitness value at the i^{th} iteration.

Last area in top panel, *Execution*, is to manage execution process. *Run* button starts execution or allow to stop it if it's running. *Show Progress* button shows an small window in which it can be seen evaluations, iterations and fitness values. This values are refreshed every second.

At the bottom of the main window there is the *Error log* area where will be shown all notifications about errors or warnings both in configuration process and execution.

Up to now, it has been seen LiO graphical interface in a static way. Next it is time to see how we can use all this elements to run, learn and fun with metaheuristics. First of all, it is necessary to select which problem and search algorithm it's wanted to run. This is done by clicking at the corresponding text field as it can be seen in figure 2.2 and 2.3. Here it can be chosen from these lists.

When it is selected a task and a search algorithm, default configuration for these objects is shown in the *Current configuration* area, also defaults objects for stop criterion and output options are selected. This is depicted in figure 2.4.

As it is been said, this is the default configuration, but it can be changed, either loading a configuration file or using the *Configure* buttons. If it is used

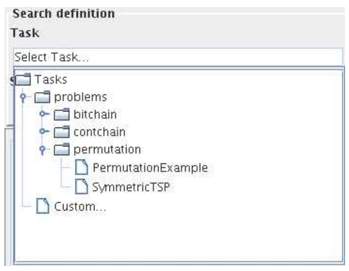


Figure 2.2: List for selecting problems included in LiO.

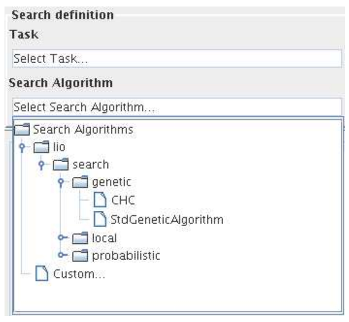


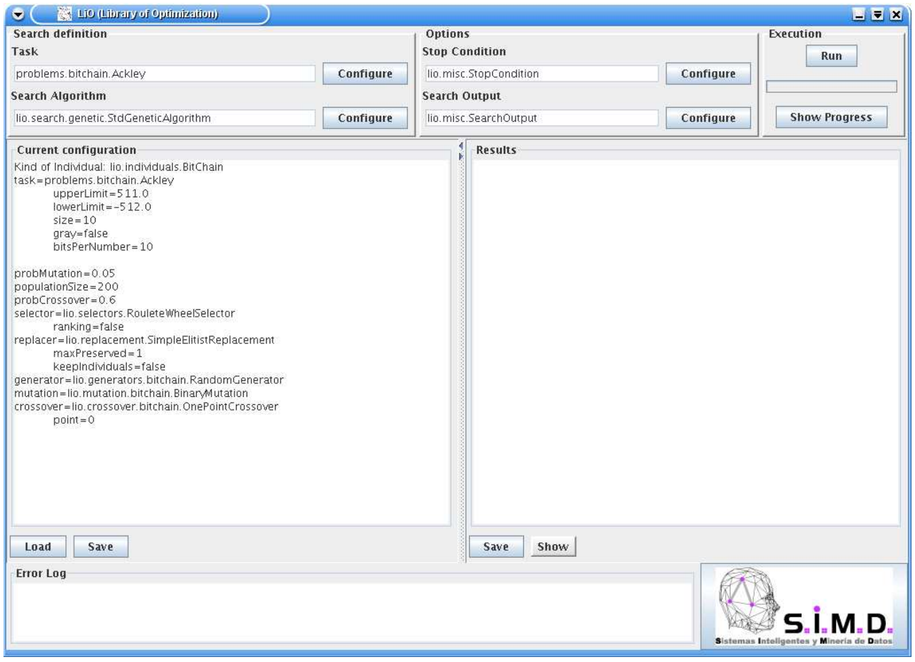
Figure 2.3: List for selecting search algorithms included in LiO.

the later option, the windows depicted in figures 2.5, 2.6, 2.7 and 2.8 will appear, for configuring task, search algorithm, stop condition and search output, respectively. In this windows it can be changed values and selected others object for operators as shown in figure 2.6, where a list to select the selector operator has appeared. For objects that have parameters to configure, another *Configure* button is presented enabled, and if it is clicked on it another window will appear like this one to change its own parameters. If mouse pointer is hold over each parameter value it can be seen a short description, if it is available.

When the configuration for these objects is finished, it is time to run the. algorithm. Then the *Run* button must be clicked and its label change to *Stop*, and it can be clicked again to stop execution. Besides the progress bar below will start to move and it can be clicked on *Show Progress* button to see in a small window how evolves our algorithm. This is shown in figure 2.9.

Once the execution is over the results are shown in *Results* area as is presented in figure 2.10.

To get a richer information for this execution, a couple of charts, shown in figures 2.11 and 2.12, can be used through *Show* button. The first chart represents number of evaluations in total done per each generation. The second chart shows the best fitness value found, globally, per each generation. If right



Task: problems.bitchain.Ackley

Search Algorithm: lio.search.genetic.StdGeneticAlgorithm

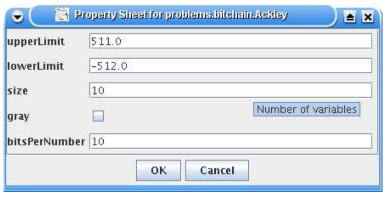
Stop Condition: lio.misc.StopCondition

Search Output: lio.misc.SearchOutput

Current configuration

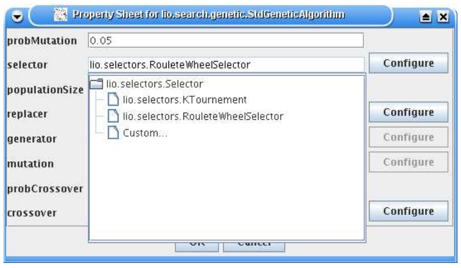
```
Kind of Individual: lio.individuals.BitChain
task=problems.bitchain.Ackley
        upperLimit=511.0
        lowerLimit=-512.0
        size=10
        gray=false
        bitsPerNumber=10

probMutation=0.05
populationSize=200
probCrossover=0.6
selector=lio.selectors.RouleteWheelSelector
        ranking=false
replacer=lio.replacement.SimpleElitistReplacement
        maxPreserved=1
        keepIndividuals=false
generator=lio.generators.bitchain.RandomGenerator
mutation=lio.mutation.bitchain.BinaryMutation
crossover=lio.crossover.bitchain.OnePointCrossover
        point=0
```

Figure 2.4: Configuration shown after selecting task and search algorithm.



upperLimit	511.0
lowerLimit	-512.0
size	10
gray	☐
bitsPerNumber	10

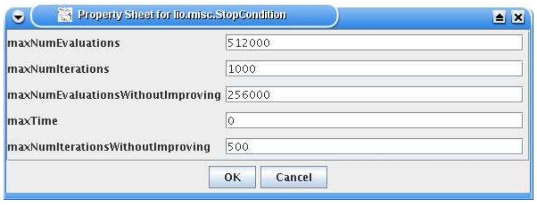
Figure 2.5: Window to modify task parameters.


Property Sheet for lio.search.genetic.StdGeneticAlgorithm
Configure
populationSize
replacer
generator
mutation
probCrossover
crossover
lio.selectors.Selector
lio.selectors.KTournement
lio.selectors.RouleteWheelSelector
Custom...

probMutation: 0.05

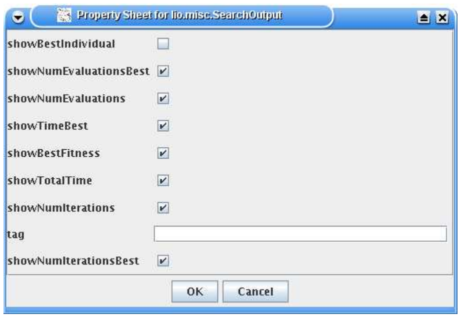
selector: lio.selectors.RouleteWheelSelector

Figure 2.6: Window to modify search algorithm parameters.


Property Sheet for lio.misc.StopCondition

maxNumEvaluations	512000
maxNumIterations	1000
maxNumEvaluationsWithoutImproving	256000
maxTime	0
maxNumIterationsWithoutImproving	500

Figure 2.7: Window to modify stop condition parameters.

