Programming metaheuristics with LiO

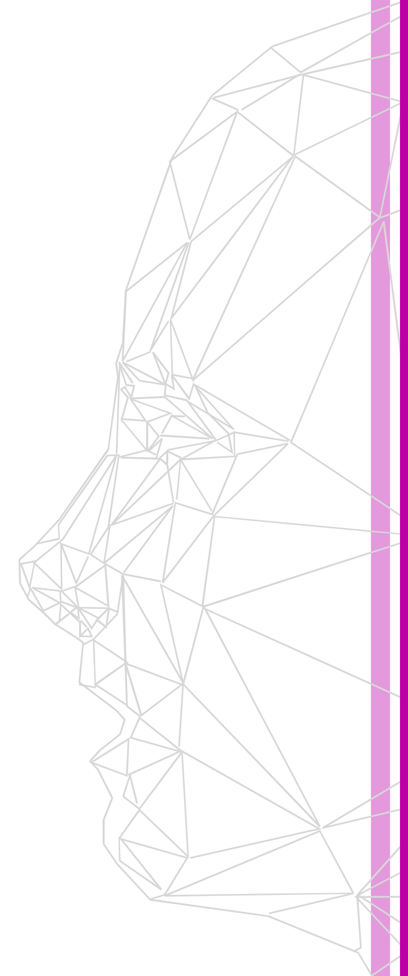
Juan L. Mateo and Luis de la Ossa

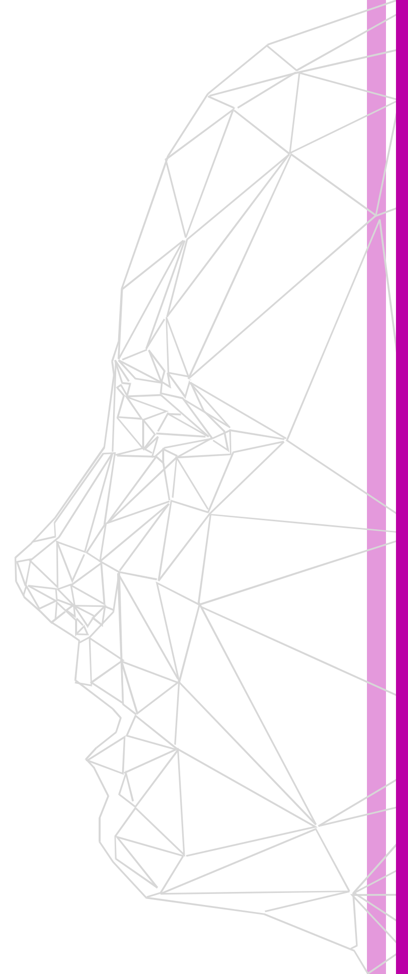
Intelligent Systems and Datamining Group
Computing Systems Department – I3A
University of Castilla-La Mancha
Spain

S.I.M.D.

Programming metaheuristics with `LiO`

- Introduction to `LiO`:
  - Individuals and data types.
  - Resorces.
- Creating new tasks.
- Implementation of operators.
- Search algorithms in `LiO`.
- Custom data types
- Internal functioning of `LiO`
- Using `LiO` from outside.
- Some useful hints





Introduction to `LiO`

Individuals and data types

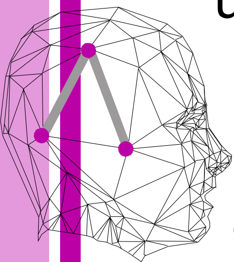
In `LiO`, solutions of problemas are codified in classes extending `lio.individuals.Individual.`

The most important method in an individual is `value()`, which returns a `double` resulting from evaluating the solution.

Currently, 3 kind of individuals are defined in `LiO`:

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

However, most algorithms can deal with datatypes defined by the user.



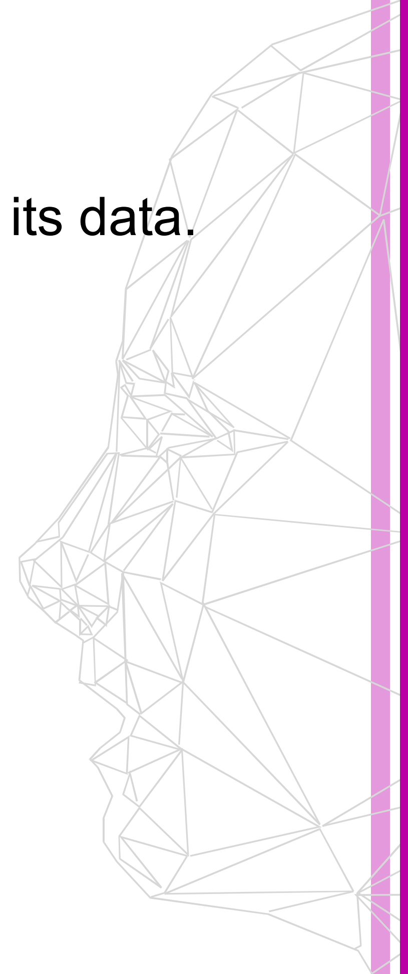
Individuals and data types

Each one of the classes provides methods to access its data.

