Knowledge Discovery and Intelligent Systems

JCLEC 4 Tutorial
**CONTENTS**

1 Introduction 5
   1.1 What is JCLEC? .................................................. 5
   1.2 Overview ......................................................... 6
   1.3 Class hierarchy ............................................... 7

2 Genetic algorithms using binary encoding: Knapsack problem 9
   2.1 Introduction ..................................................... 9
   2.2 Problem definition ............................................ 9
   2.3 Genetic algorithm ............................................. 10
   2.4 Implementation .............................................. 12
   2.5 Results ......................................................... 16

3 Genetic algorithms using integer encoding: Traveling Salesman Problem 21
   3.1 Introduction ..................................................... 21
   3.2 Problem definition ............................................ 21
   3.3 Genetic algorithm ............................................. 23
   3.4 Implementation .............................................. 26
   3.5 Results ......................................................... 30

4 Genetic algorithms using real encoding: Optimization of real functions 35
4.1 Introduction .................................................. 35
4.2 Problem definition ........................................... 35
4.3 Genetic algorithms ............................................ 35
  4.3.1 Common configuration .................................... 36
  4.3.2 Generational genetic algorithm ......................... 37
  4.3.3 Steady state genetic algorithm ......................... 37
  4.3.4 Multimodal algorithms .................................. 38
4.4 Implementation .............................................. 38
4.5 Results ..................................................... 42
  4.5.1 Generational genetic algorithm ......................... 42
  4.5.2 Steady state genetic algorithm ......................... 45
  4.5.3 Multimodal algorithms .................................. 46
  4.5.4 Global results .......................................... 49
  4.5.5 Generational vs Steady State ......................... 52
  4.5.6 Sharing vs Clearing .................................... 53
  4.5.7 General comparison .................................... 53
4.6 Study of effect of the alpha value in BLX-\(\alpha\) crossover operator .......... 54

5 Genetic programming: a symbolic regression problem 55
  5.1 Introduction ................................................ 55
  5.2 Problem definition ......................................... 55
  5.3 Implementation ............................................ 56
  5.4 Results .................................................... 63
1. INTRODUCTION

1.1 What is JCLEC?

JCLEC is a software system for Evolutionary Computation (EC) research, developed in the Java programming language. It provides a high-level software environment to do any kind of Evolutionary Algorithm (EA), with support for genetic algorithms (binary, integer and real encoding), genetic programming (Koza style, strongly typed, and grammar based) and evolutionary programming.

JCLEC architecture follows strong principles of object oriented programming, where abstractions are represented by loosely coupled objects and where it is common and easy to reuse code. JCLEC provides an EC environment that its main features are:

- **Generic.** With JCLEC, the user can execute any kind of EC, as far as it fulfills some minimum requirements. The only necessary condition is to have a population of individuals to which a sequence of evolving operations is iteratively applied. So far, JCLEC supports most mainstream EC flavors such genetic programming, bit string, integer-valued vector and real-valued vector genetic algorithms, and evolution strategy. It also includes support for advanced EC techniques such as multiobjective optimization. The user can take any of these specialized frameworks and modify them further to create a customized evolutionary algorithm.

- **User friendly.** Considerable efforts were deployed to make the use of JCLEC as easy and pleasant as possible. JCLEC possesses several mechanisms that offer a user friendly programming interface. The programming style promoted is high-level and allows rapid prototyping of applications.

- **Portable.** The JCLEC system has been coded in the Java programming language that ensures its portability between all platforms that implement a Java Virtual Machine.

- **Efficient.** To ensure efficient execution, particular attention was given to optimization of critical code sections. Detailed execution profiles of these sections were done.

- **Robust.** Verification and validation statements are embedded into the code to ensure that operations are valid and to report problems to the user.
• *Elegant.* The interface of JCLEC was developed with care. Great energy was invested in designing a coherent software package that follows good object oriented and generic programming principles. Moreover, strict programming rules were enforced to make the code easy to read, understand and, eventually, modify. The use of XML as file format is also a central aspect of JCLEC, which provides a common ground for tools development to analyze and generate files, and to integrate the framework with other systems.

• *Open Source.* The source code of JCLEC is free and available under the GNU General Public License (GPL). Thus, it can be distributed and modified without any fee.