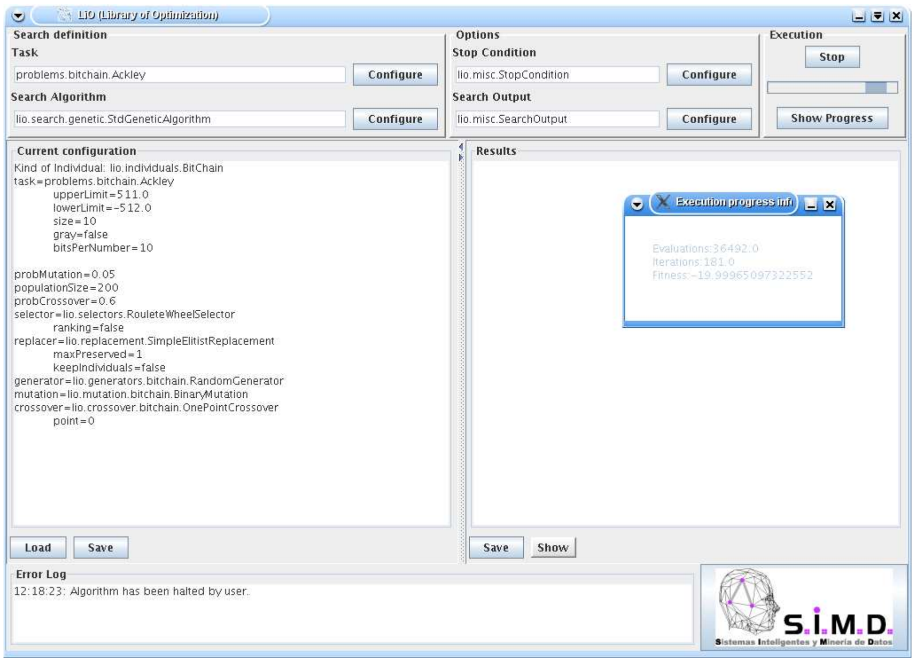

Property Sheet for lio.misc.SearchOutput

showBestIndividual	☐
showNumEvaluationsBest	✓
showNumEvaluations	✓
showTimeBest	✓
showBestFitness	✓
showTotalTime	✓
showNumIterations	✓
tag	
showNumIterationsBest	✓

Figure 2.8: Window to modify search output parameters.

botton mouse is clicked over this charts, a pop-up menu will appear with some options like zoom in and zoom out, change its legend, its font type and size, and save this chart as a picture.

And finally, there are two other methods to load objects in LiO (search algorithm, task, operators, etc.). In the previous figures showing a list of objects to load it can be seen an item at the end called *Custom*. If this item is selected, then the *Load Object* window will appear in which it can be written a class name (full qualified) if the required object is in class path, or a file name for a Java class file (with full path) otherwise. To select a file easily there is the option by clicking the *Explore* button. If this happend a standard open file dialog will appear in which it can be selected a file from file system. This can bee seen in figure 2.13



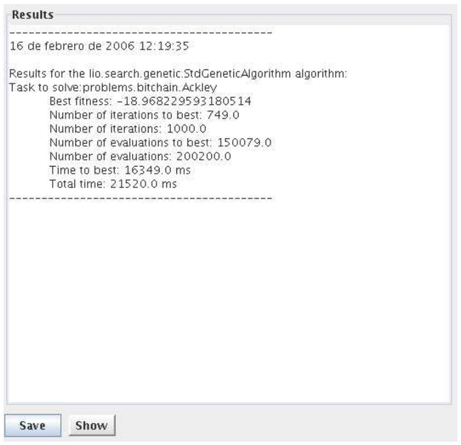
Task: problems.bitchain.Ackley
Search Algorithm: lio.search.genetic.StdGeneticAlgorithm
Stop Condition: lio.misc.StopCondition
Search Output: lio.misc.SearchOutput

```
Kind of Individual: lio.individuals.BitChain
task=problems.bitchain.Ackley
        upperLimit=511.0
        lowerLimit=-512.0
        size=10
        gray=false
        bitsPerNumber=10

probMutation=0.05
populationSize=200
probCrossover=0.6
selector=lio.selectors.RouleteWheelSelector
        ranking=false
replacer=lio.replacement.SimpleElitistReplacement
        maxPreserved=1
        keepIndividuals=false
generator=lio.generators.bitchain.RandomGenerator
mutation=lio.mutation.bitchain.BinaryMutation
crossover=lio.crossover.bitchain.OnePointCrossover
        point=0
```

```
Evaluations:36492.0
Iterations:181.0
Fitness:-19.99965097322552
```

```
12:18:23: Algorithm has been halted by user.
```

Figure 2.9: Main window while an execution is in progress.



```
-----------------------------------------
16 de febrero de 2006 12:19:35

Results for the lio.search.genetic.StdGeneticAlgorithm algorithm:
Task to solve:problems.bitchain.Ackley
        Best fitness: -18.968229593180514
        Number of iterations to best: 749.0
        Number of iterations: 1000.0
        Number of evaluations to best: 150079.0
        Number of evaluations: 200200.0
        Time to best: 16349.0 ms
        Total time: 21520.0 ms
-----------------------------------------
```

Figure 2.10: Results provided after algorithm finish its execution.

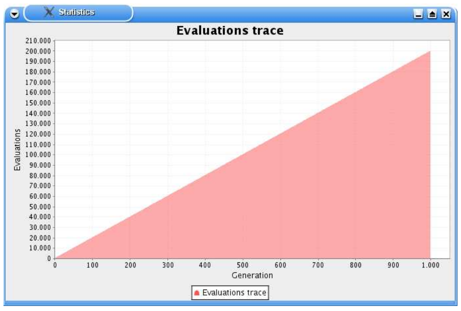


Figure 2.11: Chart showing evaluations by generations for the last execution.

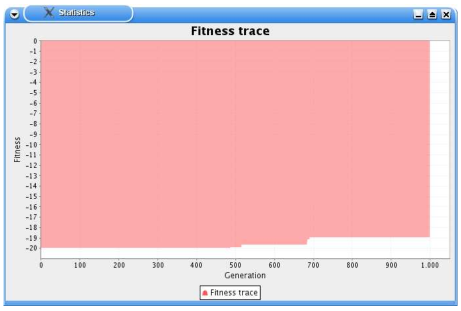
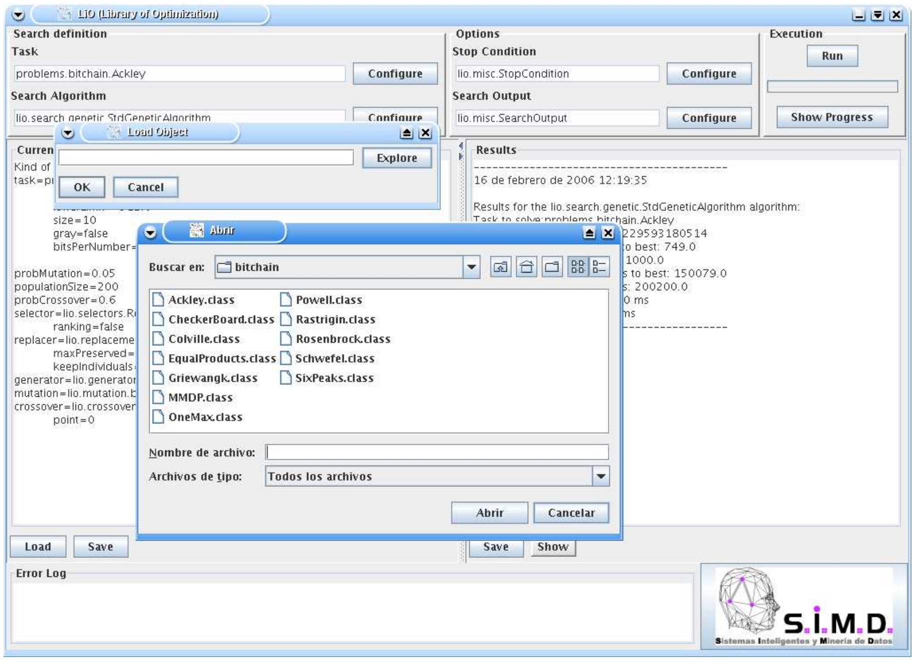


Figure 2.12: Chart showing fitness by generations for the last execution.