Thus, for the pre-defined data typed we have:

```
BitChain bCh = new BitChain(100);
int[] elements = bCh.getElements();

ContChain cCh = new ContChain(100);
double[] elements = cCh.getElements();

Permutation pm = new Permutation(100);
int[] elements = pm.getElements();
```

See lio.individuals.Individual.java, lio.individuals.ContChain, etc.



Individuals and data types

There are operators which can only deal with a certain kind of representation:

I.e, mutations, crossovers, etc.

However, there are some other, as `RouletteWheelSelector`, which work with all data types.

It is quite important pointing out that operators used in the set up of an algorithm must be compatible with the kind of individual used to code the solutions.

Resources

In LiO, almost everything (tasks, operators, search algorithms, stop conditions) is defined as a *resource*.

Technically, it means that almost everything implements the interface

lio.core.**LiOResource**

In this interface, only a function is declared:

```
public LiOResourceDefinition getDefinition();
```

Resources

Next code shows the implementation of getDefinition() for an operator which implements the *arithmetical crossover* for real vectors (ContChain).

```
public LiOResourceDefinition getDefinition() {
    LiOResourceDefinition def = new LiOResourceDefinition(
            "lio.crossover.Crossover",
            "lio.crossover.contchain.ArithmeticalCrossover",
            "lio.individuals.ContChain",
            "Implements the aritmethical crossover.");
    def.addMember("alpha", "Alpha constant. (0.35)");
    return def;
}
```

Resources

The constructor takes four parameters:

1) The interface or class implemented or extended by the object which defines its functionality.

2) The name of the class implementing the resource.

3) The kind of individuals the resource is designed to work with.

4) A description of the resource.

Resources

Moreover, a line must be added for each parameter of the resource that is going to be configured either in the GUI or in the configuration file:

```
def.addMember("alpha", "Alpha constant. (0.35)");
```

It is composed by the name of the parameter and one description.

The class must declare a member called `alpha`.

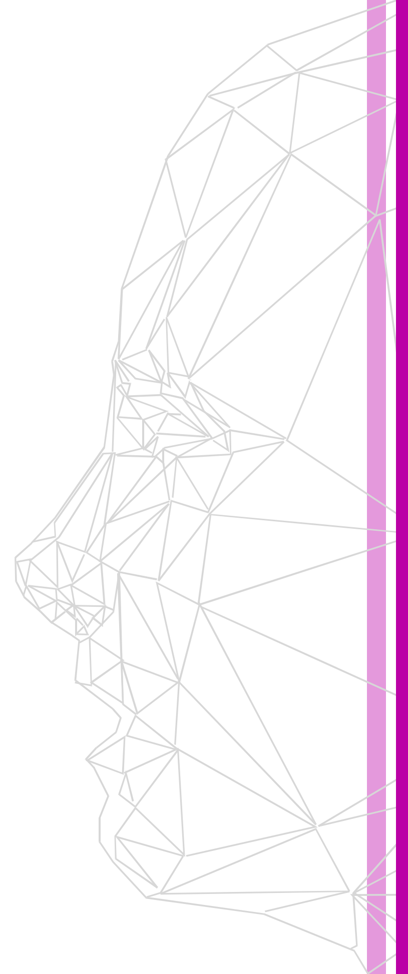
Resources

The last thing about resouces is that, for each parameter, there must be two functions defined in order to be able to automatically access to it.

```
public double getAlpha() { return alpha;}

public void setAlpha(double alpha) { this.alpha = alpha;}
```

The convention used with the names of the functions is the one from JavaBeans, consisting in prefixing with get and set the name of the variable.

See lio.crossover.contchain.ArithmeticalCrossover.java



Creating new tasks

LiOTask

Classes implementing tasks, extends the class `LiOTask`:

```
public abstract class LiOTask implements LiOResource {

    public abstract LiOBounds defineIndividuals();
    public abstract double evaluate(Individual individual);
    public double optimum() { }
    public LiOResourceDefinition getDefinition() { }
}
```

LiOTask

- The first function, `getDefinition()`, has been described and it is necessary for both defining the kind of individuals and configurating the resource.

- Since we work with maximization problem, the optimum is fixed to infinity. If the optimum of a task is known, the function must be overrided.