### 1.2 Overview

Two layers comprise the JCLEC architecture represented in Figure 1.1:

The core system is the lower layer. It has the definition of the abstract types, its base implementations and some software modules that provide all the functionality to the system. Over the core layer it is the experiments runner system in which a job is a sequence of evolutionary algorithms executions defined by means of a configuration file. It receives as input this file and it returns as result one or several reports about the algorithms executions. It allows solving a problem more easily using the available EAs from a specific interface. It configures the algorithm, executes it in an interactive way and it also generates the on-line information about the evolutionary process. The system can be also used to define execution sequences that will be executed by the batch-jobs runner module. And the user can include his/her own code whenever the developed code fulfills the hierarchy defined in the system core.
1.3 Class hierarchy

Figure 1.2 shows the interfaces that define the functionality of the JCLEC system. As it can be seen, three types of interfaces exist: The ones related with the evolutionary system (IIndividual, IFitness, ISpecies, IEvaluator and ISystem), the ones that represent actions accomplished during evolution (IProvider, ISelector, IRecombinator and IMutator) and the evolutionary algorithm itself (IAlgorithm).

![Diagram of class hierarchy]

Figure 1.2: JCLEC core class hierarchy.

The IIndividual interface represents one of the individuals that lives in a system in evolution. This interface does not declare methods related to the individual’s genotype neither to its phenotype, so that the functionality that is defined in the lower classes of the hierarchy is granted. In fact, JCLEC’s core contains several implementations of the IIndividual interface that can be distinguished by the genotype encoding (bit string, int or double arrays, expression trees, etc). Such classes can be either used directly in the implementation of evolutionary algorithms or they can be extended by the user (defining the phenotype that is mapped from a given genotype). Obviously, the user can also define new types of individuals associated to new encoding schemes. As a matter of fact, if the class implements the IIndividual interface, then the integration with the other system components is taken for granted.

All the IIndividual instances are linked to an object that implements the interface IFitness, which denotes the individual’s fitness. There are several implementations for
this interface that represent fitness in mono-objective and multi-objective problems.

As aforementioned, in every EA there is a system that represents the evolutionary process. The ISystem interface is in charge of representing this system in an abstract way. This interface has a reference to an ISpecies object that represents the structural characteristics of the individuals and to an IEvaluator object that evaluates individuals. The ISpecies class has a different implementation for each type of individual in JCLEC, because it represents the structural features of a set of individuals (e.g., in case of binary individuals it stores the size of the chromosome and the schema that represents them, or for GP it stores the maximum size of the tree, the terminal and function nodes). This information is used by genetic operators in order to handle individuals correctly. The ISpecies class also works as a factory of IIndividual objects and it provides methods to create instances of the class they represent. On the other hand, IEvaluator performs the evaluation of the individuals, that is, it computes and assigns their fitness values. In order to solve a problem with JCLEC it is necessary that the user codes a class that implements this interface, providing the system a way for obtaining the individuals’ fitness.

The abstract representation of EAs is responsible of the IAlgorithm interface. This class has a reference to the system in order to evolve it as well as some references to ITool objects. Such objects represent the actions performed in the course of an evolutionary algorithm. There are several interfaces that extend ITool. IProvider is responsible of providing new individuals to the system (it is used in the algorithm initialization state). ISelector represents a selection procedure. The IRecombinator and IMutator interfaces represent the crossover and mutator genetic operators, respectively. Many of these components can be used in any EC paradigm, while others are designed to deal only with a certain type of individual. The user is responsible for choosing correctly these components during the design of a specific application. In addition, the IAlgorithm interface has three methods related to the EA execution: doInit():void takes charge of initializing the algorithm, doIterate():void takes charge of performing an iteration, and isFinished():boolean takes charge of notifying about the finalization of the execution.

Finally, in order to obtain information about the execution of an evolutionary process we have defined a listener system similar to the one used in the management of Java events. This system consists of the IAlgorithmListener interface and the AlgorithmEvent class. IAlgorithmListener objects take charge of picking up all the events related to the algorithm execution (algorithm started, iteration completed and algorithm finished), in order to react depending on the event fired. AlgorithmEvent represents events that happen during the algorithm execution. This class has a reference to the algorithm in order to access the current state and to react properly.
2. GENETIC ALGORITHMS USING BINARY ENCODING: KNAPSACK PROBLEM

2.1 Introduction

This section aims to introduce the use of JCLEC by analyzing how to solve a simple optimization problem using a simple genetic algorithm.