Task: problems.bitchain.Ackley

Search Algorithm: lio.search.genetic.StdGeneticAlgorithm

Stop Condition: lio.misc.StopCondition

Search Output: lio.misc.SearchOutput

```
Kind of
task=pı
        size=10
        gray=false
        bitsPerNumber=

probMutation=0.05
populationSize=200
probCrossover=0.6
selector=lio.selectors.R
        ranking=false
replacer=lio.replaceme
        maxPreserved=
        keepIndividuals
generator=lio.generator
mutation=lio.mutation.b
crossover=lio.crossover
        point=0
```

```
-----------------------------------------
16 de febrero de 2006 12:19:35

Results for the lio.search.genetic.StdGeneticAlgorithm algorithm:
Task to solve:problems.bitchain.Ackley
                                  229593180514
                                  o best: 749.0
                                  1000.0
                                  s to best: 150079.0
                                  s: 200200.0
                                  0 ms
                                  ms
                                  -----------------
```

Buscar en: bitchain

```
Ackley.class        Powell.class
CheckerBoard.class  Rastrigin.class
Colville.class      Rosenbrock.class
EqualProducts.class Schwefel.class
Griewangk.class     SixPeaks.class
MMDP.class
OneMax.class
```

Nombre de archivo:

Archivos de tipo: Todos los archivos

Figure 2.13: Windows to select objects not included in LiO.

Chapter 3

The configuration file

A configuration file is a plain text file that in each line has an assignment for some parameter. These parameters can be for any search algorithm and can set values for an specific problem or for the objects that control the output and the stop criterion. An example of a configuration file is shown in figure 3.

In a configuration file lines that start with a sharp character (#) are ignored, so are used to include comments. Each assignment line is made of a left part, an equal sign and a value. The left part show which parameter has to be fixed. Each parameter must to be written in a canonical form like classes and packages in Java, that is, all objects which contain recursively that parameter have to be written in hierarchycal order. Values can be numbers, integer and decimal, strings, enclosed betwen doubled quotes if contains spaces, and Java classes for setting what resource must be used by the current resource.

Parameters and objects names are those declared in the approppiate Java class file. For example, if it is wanted to change the *point* parameter for the one point crossover in a genetic algorithm, then it should be known that in the genetic algorithm class the crossover resource is named as *crossover*, and in the one point crossover class the point parameter is named *point*. Then the line necessary, if the new value is 15, is the following: `crossover.point=15`.

Obviously, it not easy to know and remember the name of all parameter of all objects in LiO to make a configuration file. Then it can be used LiO gui interface to configure all parameter necessary and save that configuration to a file. Now, if it's needed just to change a few values to make more executions, it can be used that file as a template.

If it's necessary to establish parameters for others objects but search algorithm, then prefixes have to used. For all parameters regarding problem `task` must be used, `searchOutput` for the object controlling the output and `stopCondition` for the object controlling the stop criterion.

In a configuration file is not needed specify values for all parameters. In this case it is chosen the apropiate object from the LiO internal configuration (see

```
#search algorithm parameters
search=lio.search.genetic.StdGeneticAlgorithm
probMutation=0.05
populationSize=200
probCrossover=0.6

selector=lio.selectors.RouleteWheelSelector
selector.ranking=false

replacer=lio.replacement.SimpleElitistReplacement
replacer.maxPreserved=1
replacer.keepIndividuals=false

generator=lio.generators.bitchain.RandomGenerator

mutation=lio.mutation.bitchain.BinaryMutation

crossover=lio.crossover.bitchain.OnePointCrossover
crossover.point=0

#task parameters
task=problems.bitchain.Ackley
task.upperLimit=511.0
task.lowerLimit=-512.0
task.size=10
task.gray=false
task.bitsPerNumber=10

#output options
searchOutput.tag="This text help to identify this execution"

#stop criterion
stopCondition.maxTime=5000
```

Figure 3.1: Configuration file example

chapter 9), and its values are those defined in the class. We can for different algorithms and resources and use this file in several execution, here parameters not matching a given resource are ignored without warning.

Chapter 4

The command line

Running LiO from the command line is easy because all algorithm will have the same interface. All it is needed is to select a class implementing some search algorithm as the Java entry point. For example, assuming LiO classes are in the classpath, any search algorithm can be execute with `-h` flag to see the options. The result is shown in figure 4.1.

```
$ java lio.search.genetic.StdGeneticAlgorithm -h

Generic options for the search.

        -file <conf_file>  Name of the file containing the
            configuration for the search
        -param <param>=<value>  Name of the parameter and
            value.
        -output <out_file>  Name of the file to output
            results.
        -b  To avoid show results through standard output.
        -h  Show this help message.
```

Figure 4.1: Help message for any search algorithm.

The are two kind of options. In the first group are `-file` and `-param` and control the configuration of all objects involve in the search process. The second one, `-output` and `-b`, tells how will be generated the algorithm's output.

More specifically, the `-file` option is used to set a configuration file which contain a list of pairs of parameter and value as it had been seen in chapter 3. The `-param` option fix the given parameter to an specific value, here it has to be used the same format as a line in a configuration file. It's important to realize that always if some parameter is set both in configuration file and with a `-param` option, the one specified in configuration file is ignored. Later, some examples will be seen to clarify these concepts.

Concerning output parameters, it is needed to say that always all algorithms yield a formated output in several lines unless the `-b` option is provided. In these lines is presented the name of the search algorithm, the name of the problem to solve, the best result found, and information about iterations/generations, evaluations and time. With `-output` option it can be provided a file name in which the execution information is added to the end in a line and all fields are separated by a tab character.

Now it is time to some examples. To run an algorithm the only thing is necessary to provide it's the problem to solve. Then, in the first example the `-param` option will be used to set a task object. The command line used and its result is shown in figure 4.2

```
$ java lio.search.genetic.StdGeneticAlgorithm -param task=
    problems.bitchain.Ackley

Results for the lio.search.genetic.StdGeneticAlgorithm
    algorithm:
Task to solve:problems.bitchain.Ackley
         Best fitness: -19.264344891844445
         Number of iterations to best: 931.0
         Number of iterations: 1000.0
         Number of evaluations to best: 186427.0
         Number of evaluations: 200200.0
         Time to best: 18757.0 ms
         Total time: 20146.0 ms
```

Figure 4.2: Simple execution setting problem to solve

Next it can be seen how to use output options to send execution result to a file. In this case the `-b` option is provided to avoid standard output, and results will be stored in a file call Ackey.out. The order of all fields are the same as in standard formated output. This is shown in figure 4.3.

```
$ java lio.search.genetic.StdGeneticAlgorithm -param task=
    problems.bitchain.Ackley -b -output Ackley.out
$ cat Ackley.out
         lio.search.genetic.StdGeneticAlgorithm   problems.
             bitchain.Ackley     -19.84739791983819   921.0
             1000.0   184434.0     200200.0     18824.0  20479.0
```

Figure 4.3: Example of use of output options

In figure 4.4 it can be seen how to use a configuration file and how paramters set by `-param` option have higher priority than those set in the configuration file. Here it is used the configuration file shown in chapter 3, and in that file was provided a value for the paramter `tag` for the search output object. It can be seen how this information is shown in the first line of the results and it is taken from the `-param` option. In previous examples this line was empty.

```
$ java lio.search.genetic.StdGeneticAlgorithm -param
    searchOutput.tag=example-1 -file execution1.conf
example-1
Results for the lio.search.genetic.StdGeneticAlgorithm
    algorithm:
Task to solve:problems.bitchain.Ackley
         Best fitness: -19.999982525215277
         Number of iterations to best: 114.0
         Number of iterations: 237.0
         Number of evaluations to best: 23012.0
         Number of evaluations: 47600.0
         Time to best: 2446.0 ms
         Total time: 5006.0 ms
```

Figure 4.4: Execution showing both optiong `-param` and `-file`

With LiO some resources can be loaded directly from its compiled class file. To do this the class file name can be given, with its path, and LiO will load the resource implemented in that file. This is depicted in figure 4.5.

```
$ java  lio.search.genetic.StdGeneticAlgorithm -param task=
    problems.bitchain.Ackley -param crossover=custom/
    CustomOnePointCrossover.class

Results for the lio.search.genetic.StdGeneticAlgorithm
    algorithm:
Task to solve:problems.bitchain.Ackley
         Best fitness: -19.907290030232772
         Number of iterations to best: 817.0
         Number of iterations: 1000.0
         Number of evaluations to best: 163610.0
         Number of evaluations: 200200.0
         Time to best: 16884.0 ms
         Total time: 20694.0 ms
```

Figure 4.5: Use of an external resource by its class file.