LiOTask

• The third function must be implemented. It is necessary to specify the ranges of the individuals.

```
public LiOBounds defineIndividuals() {
     ContChainBounds bounds = new ContChainBounds(size, 0, 1);
     return bounds;
}
```

In this case, we use a `ContChainBounds` object (which extends `LiOBounds`), necessary to define size and ranges ([0,1]) for the variables in a chain of real numbers. `LiO` also implements `BitChainBounds` and `PermutationBounds`

See lio.individuals.LiOBounds and lio.individuals.ContChainBounds

LiOTask

Last, the function `evaluate()` evaluates an individual and returns the value of the solution represented with it. The example shows the evaluation in OneMax for continuous problems.

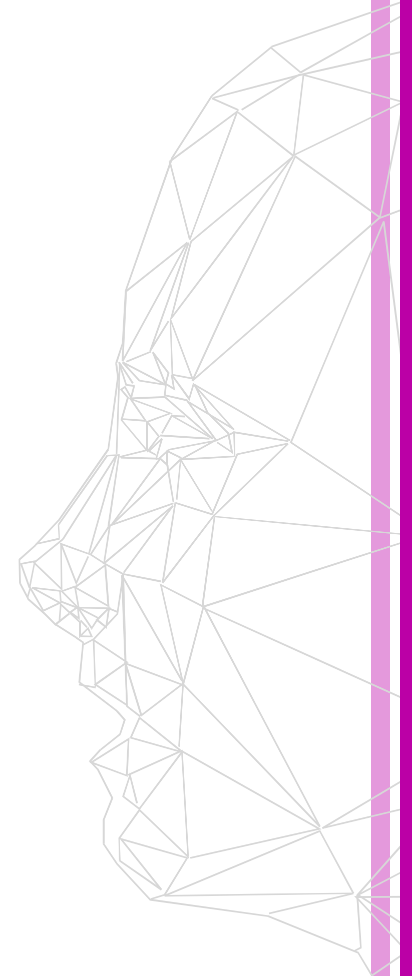
```
public double evaluate(Individual individual) {
    double fitness = 0;
    double[] elements = ((ContChain)individual).getElements();
    for (int i=0;i<elements.length;i++)
        fitness=fitness+elements[i];
    return fitness;
}
```

Notice that it is necessary to make a cast to the specific kind of individual used by the task!

See lio.LiOTask and problems.contchain.OneMax

LiOTask

Once a task is implemented by the user, it can be selected by using "*custom*" in the menú, or declare the whole name of the class in the configuration file.



Implementation of operators

Implementation of operators

As mentioned, an operator is also a *Resource*, thus, it must implement the interface `LiOResource` described above.

For instance, in the ArithmeticalCrossover, the `getDefinition()` function looks as follows:

```
public LiOResourceDefinition getDefinition() {
    LiOResourceDefinition def = new LiOResourceDefinition(
            "lio.crossover.Crossover",
            "lio.crossover.contchain.ArithmeticalCrossover",
            "lio.individuals.ContChain",
            "Implements the aritmethical crossover.");
    def.addMember("alpha", "Alpha constant. (0.35)");
    return def;
}
```

Moreover, the `get` and `set` methods must be declared for each parameter, in this case, for the member `alpha`.

See lio.individuals.LiOBounds and lio.individuals.ContChainBounds

S.I.M.D.

Implementation of operators

Besides the common interface `LiOResource`, each operator has a certain functionality. In the case considered, as it is shown in the resource definition this is given by the interface:

lio.crossover.**Crossover**

```
public interface Crossover extends LiOResource {

    /**
     * @return The size of the offspring produced by the operator.
     */
    public int getSizeOffSpring();
    /**
     * Crosses two individuals.
     * @param parent1 First parent
     * @param parent2 Second parent
     * @return An array of individuals containing the offspring.
     */
    public Individual[] cross(Individual parent1, Individual parent2);
}
```

Implementation of operators

As it can be seen, the interface is generic, that is, doesn't depend on the kind of data, and neither the class implementing it. Thus, a cast must be done in the implementation.

```
public Individual[] cross(Individual parent1, Individual parent2) {
    size = parent1.getSize();
    // Gets the elements.
    double[] p1Elements = ((ContChain) parent1).getElements();
    double[] p2Elements = ((ContChain) parent2).getElements();
    // Crosses them
    double h1, h2;
    for (int i = 0; i < size; i++) {
        h1 = alpha * p1Elements[i] + (1 - alpha) * p2Elements[i];
        h2 = alpha * p2Elements[i] + (1 - alpha) * p1Elements[i];
        p2Elements[i] = h2;
        p1Elements[i] = h1;
    }
    ContChain ind1, ind2;
    ind1 = new ContChain(p1Elements);
    ind2 = new ContChain(p2Elements);
    Individual[] ind = { ind1, ind2 };
    return ind;
}
```

Implementation of operators

An operator designed by the user, as it happened with tasks, can be used with the search algorithms of the library by clicking the "custom" option in the GUI or giving the full name in the configuration file.

If the operator is not compatible with the kind of data necessary to represent the task, then a warning will be generated

S.I.M.D.