The problem considered is the knapsack problem. In this problem, given a set of items, each with a weight and a value, the objective is to determine the number of each item to include in the knapsack, so that the total weight is less than a given limit and the total value is as large as possible.

2.2 Problem definition

To solve the knapsack problem and in general any other problem in JCLEC, firstly it is necessary to define the problem that is intended to be solved, and then it is necessary to select one of the algorithms available in the library and set up its parameters.

Consider \( n \) distinct objects and a knapsack. Each object has a weight \( w_i \) and a price \( p_i \) associated for \( i = 1, 2, \ldots, n \). The knapsack can carry a weight not exceeding a given amount \( W \). Taking into account these considerations, the goal of the problem is to fill the knapsack in order to maximize the value of the objects contained. Notice that any object can be put in the knapsack if there is enough space for it, but the knapsack can not carry an object fraction. Therefore, to achieve the desired solution, for each object it is necessary to decide either it is put it in the knapsack or not. Each object has a variable \( x_i \) associated that takes the value 1 if the object is introduced in the knapsack, or 0 otherwise.

The problem can be formulated mathematically as shown below:

\[
\sum_{i=1}^{n} x_i \times v_i \text{ subject to } \sum_{i=1}^{n} x_i \times w_i \leq W
\]

With respect to the parameter configuration, the following restrictions to the knapsack
problem are considered: the maximum weight allowed is equal to 4200 grams, the number of articles per type can not exceed from 10 and there are 10 different kinds of objects.

2.3 Genetic algorithm

Genetic algorithms are adaptive methods that can be used to solve search and optimization problems. They are based on the genetic process of living organisms. Over generations, natural populations evolve according to the principles of natural selection and survival of the fittest, postulated by Darwin. In imitation of this process, genetic algorithms are capable of working towards solutions to real world problems. The evolution of these solutions to optimal values of the problem depends largely on an adequate coding of them.

A simple genetic algorithm performs the following sequence of operations. Firstly, an initial population of individuals is randomly generated. When the population is generated, a fitness value is calculated for each individual, so that individuals that adapt better to the problem will have a better fitness. The next step is to select those individuals who will act as parents in the next generation. Once they have been selected, genetic operations (crossover and mutation) are performed over them. If the stop criterion is satisfied, the algorithm shows the best result, if not, the evolutionary process continues.

To solve any problem, the genetic algorithm must be configured and coded according to our needs. In particular, the configuration of the algorithm to solve the knapsack problem will be described next.

- **Encoding criterion:** it is necessary to determine that an individual follows a binary encoding approach. In this scheme, the individual encodes a binary genotype, whose size is given by the number of different articles considered and the maximum number of articles per type (i.e., in our example the size is 10*10 = 100 genes), and where each gene corresponds to the object number in Table 2.1. Notice that a value of 1 in the $i$-th gene indicates that the $i$-th article has been put in the knapsack, while a value of 0 indicates the opposite. For example, the following genotype indicates that the individual includes the first, third, fourth and sixth objects in the knapsack:

  \[
  \begin{bmatrix}
    1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\
  \end{bmatrix}
  \]

- **Not feasible individual treatment:** this criterion specifies how individuals that do not fulfill the restrictions are going to be treated. This control can be done in two different stages of the algorithm, either at the creation and reproduction stage, avoiding to obtain invalid offsprings, or at the evaluation stage. The infeasible individuals are controlled at the evaluation stage, scoring negatively the fitness of
2.3 Genetic algorithm

<table>
<thead>
<tr>
<th>Object</th>
<th>Weight</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>150</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>325</td>
<td>40</td>
</tr>
<tr>
<td>3</td>
<td>600</td>
<td>50</td>
</tr>
<tr>
<td>4</td>
<td>805</td>
<td>36</td>
</tr>
<tr>
<td>5</td>
<td>430</td>
<td>25</td>
</tr>
<tr>
<td>6</td>
<td>1200</td>
<td>64</td>
</tr>
<tr>
<td>7</td>
<td>770</td>
<td>54</td>
</tr>
<tr>
<td>8</td>
<td>60</td>
<td>18</td>
</tr>
<tr>
<td>9</td>
<td>930</td>
<td>46</td>
</tr>
<tr>
<td>10</td>
<td>353</td>
<td>28</td>
</tr>
</tbody>
</table>

Table 2.1: Objects configuration.

this kind of individuals in such a way that they will not be included in future generations.