Finally, to end with use of command line, is presented a tool which can show all information about parameters of all resources in LiO. This functionality is in the `lio.misc.DescribeResource` class. This class should be execute with a full qualified class name as argument and it presents as result useful information for that class. The given class must be accesible in Java class path. For basic data-types parameters their default values are shown, and for each element a sort description can be shown (`Tip` field). This description is given by the programmer. An example of this is in figure 4.6.

```
$ java lio.misc.DescribeResource lio.search.genetic.
    StdGeneticAlgorithm
lio.LiOSearch=lio.search.genetic.StdGeneticAlgorithm
Description: Implements the standard genetic algorithm
Resources:
         @ selector:
         Tip: Selects individuals from a population in order to
              cross them.
         @ replacer:
         Tip: Generates a new population by keeping some of the
              individuals in the former one.
         @ generator:
         Tip: Generates the initial population.
         @ mutation:
         Tip: Operator of mutation.
         @ crossover:
         Tip: Operator of crossover.
Parameters:
         & probMutation of type double = 0.05
         Tip: Probability of mutation.
         & populationSize of type int = 200
         Tip: Size of the population.
         & probCrossover of type double = 0.6
         Tip: Probability of crossover.
```

Figure 4.6: Use of `lio.misc.DescribeResource` to see information about resources.

Part II

Programming metaheuristics with LiO

Chapter 5

Introduction

In the previous chapters it has been shown how to instanciate and execute the algorithms available in LiO. However, in most cases the user will need to modify the resources of the library, or even create new ones, to perform his work. In this part we go into some aspects of the library that makes possible to cover such needs.

As it will be seen, the software architecture of LiO make it totally expansible and flexible, since new resurces can be incorporated to it inmediatly and indefinitely. Moreover, such resources can be reused in programs external to the library, so it can also be seen as a repository of objects that prenvents researchers from implementing a big amount of code.

In order carry out progressive approach to programming with LiO, we firstly explain the basic structures used to encode the potential solutions to the problem and, afterwards, we will study how to build resources that work with such data. Last, it will be explained how to build search algorithms from this resources, either not integrated or integrated in LiO.

Chapter 6

Individuals and data types

6.1 Individuals

The abstract class `Individual` is used to represent each potential solution to a problem regardless of the way it is encoded. It basically implements three main functions:

- `public double value()`: It returns the result of evaluating the individual with the current function. This value is stored in the private member `value`.
- `public boolean isEvaluated()`: Allows knowing if the individual has already been evaluated. It is used by `value()` so as to avoid carrying out the same evaluation several times.
- `public void change()`: This method changes the state of the individual to "not evaluated".

This class has also some members common to every kind of individual as it can be `size`.

Besides `Individual`, LiO uses a particular class (descendent of the former one) to work with each particular encoding. For the time being there are three kinds of data types supported on the library:

- `BitChain` : Chains of bits.
- `ContChain` : Chains of real numbers.
- `Permutation` : Permutations.

As it can be seen, in figure 6.1 elements in each object can be accessed directly if necessary.

In chapter 10, it will be described how algorithms in LiO can work with particular (*custom*) encodings, different than the existing.

```
BitChain bCh = new BitChain(100);
int[] bcElements = bCh.getElements();
ContChain cCh = new ContChain(100);
double[] ccElements = cCh.getElements();
Permutation perm = new Permutation(100);
int[] perm = per.getElements();
```

Figure 6.1: Example of access to the elements of the individuals.

6.2 Defining data types: LiOBounds

Besides the data type used to encode the solutions of a problem, it becomes necessary some other information about them such as their size, range of values that can take each variable, etc.

Objects descendent of `lio.individuals.LiOBounds` are used to completely specify the potential solutions. This abstract class contains a member `size` which determines their maximum size.

Each class descendent of `LiOBounds` is used to work with a certain encoding. For instance, figure 6.2 shows a portion of code from the class `ContChainBounds`. As it can be seen, it is used to define problems that work with chains of real numbers (objects `ContChain`), and provides members and methods to define the size of these chains and the range of values that can take each variable.

```
public class ContChainBounds extends LiOBounds{
     double[] lowerlimits,upperlimits;
     public ContChaintBounds(int pSize){
             size=pSize;
             lowerLimits = new double[size];
             upperLimits = new double[size];
     }
     public void setRanges(double lower, double upper){
             Arrays.fill(lowerLimits,lower);
             Arrays.fill(lowerLimits,upper);
     }
}
```

Figure 6.2: Some code of the class ContChainBounds

LiO also provides classes to define problems which work with chains of bits and permutations.

Chapter 7

Resources and tasks

7.1 Resources

As it was mentioned before, the LiO library was initially designed to avoid implementing pieces of code which are frequently used when solving problems by means of metaheuristics and evolutionary computation. In this sense, LiO can be seen as a source of objects necessary to build search algorithms. These objects are called resources and can be operators, problems, objects to manage statistics, output settings, and even the algorithms themselves. The functionality of everyone is given by the interface they implement. For instance, all resources (objects) which implement the interface `Crossover` will be crossover operators.

From the point of view of design, a resource is an object whose behavior is known, and is thought to be as independent as possible from the rest of the resources. That philosophy makes them easy to use and useful in different contexts. This modularity also deals with the other design goal: Developing an structure which allows the library growing indefinitely.

Due to these facts the search algorithms, which are the executable part of the library, do not have complete information about the resources they need to instantiate. Therefore, it becomes necessary for a resource to describe itself, and so, this is their common functionality. Thus, the interface `lio.core.LiOResource` is implemented by every resource, and basically consists of one function `getDefinition()` that returns a description of it. The class `lio.core.LiOResourceDefinition` is the one used to handle such descriptions.

Figure 7.1 shows an example of construction of a `LiOResourceDefinition` in the implementation of the function `getDefinition()`. As it can be seen, the description contains the interface that describes the functionality of the resource, its name and description, and the members with their corresponding descriptions. In this definition we must set the kind of data that the resource is compatible with. If the resource can manage several kind of data we can omit this sentence.

```
public LiOResourceDefinition getDefinition() {
     LiOResourceDefinition def = new LiOResourceDefinition(
             "lio.crossover.Crossover"
             "lio.crossover.bitchain.OnePointCrossover"
             "Implements the one-point crossover.");
     def.addMember("point", "Point of crossover");
     def.setKindOfIndividuals("lio.individuals.BitChain");
     return def;
}
```

Figure 7.1: Implementation of getDefinition for a crossover operator

As it can be seen, definitions only provide the name of the members and no information about their data types is given, so it becomes necessary a way for the search algorithms to instantiate and set up the resources. Such way is grounded in *reflexivity* and allows the core of the library accessing the resources by only knowing their definitions.

The instantiable members of each resources can be integers, booleans, Strings, doubles or even resources. For each one of them, access functions must be implemented. These functions must follow a simple rule also used by the JavaBeans convention. Thus, if we have a member called *member* the method to put its value must be named as *setMember*, whereas the method to read its value must be implemented by *getMember*. Regard the capital letter after set or get.

Figure 7.2 shows the implementation of the access methods for the member *point* in the one point crossover.

```
public void setPoint(int pPoint) {
     point = pPoint;
}
public int getPoint() {
      return point;
}
```

Figure 7.2: Implementation of the access methods for the one point crossover

7.2 Dependent resources

In some cases, resources need some information about the task being solved to perform. For instance, a mutation which works with continuous vectors must generate numbers in the ranges defined on the task. These resources are called dependent and so implement the interface `lio.core.LiODependentResource`.

This interface only has a method `getTaskInformation(LiOTask)` that allows extracting the useful information. An example of it can be seen in Figure 7.3.

```
int size;
double upperLimits[];
double lowerLimits[];
public void getTaskInformation(LiOTask task){
   ContChainBounds bounds = (CountChainBounds)task.defineIndividuals();
   size = bounds.getSize();
   upperLimits = bounds.getUpperLimits();
   lowerLimits = bounds.getLowerLimits();
}
```

Figure 7.3: Implementation of the access methods for the one point crossover

7.3 Defining tasks

LiO uses objects descendent of the abstract class `LiOTask` to define the problems that are going to be solved. These objects must define three functions:

- `public LiOBounds defineIndividuals()`: Which returns a `LiOBounds` object that defines the potential solutions.
- `public double evaluate(Individual individual)`: Which receives an individual and returns its fitness.
- `public double optimum()`: Which returns the optimum value of the problem.[1]

Moreover, as almost every object in the library, `LiOTask` is a resource, so it must implement the interface `LiOResource`. In order to make things easier, the function `getDefinition()` is not abstract, so it only has to be written when the tasks have some accessible members.