Search algorithms in `LiO`

S.I.M.D.

LiOEnv

This class provides some statics members that must be accesed from several parts of LiO such as (algorithms, individuals, etc.)

The most important are shown below:

```
public abstract class LiOEnv {

    /** This object is used to manage error and exception messages */
    public static LiOErrorLog errorLog = new LiOPrintStreamErrorLog();

    /** Task which is going to be solved */
    public static LiOTask task = null;

    /** Object that contains and computes statistics of the search. */
    public static Statistics statistics = new Statistics();
```

Search algorithms in `LiO`

The search algorithms implemented in LiO allow the configuration of some of their components.

They can work with several data types always that resources required are defined for them.

The execution process of an algorithm can be decomposed in 4 steps:

- Read of the algorithm configuration, either from GUI or from a configuration file.
- Consistency checking among data types used in tasks, resources, and the algorithm itself. Assignament of default resources for non specified parameters.
- Objects instantiation
- Algorithm execution

S.I.M.D.

Search algorithms in `LiO`

All these actions are transparent even for the programmer!!

..and are implemented in the class `lio.search.LiOSearch`.

```
public abstract class LiOSearch implements LiOResource, Runnable {
```

All search algorithms must inherit this class, composed by two abstract functions that must be implemented:

```
public abstract boolean worksWith(LiOTask kindOfTask);

public abstract void run();
```

Search algorithms in `LiO`

The first function, `worksWith()`, is to use the compatibility of the task with the search algorithm: For instance, a Greedy based algorithm can only process `LiOGreedyTask` tasks.

The second, `run()`, implements the main cycle of the search.

Search algorithms in `LiO`

Next, some members and methods of `LiOSearch` that can be usefull for the programmer are shown.

- `public static SearchOutput searchOutput`:

  Provides an interface with the statistics `LiOEnv.statistic` that allows selecting which data are shown and how.

  The class `Statistics` is a resource, thus, it can be extended and configured so that some information non computed by default can be processed. For instance, since steps necessary to build a valid solution with a Greedy algorithm which are not evaluations of whole solutions, it should be accounted apart.

  `SearchOutput` also implements `LiOResource` in order to adapt to the different kind of statistics. Thus, a GreedySearchOutput is used to show the statistics in GreedyStatistics.

See lio.search.local.greedy.GreedySearchOutput

Search algorithms in `LiO`

`public static StopCondition stopCondition:`

This object determines the stop conditions of the algorithm according to `LiOEnv.statistics.`

Thus, it is also a resource that can be configured to adapt to particular algorithms and statistics.

`LiOSearch` implements the function:

```
protected boolean stopCondition()
```

It returns the value depending on the `stopCondition` object or an external interruption.

S.I.M.D.

Search algorithms in `LiO`

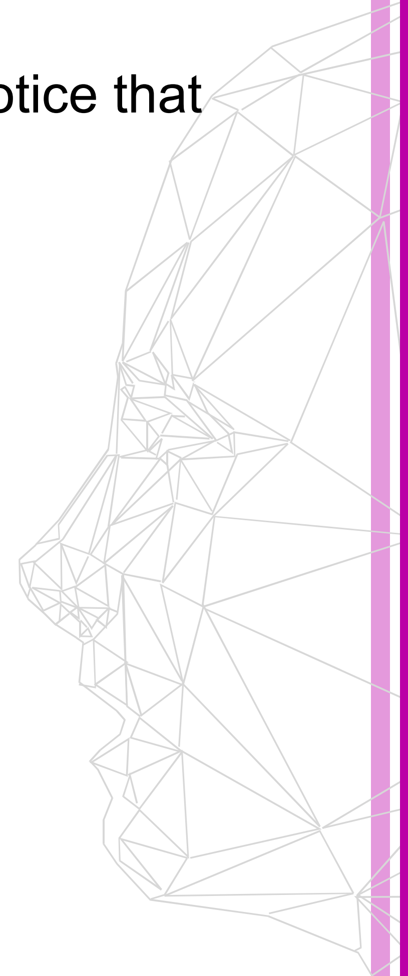
Last, `LiOSearch` provides an static method to carry out all the tasks necessary to build an algorithm and run it in an independent thread.

```
public static boolean execute(LiOSearch algorithm, String[] options) {...}
```

Search algorithms in `LiO`

Next, the code of the StdGeneticAlgorith is shown, notice that declarations use generic interfaces or abstract classes.

```
public class StdGeneticAlgorithm extends LiOSearch {
    // Parameters of the search.
    /** Size of the population */
    private int populationSize;
    /** Probability of crossover */
    private double probCrossover;
    /** Probability of mutation */
    private double probMutation;

    // Resources used by the search.
    /** Generator of individuals */
    private Generator generator;
    /** To select individuals */
    private Selector selector;
    /* Mutation operator */
    private Mutation mutation;
    /* Crossover operator */
    private Crossover crossover;
    /** Replacement operator */
    private Replacement replacer;

    // Other private members.
    /** Population of individuals */
    private StaticPopulation population;
    /** Intermediate population */
    private StaticPopulation intPopulation;
```



S.I.M.D.