Figure 7.4 shows the implementation of the function OneMax. As can be seen, it has an instantiable member, which determines the size of the problem

[1]This function is not abstract so it has not got to be overridden

and thus, of their potential solutions. `defineIndividuals()` returns and object `BitChainBounds` built with the current size, so that the algorithms can "know" that this problems is encoded with chains of bits. Concerning to the function `evaluate` it is worth pointing out that it receives an `Individual` as parameter and not a `BitChain`. This is done for generality. In order to access the methods of the `BitChain` a cast must be done. In this case, the optimum of the problem is given by the size. Last, we can see the implementation of `getDefinition()` , where the information about the kind of resource and its parametrizable members is given.

```
public class OneMax extends LiOTask {
   int size=100;
   public LiOBounds defineIndividuals(){
       BitChainBounds bounds = new BitChainBounds(size);
       return bounds;
   }
   public double evaluate(Individual individual){
       double fitness=0;
       BitChain bitChain = (BitChain) individual;
       int[] elements = bitChain.getElements();
       for (int i = 0; i < elements.length; i++)
            fitness = fitness + elements[i];
       return fitness;
   }
   public double optimum(){ return size; }
   public LiOResourceDefinition getDefinition(){
       LiOResourceDefinition def = new LiOResourceDefinition("lio.LiOTask",
            "problems.bitchain.OneMax", "Implements the One Max problem.");
       def.addMember("size", "Size of the problem");
       def.setKindOfIndividuals("lio.individuals.BitChain");
       return def;
   }
   public int getSize(){ return size; }
   public void setSize(int pSize){ size=pSize; }
}
```

Figure 7.4: Code for the binary function OneMax

Chapter 8

Programming metaheuristics

As it has been seen in the previous chapters, LiO can be considered as a set of resources with a defined behaviour and interfaces that can be used to instantiate the existing search algorithms or build new ones.

This chapter shows how to build search algorithms by using these resources in two circumstances: When the algorithms obey to a particular use and are not integrated in LiO and when they are wanted to be a part of the library.

8.1 The class LiOEnv

The class LiOEnv has some items that are decared with `static` modifier, so that they can be accessed anywhere in the library. It can be seen in figure 8.1.

This class has four members:

- `errorLog`: this object, of type `LiOErrorLog`, is used as an interface for the system of notification messages. Any object can use it to send message about errors or warnings.
- `task`: is the problem to be solved.
- `statistics`: this object holds all statistical information about algorithm execution. It has some methods to access this statistics and to provide it information about execution status.
- `random`: is the random numbers generator for the whole library, so we can set a fixed seed in order to make repeatable executions.

The last two methods are related with `task` object. With `problemSize` any resource can know individuals size defined in the problem. `setTask` is used to set which problem is to be solved.

```
public abstract class LiOEnv {

  public static LiOErrorLog errorLog = new LiOPrintStreamErrorLog();

  public static LiOTask task = null;

  public static Statistics statistics = new Statistics();

  public static Random random = new Random();

  public static double random(){
      return random.nextDouble();
  }

  public static int problemSize() {
      return task.defineIndividuals().getSize();
  }

  public static void setTask(String pTask) {
      task = (LiOTask) LiOResourceFactory.getHandle().createResource(
              "lio.LiOTask", pTask);
  }
}
```

Figure 8.1: The class LiOEnv.

8.2 Using the LiO classes from the outside

The first case of use of LiO arises when it becomes necessary to program a new algorithm with some specific purpose but there is no need to integrate it on the library. In this sense, LiO can be seen as a repository which provides pieces of code that can be used without any restriction. Next, we show how to use it with an example.

Imagine that we want to program the pseudo-code in figure 8.2. It only generates an individual and mutates it until there is no improvement in 10 consecutive mutations. And let's suppose that we have wanted to solve the function in `problems.contchain.OneMax`, which takes this form:

$$f(x) = x_1 + \ldots + x_n, n = 100$$

where x_n is a real number in $[0, 1]$.

First of all, we must know the kind of data we are working with so that to determine the available resources. In this case, as mentioned above, the problem will be codified on a vector of real values, so we must use the class `lio.individual.ContChain` to represent the individuals.

Next step consists of determining which resources are needed to build the algorithm. In this case, we only need two:

- A *generator* to create an initial solution.

```
testSearchAlgorithm(){
   solution=GenerateSolution();
   best=fitness(solution);
   noImprove=0;
   while (noImprove<10){
       newSolution=mutate(solution);
       candidateFitness=fitness(newSolution);
       if (candidateFitness>mejor){
            solution=newSolution;
            best=candidateFitness;
            noImprove=0;
       }
       else
            noImprove++;
   }
}
```

Figure 8.2: Pseudo-code of an example search algorithm.

- A *mutation* to mutate the individuals.

The first one must be chosen among all resources which implements the interface `lio.generators.Generator` and works with chains of continuous values, for instance, `lio.generators.contchain.RandomGenerator`. In the case of the mutation, the resources must implement the interface `lio.mutation.Mutation`. Among all the available operators we have chosen `lio.contchain.MinMaxMutation`

With these two objects we can already build the algorithm without even using the class LiOEnv. In this case, we would get the evaluation value by accessing the evaluate function of the task directly. This would be done as follows:

```
value = task.evaluate(Individual)
```

However, in order to make things easier, we can also take advantage of the members of the class LiOEnv. Thus, we can use the member `value()` of the individuals as well as the `statistics` member.

The declarations for the search algorithm can be seen in figure 8.3. As we see, we have only had to declare the two resources described, and a couple of individuals for the current and new solutions.

Figure 8.4 shows the constructor. As can be seen, the first thing to do when working with the LiO class is instantiating the task used, in this case, to perform evaluations. This way, statistics will be computed automatically. Other interesting thing to point out is the way that information concerning to the task

```
import lio.LiOEnv;
import lio.individuals.ContChain;
import lio.generators.contchain.RandomGenerator;
import lio.contchain.MinMaxMutation;
import problems.contchain.OneMax;
public class testSearchAlgorithm{
   private ContChain solution;
   private ContChain newSolution;
   private RandomGenerator generator;
   private MinMaxMutation mutation;
. . .
```

Figure 8.3: Declarations for the search algorithm

is passed to the resources. This is necessary, as seen in section 7.1 because both of them implement the interface `LiODependentResource`.

```
public testSearchAlgoritm(){
   LiO.task = new OneMax();
   generator = new RandomGenerator();
   mutation = new MinMaxMutation();
   generator.getTaskInformation(LiOTask);
   mutation.getTaskInformation(LiOTask);
}
```

Figure 8.4: Constructor of the search algorithm

Once all objects have been declared, it is time to build the search function. In the code of Figure 8.5 can be seen that there are only necessary a few lines of code to write it.

8.3 Programming generic metaheuristics

We have seen how to program an algorithm by using the resources in LiO. The integration of an algorithm in the library needs some additional steps. However, it could be interesting because of two main reasons:

- They can be executed by means of the interfaces provided by the library (GUI and command line).
- We can take advantage of the architecture of the library in order to create generic and instantiable algorithms.

```
public void search(){
  solution = generator.generate(1)[0];
  while (LiO.statistics.getNumEvalsWithoutImproving()<10){
      newSolution = (ContChain)solution.clone();
      mutation.mutate(newSolution);
      if (newSolution.value()>solution.value()
           solution=newSolution;
  }
  System.out.println("Result = "+LiO.statistics.getBestFitness());
}
```

Figure 8.5: Search function of the test algorithm

In order to know how to carry it out, it is necessary to understand the behaviour of LiO when executing an algorithm. This can be divided into 4 parts:

- Reading of the algorithm configuration. This will be provided either by a configuration file or the graphical user interface, and contains the chosen instances for the required resources and the values for the parameters. Moreover, the configuration also gathers information about stop conditions, output formatting, etc.
- Checking of the datatype consistency among the task, the resources and the own algorithm. As mentioned, there is a lot of resources or algorithms which work with a certain kind of data. LiO checks whether the datatypes of the specified resources are consistent. If the algorithm can be executed, it assigns default values for those resources and members which hasn't been specified on the configuration file.
- Once that LiO has automatically loaded a correct and concrete configuration, all the objects are built.
- Last, the algorithm is executed.

All this tasks are transparent to programmer and are implemented in the class `lio.search.LiOSearch`. All the classes which implement search algorithms must inherit from this one and so, must implement two abstracts functions:

- `public abstract boolean worksWith(LiOTask kindOfTask)`. There are some algorithms which are not generic, that is, can not work with every kind of data. For instance PBIL can only deal with binary problems. This function allows knowing if the algorithm is able to work with the data used by the task.

- `public abstract void run();` Which implements the main loop of the algorithm.

There are also some members of `LiOSearch` that are instantiable (search is also a resource) and can be useful for the programmer:

- `public static SearchOutput searchOutput`. It yields an interface with the `LiO.statistics` object that allows defining the format of the output. This class implements the interface `LiOResource` so that it can be adapted to the different kind of statistics or it can show them in a different way.
- `public static StopCondition stopCondition`. This object configures the stop conditions of the algorithms, so it is also connected with the object `LiO.statistics`. This class is also a resource, so it can be extended if there must be implemented an stop condition not usual as, for instance, the total distance run by ants when solving an instance of the tsp.

There is also a function, `protected boolean stopCondition()` that returns true depending on the `stopCondition` seen above or whether there has been some indication for the main thread to stop.

Last, it provides a static function that allows performing all the necessary operations to build an algorithm and running it in a different thread:
`public static boolean execute(LiOSearch algorithm, String[] options);`

8.3.1 An example

In order to give a clearer view of the algorithms construction, this section shows how to implement an algorithm so that it can be integrated in LiO. Furthermore, the algorithm will be generic, that is, it will be able to work with every kind of data.

Firstly, let's on the declarations. As can be seen in figure 8.6, since we don't know the kind of data necessary to solve the task, we use the abstract class `Individual`. The same way, the generator of individuals and the mutation operator are not instantiated with a certain class, but they are declared with the interfaces which define the functionality.

As mentioned in the section above, the concrete classes and objects will be generated by automatically by LiO depending on the configuration file.

Next, figure 8.7 shows the implementation of the function `worksWith`. Since there are no restrictions in this case, it is not necessary checking the task and, therefore, function always return true.

Since a search algorithm is also a resource, it becomes necessary to define it. In figure 8.8 we see such definition. As can be seen, it implements the abstract class `LiOSearch`. And it has three members, which are declared the same way regardless whether they are resources or not.

```
import lio.LiOEnv;
import lio.individual.Individual;
import lio.generator.Generator;
import lio.mutation.Mutation;
public class genTestSearchAlgorithm{
   private Individual solution;
   private Individual newSolution;
   private Generator generator;
   private Mutation mutation;
   private double prMutation;
```

Figure 8.6: Declarations for the generic version of the search algorithm.

```
public boolean worksWith(LiOTask kindOfTask){
   return true;
}
```

Figure 8.7: Method `worksWith` return always true because this algorithm can deal with any task.

```
public LiOResourceDefinition getDefinition{
   LiOResourceDefinition def;
   def = new LiOResourceDefinition("lio.LiOSearch",
       "lio.search.genTestSearchAlgorithm");
   def.setDescription("Example algorithm");
   def.addMember("prMutation","Probability of Mutation");
   def.addMember("generator","Generator of solutions");
   def.addMember("mutation","Mutation Operator");
   return def;
}
```

Figure 8.8: Declaration for the `getDefinition` method, showing algorithm's members, description, name and type.

Next, figure 8.9 shows the definition of the method implementing the search loop. As can be seen, it is very similar to the shown in the previous chapter. It is worth noticing that the resources are compatibles with the kind of data so, despite the work is done with individuals, resources perform a cast.

The last function in the loop, `cleanExecution()`, notifies to LiO the end of the search algorithm.

```
public void run(){
   solucion = generador.generate(1)[0]
   while (!stopCondition()) {
       statistics.newIteration();
       newSolution = (Individual) solution.clone();
       mutation.mutate(newSolution,prMutation);
       if (newSolution.value()>solution.value())
            solution = newSolution
       }
       cleanExecution();
}
```

Figure 8.9: Method for the search process of this algorithm.

Last, the only thing left is the main method so that this algorithm can be executed. In figure 8.10 is shown how to implement this method. Only to lines are needed in all cases, one to instantiate an object for the algorithm, and another in which `execute` method are used to begin execution.

```
public static void main(String[] args) {

  genTestSearchAlgorithm alg = new genTestSearchAlgorithm();
  execute(alg, args);

}
```

Figure 8.10: Main method for executing this algorithm.

Chapter 9

Internal Configuration of LiO

LiO has a file where are registered all resources included on it, grouped by the data type used. This list of resources is used in the graphical interface to offer us the choices in configuration process. Beside, when a search algorithm is built and all its resources must to be loaded, if a particular resource has not been declared either by configuration file or by command line default values have to be used. This default values are what resource must be used for a particular data type, and are in that file too.

```
@lio.individuals.ContChain
    & lio.probdistributions.ProbDistribution
        * lio.probdistributions.contchain.MarginalProbabilityVector

    & lio.crossover.Crossover
        * lio.crossover.contchain.ArithmeticalCrossover
        - lio.crossover.contchain.SimpleCrossover
        - lio.crossover.contchain.LinearCrossover
        - lio.crossover.contchain.BLXAlphaCrossover
        - lio.crossover.contchain.DiscreteCrossover
        - lio.crossover.contchain.ExtendedIntermediateCrossover
        - lio.crossover.contchain.ExtendedLineCrossover
        - lio.crossover.contchain.FlatCrossover
        - lio.crossover.contchain.WrightsHeuristicCrossover

    & lio.mutation.Mutation
        * lio.mutation.contchain.RandomMutation
        - lio.mutation.contchain.MinMaxMutation

     & lio.generators.Generator
        * lio.generators.contchain.RandomGenerator
```

Figure 9.1: Part of LiO internal configuration file.

In figure 9.1 it can be seen a part of the LiO internal configuration file up to now. In this figure it is shown resources compatible with continuous chain individuals. Each data type is preceded by an 'at' symbol (@), each kind of resource must have an andpersand character (&), and for each concrete implementation of that resource has an asterisk (*) as prefix if it's the default resource or a hyphen (-) otherwise. Each element must be in a single line and the order is important, because for each resource read from that file LiO understand that it is of the last kind of resource read and for the last data type read. Indentation is only for clearlyness.

In LiO there are other resources that can be used for every data type. In this case, this resources are under the 'nondependent' category as it can be seen in figure 9.2.

```
@nondependent

    &lio.generators.GreedyConstructor
        * lio.generators.GreedyConstructor

    & lio.selectors.Selector
        * lio.selectors.RouleteWheelSelector
        - lio.selectors.KTournement

    & lio.replacement.Replacement
        * lio.replacement.SimpleElitistReplacement

    & lio.memetic.HillClimbing
        * lio.memetic.HillClimbing
    & lio.memetic.SimulAnnealing
        * lio.memetic.SimulAnnealing

    & lio.perturbation.Perturbation
        * lio.perturbation.MutationBasedPerturbation

    & lio.misc.SearchOutput
        * lio.misc.SearchOutput
    & lio.misc.StopCondition
        * lio.misc.StopCondition
```

Figure 9.2: List of resources compatible with all data types.

To end with this file, there is another section in which are all search algorithms and tasks included in LiO. This is only for LiO gui in order to make the lists to chose this objects. As it can be seen in figure 9.3, tasks have a percentage symbol (%) as prefix and search algorithms have a dollar sign ($).

```
%problems.bitchain.OneMax
%problems.bitchain.SixPeaks
%problems.bitchain.EqualProducts
%problems.bitchain.CheckerBoard
%problems.bitchain.MMDP
%problems.bitchain.Rastrigin
%problems.bitchain.Ackley
%problems.bitchain.Colville
%problems.bitchain.Griewangk
%problems.bitchain.Rosenbrock
%problems.bitchain.Schwefel
%problems.bitchain.Powell

%problems.contchain.OneMax
%problems.contchain.Colville
%problems.contchain.Ackley
%problems.contchain.Griewangk
%problems.contchain.Rastrigin
%problems.contchain.Rosenbrock
%problems.contchain.Schwefel
%problems.contchain.SumCan
%problems.contchain.Powell

%problems.permutation.PermutationExample
%problems.permutation.SymmetricTSP

$lio.search.genetic.CHC
$lio.search.genetic.StdGeneticAlgorithm
$lio.search.local.greedy.GRASP
$lio.search.local.greedy.GreedyConstruction
$lio.search.local.hillclimbing.ILS
$lio.search.local.hillclimbing.MRHillClimbing
$lio.search.local.hillclimbing.MRStochasticHillClimbing
$lio.search.local.simulatedannealing.SA
$lio.search.probabilistic.EDA
$lio.search.probabilistic.PBIL
```

Figure 9.3: List of tasks and algorithms registered in LiO.