Search algorithms in LiO

The constructor is simple since resources are configured outside the class and depending on the kind of data.

```
/** Creates a new instance of StdGeneticAlgorithm */
public StdGeneticAlgorithm() {
    // Some default values for the members.
    populationSize = 200;
    probCrossover = 0.6;
    probMutation = 0.05;
}
```

Since the Genetic Algorithm can a priori work with every kind of tasks:

```
// Inherited from LiOSearch
public boolean worksWith(LiOTask kindOfTask) {
    // It works with all kind of data and tasks
    return true;
}
```

Search algorithms in LiO

As we are working with resources:

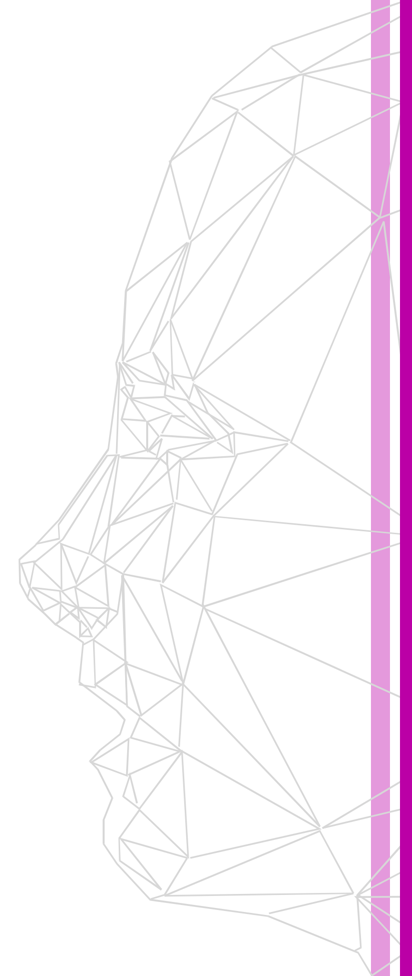
```
public LiOResourceDefinition getDefinition() {
    LiOResourceDefinition def;
    def = new LiOResourceDefinition("lio.LiOSearch",
            "lio.search.genetic.StdGeneticAlgorithm");
    def.setDescription("Implements the standard genetic algorithm");
    def.addMember("populationSize", "Size of the population.");
    def.addMember("probMutation", "Probability of mutation.");
    def.addMember("probCrossover", "Probability of crossover.");
    def.addMember("generator", "Generates the initial population.");
    def.addMember("selector","Selects individuals from a population in order to cross them.");
    def.addMember("crossover", "Operator of crossover.");
    def.addMember("mutation", "Operator of mutation.");
    def.addMember("replacer","Generates a new population by keeping some of the individuals in the former one.");
    return def;
}
```

We must declare the methods `getGenerator()`, `setGenerator()`, etc.

Search algorithms in `LiO`

The main bucle is as follows:

```
// Implementation of the search algorithm.
public void run() {
    init();
    while (true) {
        selectBestIndividuals();
        applyCrossover();
        applyMutation();
        if (stopCondition())
            break;
        replacePopulation();
        LiOEnv.statistics.newGeneration();
    }
    cleanExecution();
}
```



Search algorithms in LiO

Notice that functions work with Individuals:

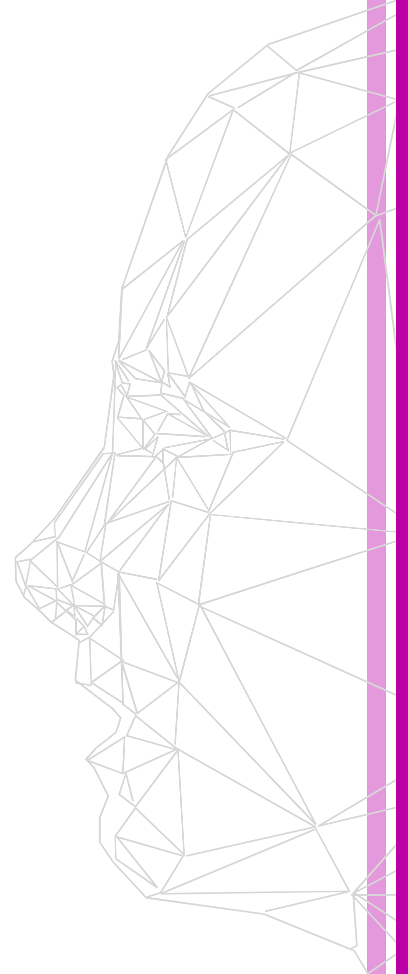
```
/** Mutate individuals. */
private void applyMutation() {
    for (int i = 0; i < populationSize; i++)
        mutation.mutate(intPopulation.elementAt(i), probMutation);
}

/** Generates new generation from both the current and the former one. */
private void replacePopulation() {
    population = new StaticPopulation(replacer.replace(intPopulation, population));
}
```

Search algorithms in `LiO`

Last, the main function allows executing the algorithm:

```
public static void main(String[] args) {
    StdGeneticAlgorithm ga = new StdGeneticAlgorithm();
    execute(ga, args);
}
```

Custom Data Types



Custom data types

LiO search algorithms allow working with data types other than BitChain, ContChain, or Permutation.