Chapter 10

Custom data types

Apart from using data types included in LiO as was described in section 6, this library has the choice of building new ones quickly and easily. In spite of LiO will grow up in the future with new algorithms, resources and data types, it's very likely that for a particular situation an special data type was needed.

Assuming this need, in this chapter will be seen steps required to solve with LiO a new task that works with a data type that encoded an string. This task will be called `CustomTask` and its aim is find an individual that has the same value, an string, that those defined in this task. So the fitness function return the number of characters of the individual that coincide with the same character in the reference string.

First of all, it is needed have the individual class to represent potential solutions for this task. This class will be called `CustomIndividual` and it's shown in figure 10.1. Here there is nothing strange, this class has its constructors and its method `getElements` as the more important things, and also methods for updating and querying some position and for cloning itself.

Next, for every data type is needed a class descendant from `LiOBounds` which give some useful information about each data type. This class for this individual will be `CustomBounds` and is depicted in figure 10.2. In this case, this class is the simplest because only has a constructor with the individual size. It's supposed that this classes are in the `custom` package.

And finally, it's necessary to implement the task class. This class is very simple too, as it can be seen in figure 10.3. `CustomTask`, as it is called, in its constructor defines the string that it has to be searched. Obviously, the optimum value is the lenght of this string and the `evaluate` method only check every string positon one by one.

Now, all basic things necessary to use this new individual are ready. It can be thought to use genetic algorithm seen in section 8.2 or the one seen in section 8.3.1. This algorithms can deal with any data type, but in both cases

```
public class CustomIndividual extends Individual {
  private char[] contenido;

  public CustomIndividual(char[] aux) {
      for (int i = 0; i < size; i++) {
          contenido[i] = aux[i];
      }
  }

  public CustomIndividual() {
      contenido = new char[size];
  }

  public char get(int i) {
      return contenido[i];
  }

  public void set(char letra, int pos) {
      contenido[pos] = letra;
  }

  public char[] getElements() {
      return contenido;
  }

  public String toString() {
      return new String(contenido);
  }

  public Object clone() {
      return new CustomIndividual(this.contenido);
  }
}
```

Figure 10.1: Custom individual.

```
public class CustomBounds extends LiOBounds {

  public CustomBounds(int size) {
      this.size = size;
  }
}
```

Figure 10.2: Bounds for the custom individual.

```
public class CustomTask extends LiOTask {
  private char[] referencia;

  public CustomTask() {
      String aux = new String("En un lugar de la mancha...");
      referencia = aux.toCharArray();
  }

  public LiOBounds defineIndividuals() {
      LiOBounds lbounds = new CustomBounds(referencia.length);
      return lbounds;
  }

  public double optimum() {
      return referencia.length;
  }

  public double evaluate(Individual individual) {
      double result = 0;
      char[] aux;
      aux = ((CustomIndividual) individual).getElements();
      for (int i = 0; i < referencia.length; i++)
          if (referencia[i] == aux[i])
              result = result + 1;
      return result;
  }

  public LiOResourceDefinition getDefinition() {
      LiOResourceDefinition def = new LiOResourceDefinition(
              "custom.CustomTask", "lio.LiOTask");
      def.setKindOfIndividuals("custom.CustomIndividual");
      return def;
  }

}
```

Figure 10.3: Task that works with this custom individual.

are necessary the generator and mutation operator for this data type. They can be seen in figures 10.4 and 10.5.

```
public class CustomGenerator implements Generator, LiODependentResource {

  private int size;

  public void getTaskInformation(LiOTask task) {
      setSize(task.defineIndividuals().getSize());
  }

  public void setSize(int pSize) {
      size = pSize;
  }

  public Individual[] generate(int nIndividuals) {
      CustomIndividual[] newIndividuals = new CustomIndividual[nIndividuals];
      char[] aux = new char[size];

      for (int i = 0; i < nIndividuals; i++) {
          for (int j = 0; j < size; j++)
              aux[j] = (char) (Math.random() * 255);
          newIndividuals[i] = new CustomIndividual(aux);
      }
      return newIndividuals;
  }

  public LiOResourceDefinition getDefinition() {
      LiOResourceDefinition def = new LiOResourceDefinition(
              "lio.generators.Generator", "custom.CustomGenerator",
              "Generates CustomIndividuals randomly.");
      def.setKindOfIndividuals("custom.CustomIndividual");
      return def;

  }

}
```

Figure 10.4: Random generator for `CustomIndividual`.

```
public class CustomMutation implements Mutation {

  public void mutate(Individual ind, double probability) {
      for (int i = 0; i < ind.getSize(); i++) {
          if (Math.random() < probability) {
              ((CustomIndividual) ind).set((char) (Math.random() * 255), i);
          }
      }
  }

  public void mutate(Individual individual) {
      int pos = (int) (Math.random() * individual.getSize());
      ((CustomIndividual) individual).set((char) (Math.random() * 255), pos);
  }

  public LiOResourceDefinition getDefinition() {
      LiOResourceDefinition def = new LiOResourceDefinition(
              "lio.mutation.Mutation", "custom.CustomMutation",
              "Implements a custom example mutation.");
      def.setKindOfIndividuals("custom.CustomIndividual");
      return def;
  }
}
```

Figure 10.5: Mutation operator for `CustomIndividual`.

Part III

Algorithms in LiO

Chapter 11

Introduction

One of the advantages of using Java is the way that software can be documented by using javadoc functionality. In this sense, the javadoc obtained from the source code of LiO contains all the information concerning to the resources. Not only is this information related with the point of view of programming, but also with the function of each one.

Although the search algorithms are also resources, and thus, they are described in the javadoc, in this chapter we briefly describe those that have been included in the library. In order to do that, we briefly explain the way that each one of them performs as well as the parameters and resources it needs to fix in order to run.

Chapter 12

Genetic Algorithms

12.1 Standard Genetic Algorithm

The class `lio.search.StdGeneticAlgorithm` implements the well known standard genetic algorithm.

The parameters of this search algorithm are:

- `populationSize`: Size of the population.
- `probCrossover`: Probability of crossover.
- `probMutation`: Probability of mutation.

This class needs to instantiate 5 resources. As it is generic, all of them are declared with the name of the generic interfaces:

- `generator`: It must be chosen among those which implement the interface `Generator` and generates the initial population, therefore, it depends on the task.
- `selector`: This object is used to select some individuals from a population by taking into account their fitness. It deals with individuals, so it doesn't depend on the kind of data. Selectors are defined by the interface `Selector`.
- `crossover`: This reference must contain the crossover operator, which depends on the task. Crossovers implement the interface `Crossover`.
- `mutation`: Points to the mutation operator, which is task dependent and implements the interface `Mutation`.
- `replacer`: Chooses a subset of individuals between two populations in order to generate a new one.

The main loop of this algorithm is shown in figure 12.1.

```
population <- generate population of size S

while not stopcondition

     newpopulation <- select S elements from population

     apply_crossover to newPopulation

     apply_mutation to newPopulation

     population <- select S from elements from population and newpopulation
```

Figure 12.1: Standard genetic algorithm.

12.2 CHC

The algorithm CHC is implemented in the class `lio.search.genetic.CHC`. It works only with binary problems and, opposite to the standard genetic algorithm, most of the resources it uses are previously defined. Thus, this algorithm always uses a crossover operator called `HUX`, a binary mutation (`BinaryMutation`) and the elitist selection scheme (`ElitistSelection`).

It only has two parameters:

- `populationSize`: Size of the population.
- `r`: Percentage of mutating positions.

And only one resource:

- `generator`: Generates the initial population. It must implement `Generator`.