Thus, a new datatype can be created with the only condition that it extends the class Individual

```
public class CustomDataType extends Individual {
    private double[][] elements;

    public CustomDataType(double[][] pElements){
        elements = pElements;
    }

    //......
```

Custom data types

Resources necessary to perform the search with this kind of individual must be also implemented.

`getDefinition()` would take this form:

```
public LiOResourceDefinition getDefinition() {
    LiOResourceDefinition def = new LiOResourceDefinition(
            "lio.crossover.Crossover",
            "external.custom.CustomCrossover",
            "external.custom.CustomDataType",
            "Implements a Crossover for CustomDataType.");
    return def;
}
```

Custom data types

In the case of custom datatypes, there are no default values for the resources used in an algorithm.

Thus, a whole configuration file must be specified to run the algorithms either from the GUI or the command line.

Internal functioning of LiO

Internal functioning of LiO

As mentioned above, one of the main tasks that must be carried out before the algorithm is executed consists of checkint data type consiscency. This is done trough their definitiones.

`LiO` also contains a configuration file where all resources are registered, besides the data type they can work with.

This file is used to select default resources when no instantiations of them have been specified.

Moreover, the file is also used for integrating the resources in the GUI menus.

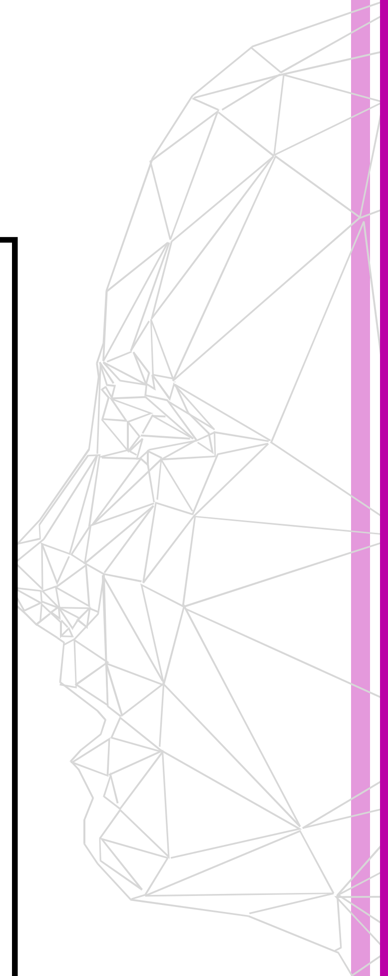
See lio.core.LiO.conf

S.I.M.D.

Internal functioning of LiO

The file gathers resources available for each data type

```
@lio.individuals.ContChain

    & 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
```

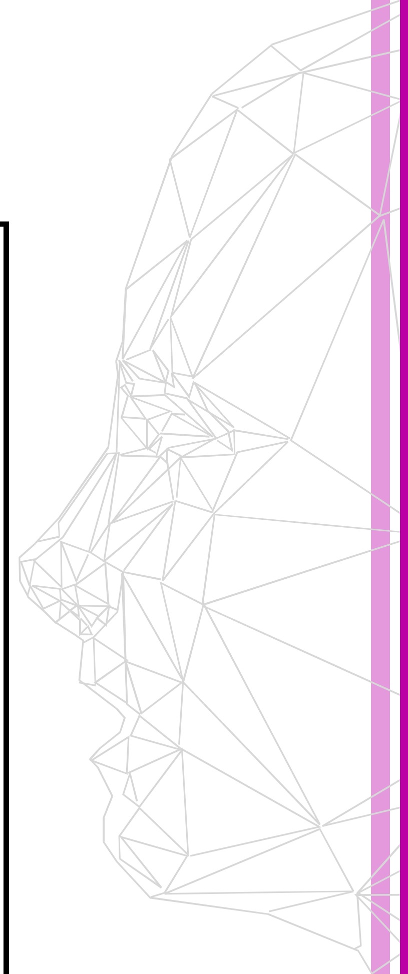


S.I.M.D.

Internal functioning of LiO

Thus, some operators are generic.

```
@nondependent

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

        & lio.selectors.Selector
                * lio.selectors.RouleteWheelSelector

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

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

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

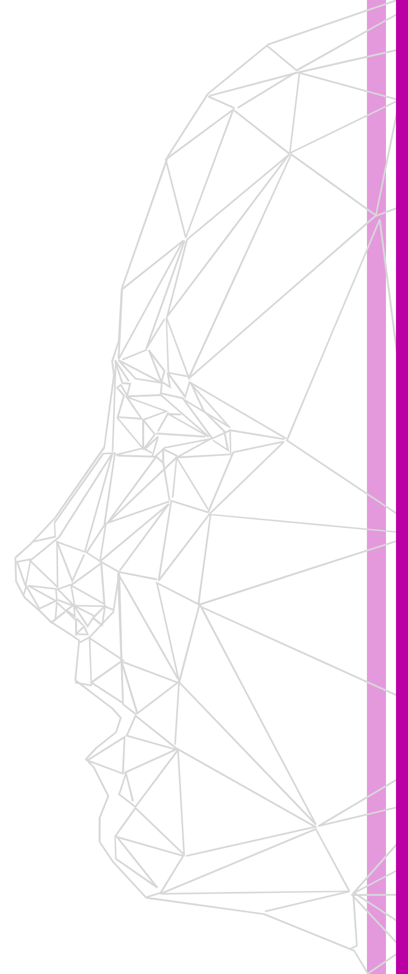


S.I.M.D.

Internal functioning of LiO

The file also keeps a register with tasks and algotithms in order to show them on the menus:

```
%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.simulatedannealing.SA
$lio.search.probabilistic.EDA
$lio.search.probabilistic.PBIL
$lio.search.pso.PSO
```

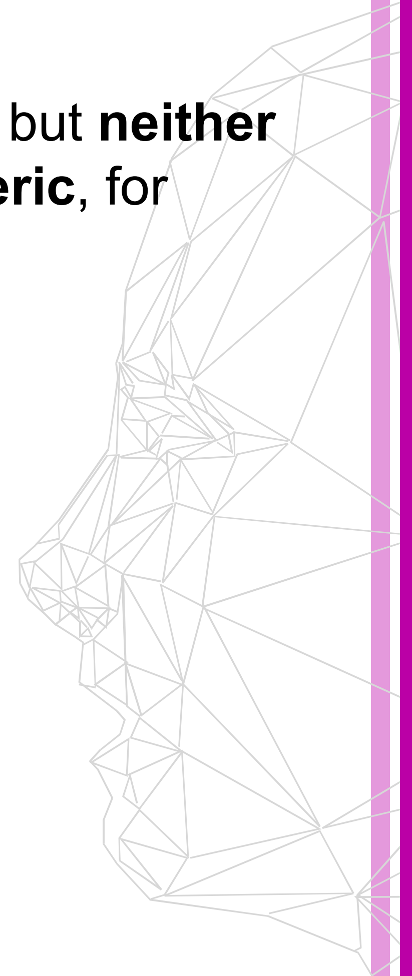


Using LiO from outside

Using LiO from outside

Lets imagine that we want to program some algorithm but **neither want it to be integrated in the library, nor to be generic**, for instance, this dummy algorithm:

```
solucion = generateSolucion()
best = fitness(solucion);
noImproves = 0;

while (noImproves<10) do
        newSolution = Mutation(solution);
        fNS = fitness(newSolution);
        if fNS>best
                best=fNS;
                noImproves=0;
                solution = newSolution;
        if not
                noImproves++;
```



Using LiO from outside

Lets also imagine that we want to solve this evaluation function:

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

Which is already implemented (although it doesn't have to):

```
problems.continuous.OneMaxCont
```

And codifies solutions in objects:

```
lio.individuals.ContChain
```

Using LiO from outside

First, we need to know which objects we need:

An object which generates individuals (ContChain)

```
lio.generators.contchain.RandomGenerator
```

An object to mutate individuals

```
lio.mutation.contchain.MinMaxMutation
```

S.I.M.D.

Using LiO from outside

Let's show the declarations necessary to implement this algorithm

```
import lio.individuals.ContChain;
import lio.generators.contchain.RandomGenerator;
import lio.mutation.contchain.MinMaxMutation;
import problems.contchain.OneMax;

public class Tutorial1 {

    private ContChain solution;
    private ContChain newSolution;
    private RandomGenerator generator;
    private MinMaxMutation mutation;
    private OneMax task;

    private double bestValue;
    private double fitnessNewSolution;

    private int evalsWithoutImproving = 0;
```

S.I.M.D.

Using LiO from outside

Next, constructor is shown:

```
public Tutorial1(){
    // Creates objects
    task = new OneMax();
    generator = new RandomGenerator();
    mutation = new MinMaxMutation();
    // Some objets need information about the task.
    generator.getTaskInformation(task);
    mutation.getTaskInformation(task);
}
```

Using LiO from outside

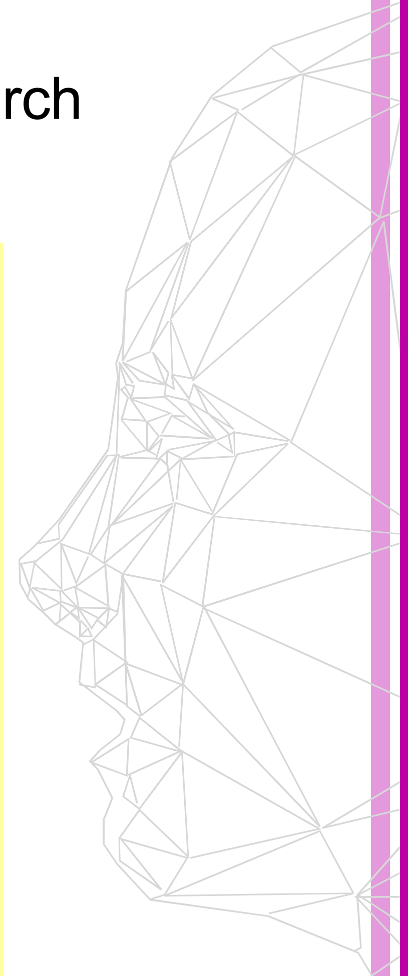
Last, we show the search function:

```
public void search(){
    solution = (ContChain)generator.generate(1)[0];
    bestValue = task.evaluate(solution);
    while (evalsWithoutImproving<10){
        newSolution = (ContChain)solution.clone();
        mutation.mutate(newSolution);
        fitnessNewSolution = task.evaluate(newSolution);
        if (fitnessNewSolution>bestValue){
            evalsWithoutImproving=0;
            solution = newSolution;
            bestValue = fitnessNewSolution;
        }
        else{
            evalsWithoutImproving++;
        }
    }
    System.out.println(bestValue);
}
```

Using LiO from outside

We can also use the class `LiOEnv`. Then, the search algorithm described could be written as follows:

```
import lio.LiOEnv;
import lio.individuals.ContChain;
import lio.generators.contchain.RandomGenerator;
import lio.mutation.contchain.MinMaxMutation;
import problems.contchain.OneMax;

public class Tutorial2 {

    private ContChain solution;
    private ContChain newSolution;
    private RandomGenerator generator;
    private MinMaxMutation mutation;

    public Tutorial2(){
        // Creates objects
        LiOEnv.task = new OneMax();
        generator = new RandomGenerator();
        mutation = new MinMaxMutation();
        // Some objets need information about the task.
        generator.getTaskInformation(LiOEnv.task);
        mutation.getTaskInformation(LiOEnv.task);
    }
```

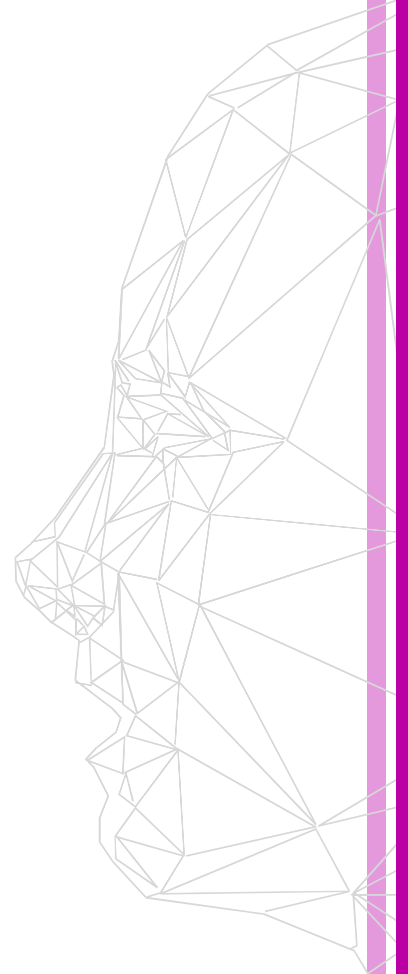


Using LiO from outside

And the main cycle is reduced to:

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

Some useful hints



Some usefull hints

Individuals

It is necessary to take a look at the `value()` method:

```
public double value() {
    if (!(evaluated)) {
        value = LiOEnv.task.evaluate(this);
        evaluated = true;
        // Notifies the evaluation to the statistics register.
        if (LiOEnv.statistics != null)
            LiOEnv.statistics.newEvaluation(this, value);
    }
    return value;
}
```

As it can be seen, it only makes a real evaluation of the individual if it hasn't been evaluated yet.

Individuals

Thus, if some change is carried out in the individual by some operator like a mutation, it must be evaluated again.

The change is notified as follows:

```
public void change() {
    evaluated = false;
}
```

In the class inheriting Individual implemented in LiO, calls to `change()` are made automatically whenever a change on the elements is done.

FIN