Its pseudo-code can be seen in figure 12.2

```
L = size of the individuals
D = L / 4
population <- generate population of size S
while not stopcondition
    newpopulation <- select S elements from population in random order
    form S/2 couples
    for each couple
        if hamming distance is bigger than D
            swap half of the bits at random
        otherwise
            delete the elements
    newpopulation <- select S from elements from population and newpopulation
    if population=newpopulation
        d--
    if d<0
        for elements except the best one flip r*L positions at random
        d = r * (1 - r) * L
```

Figure 12.2: Pseudo-code for the CHC algorithm

Chapter 13

Estimation of Distribution Algorithms

13.1 Generic EDAs

Estimation of Distribution Algorithms are a family of evolutionary algorithms characterized for learning a probability model from the best individuals in a population and sampling the new population from this model.

Although there are many approaches (*UMDA, MIMIC, TREE, EBNA, HBOA*) most of them share the main scheme of functioning and they only differ on the probabilistic model used to learn the features of the population. Thus, they all are implemented in LiO by one class, `lio.search.probabilistic.EDA`.

One of the main advantages of EDAs is the few number of parameters they have to fix. In this case, there is only one:

- `populationSize`: Size of the population.

Concerning to the resources these are the ones which needs the algorithm to run:

- `generator`: Generates the initial population. It must implement `Generator` and depends on the kind of data.
- `probDistribution`: It is the probability distribution learnt from the models and used to sample the new ones. It is dependent on the kind of data.
- `replacer`: Chooses a subset of individuals between two populations in order to generate a new one.

It's worth remarking again that we can change the EDA we are using by only changing the `probDistribution` resource so, the pseudo-code for all EDAs is the same and it is shown in figure 13.1.

```
population <- generate population of size S

while not stopcondition

     learn a probabilistic model from the S/2 best individuals in the population

     new population <- sample S elements from the model

     population <- select S from elements from population and newpopulation
```

Figure 13.1: Pseudo-code for a generic EDA

13.2 PBIL

PBIL algorithm is defined only for binary problems. It works slightly different than rest of EDAs and therefore, it is implemented apart. All this resources are previously defined so they don't have to be instantiated. However, this algorithm requires to fix 5 parameters:

- `populationSize`: Size of the population.
- `learningRate`: Learning rate.
- `negativeLearningRate`: Whether to use or not negative learning and its rate.
- `mutationProbability`: Probability of mutation of the probability vector.
- `mutationShift`: Amount of mutation that affects probability vector.

It's pseudo-code can be seen in figure 13.2

```
initialize vector V.

while not stopcondition

     population <- generate from V

     update V towards best solution with learningrate

     update V away the worst solution with negativelearningrate

     mutate with probability prob by using mutationshift
```

Figure 13.2: Pseudo-code for the PBIL algorithm.

Chapter 14

Local Search Algorithms

14.1 Multiple Restart Hill Climbing

This algorithm generates several solutions and perform a local search in each one of them.

There is only a parameter:

- `startingPoings`: Number of solutions generated.

And two resources:

- `generator`: Generates the initial population. It must implement `Generator` and depends on the kind of data.
- `hillClimbing`: It is the operator that perform the hill climbing given a solution. This operator is generic and can be applied to all kind of individuals. However, one of its resources `neighbourhood` is used to define the neighbourhood of a given solution and thus, it depends on its encoding.

The pseudo-code is given in figure

```
generate s solutions

for each solution

   perform a hillclimbing
```

Figure 14.1: Pseudo-code of the multiple restart hill climbing

14.2 Iterated Local Search

This algorithm seems like the previous one, but instead of choosing several random solutions it only generates one and, when the hillclimbing is done, next solution is generated as a perturbation of the current one.

Again, the algorithm has got only one parameter:

- `startingPoings`: Number of solutions generated.

Whereas it has three resources:

- `generator`: Generates the initial population. It must implement `Generator` and depends on the kind of data.
- `hillClimbing`: It is the operator that perform the hill climbing given a solution.
- `perturbation`: This operator takes a solution and makes a perturbation of it so as to generate another individual.

The algorithm works as it is shown in figure in 14.2.

```
solution <- generate solution

while not stop condition

solution <- hillclimbing of the solution

solution <- perturbate the obtained solution
```

Figure 14.2: Pseudo-code of the iterated local search algorithm

14.3 GRASP

This algorithm is different from the rest in the sense that it requires special information from the task. The *Greedy Randomized Adaptive Search Procedure*, GRASP, belongs to the family of greedy algorithms. They don't part from a solution, but build iteratively it. For instance, in the TSP problem, a greedy algorithm would start in one city and would take the nearest city until it completes a tour.

It becomes necessary for a task to be solved with GRASP offering the mechanisms that allow doing it. Therefore, they must extend `LiOTask`. This is done by the class `LiOGreedyTask`. It basically consists on three functions:

- `generateStartingPoint`: That generates a starting point for a solution. In the TSP it would choose the origin city.
- `getExtensionsToSolution`: It would generate extensions to the solutions. For TSP, it would return the nearest solutions.
- `isComplete`: It determines whether it is a complete solution or not.

Figure 14.3 shows the pseudo-code of the GRASP:

```
while not stopcondition

   solution <- generate solution

   solution <- greedyconstruction

   solution <- hillclimbing of the solution
```

Figure 14.3: Pseudo-code of the GRASP algorithm

This algorithm has only one parameter:

- `numberOfIterations`: Number of solutions generated or iterations.

Whereas these are the resources:

- `greedyConstructor`: Instantiates the `GreedyConstructor` resource with the given task.
- `hillClimbing`: It is the operator that perform the hill climbing given a solution. This operator is generic and can be applied to all kind of individuals. However, one of its resources `neighbourhood` is used to define the neighbourhood of a given solution and thus, it depends on its encoding.

14.4 Simulated Annealing

This algorithm does a local search from a single individual, but allow choosing some "bad solutions" to avoid local optimum prematurely. This strategy is based on thermodynamic, so some temperature values are used to control how this bad solutions are selected. At the beginning, a given temperature is setted an in each iteration this value is decreased to reduce probability of going to solutions very different that current one. There are four parameters to fix:

- `initialTemp`: Initial temperature.
- `finalTemp`: Final temperature.

- `fatorA`: constant value by which temperature is decreased.
- `maxIntIter`: Number of internal iterations.

And there are two resources, besides:

- `generator`: Generates the initial solution. It must implement `Generator` and depends on the kind of data.
- `simulatedAnnealing`: Performs the selection of next solution in the current individual neighbourhood, using a temperature value as mean to, stochastically, allow movements towards bad solutions.

```
solution <- generate solution

currentTemp <- initialTemp

while currentTemp > finalTemp

   for 1 to maxIntIter

      solution <- select neighbour by simulated annealing with currentTemp

   update currentT by factorA
```

Figure 14.4: Pseudo-code of the Simulated Annealing algorithm

Chapter 15

Particle Swarm Optimization

In this kind of algorithms, each particle represents a solution that moves across the search space according to the information from both itself and its neirbough-hood.

Although the algorithm was originally proposed to work with real coded problems, there are some adaptations to other representations. Thus, despite for the time being only problems represented by `ContChain` can be solved with LiO, the code is open to incorporate the variations.

The psoudocode of a particle swarm algorithm is shown in 15.1

```
swarm <- generate solutions

while not stop condition

   for 1 to num particles

      calculate new position of the particle according to the information.

 reestructure neighbourhood if necessary
```

Figure 15.1: Pseudo-code for a Particle Swam Optimization algorithm

The generic algorithm only takes 3 parameters:

- `numParticles`: Number of particles that compose the swarm.
- `neighbourhood`: Defines the structure of the neighbourhood of each particle.
- `generator`: Generates the particles.

The parameter `neighbourhood` must be a resource implementing `Neighbourhood`. It associates, to each particle, the best position reached by another one. This information is used for the particle to update the velocity vectors. Two neighbourhood structures are currently defined: `GlobalNeighbourhood`, which makes all particles using the best position achieved so far, and `RingNeighbourhood`, that defines several neighbours for a particle at the beginning of the process and then incorparates the best position achieved by one of them to it.

In other hand, the particles and their behaviours must be defined. A `Particle` always have a reference to the best position reached so far, to some other particle (on its neighbourhood) and a method to calculate its next point. LiO implements `StdContParticle` that represents the velocity with a vector of real numbers and updates it according with the state of the art techniques.

The `generator` parameter must be instantiated to some object whose class implements `SwarmGenerator`. This objects generate particles of a certain kind. In LiO, it is implemented `StdContParticleGenerator`, that generates the kind of particles described above with the parameters specified.