- **Initialization**: the population will be randomly initialized.

- **Evolution schema**: a generational elitism schema (that corresponds to SGE algorithm, i.e., the best individual found so far is included in the next generation of the algorithm) is considered.

- **Replacing criterion**: the offsprings automatically replace their parents. To preserve elitism, if the best solution from the previous generation does not survive, the worst solution is replaced by the new one.

- **Stop Criterion**: 1000 generations.

- **Fitness function**: the goal is to maximize the sum of the benefits of the objects included in the knapsack. This value can be calculated by adding the price value of the articles having an associated value of 1.

- **Criterion for selecting parents**: in order to apply genetic operators it is necessary to select a subset of the population. The tournament selector will be used and different tournament sizes will be tested.

- **Crossover operator**: a new individual based on the genotype of its parents is obtained. Several crossover operators which are implemented in the JCLEC library will be tested.

- **Mutation operator**: it modifies the genetic code of an individual, having each gene a probability of being mutated. Several mutation operators which are implemented in the JCLEC library will be tested.

- **Other parameters**: it will be analyzed how the population size affects the solution.
This section describes how to encode the configuration file required to run the algorithm in the library JCLEC and how to implement basic methods for evaluating and comparing individuals in the population.

- Firstly, the package `net.sf.jclec.algorithm` and a simple generational genetic algorithm are selected:

  ```
  <process algorithm-type="net.sf.jclec.algorithm.classic.SG">
  ```

- **Coding**: the class that represents the solution encoding has to be indicated. To use a binary representation, the JCLEC packages `net.jclec.binarray` and `BinArrayIndividualSpecies class` are selected. There is no more to implement because these individuals are directly valid for the representation that has been chosen. The size, the number of genes for each individual which depends on the number of items to be considered will be specified.

  ```
  <species type="net.sf.jclec.binarray.BinArrayIndividualSpecies" genotype-length="100"/>
  ```

- **Initialization**: the population of binary individuals is randomly initialized employing the class `net.sf.jclec.binarray.BinArrayCreator`.

  ```
  <provider type="net.sf.jclec.binarray.BinArrayCreator"/>
  ```

- **Stopping criterion**: it is set to the maximum number of generations.

  ```
  <max-of-generations>1000</max-of-generations>
  ```

- **Fitness function**: this step implements an evaluator appropriated to the characteristics that were determined to evaluate the population and to treat infeasible individuals. An example can be seen in `KnapsackEvaluator.java` partially shown below. This class will extend from `AbstractEvaluator`, located in the package `net.sf.jclec.base`. By extending it, those abstract methods will be implemented. The method `evaluate()` evaluates individuals given as argument and sets their fitness values.

  ```
  <evaluator type="tutorial.KnapsackEvaluator"/>
  ```

Below is the method `evaluate()` included in the file `KnapsackEvaluator.java` necessary for the evaluation of the individuals.
2.4 Implementation

```java
protected void evaluate(IIndividual ind)
{
    // Individual genotype
    byte[] genotype = ((BinArrayIndividual)ind).getGenotype();
    // Total weight and profit
    int totalWeight = 0, totalProfit = 0;
    // Calculate weight
    for (int i = 0; i < genotype.length; i++)
    {
        totalWeight += genotype[i] * weights.get(i);
        totalProfit += genotype[i] * profits.get(i);
    }
    // Calculate profit (if necessary)
    if (totalWeight <= maxWeight)
    {
        // Set individual fitness
        ind.setFitness(new SimpleValueFitness(totalProfit));
    }
    else
    {
        ind.setFitness(new SimpleValueFitness(-totalProfit));
    }
}
```

The method `getComparator()` sets and returns an object that establishes a relation order between the individuals fitness. This object is used to order individuals in different parts of the algorithm. `SimpleValueFitness` and `ValueFitnessComparator` classes are defined in the system, so the user only has to know their meaning and use them whenever it is needed:

```java
public Comparator<IFitness> getComparator()
{
    // Set fitness comparator (if necessary)
    if (COMPARATOR == null)
        COMPARATOR = new ValueFitnessComparator(!maximize);
    // Return comparator
    return COMPARATOR;
}
```

- **Selection criterion** of the parents can be set by using the `net.sf.jclec.selector` package. The tournament selector is implemented in this package.

```xml
<parents selector-type="net.sf.jclec.selector.TournamentSelector"/>
<tournament size="2"></tournament>
</parents>
```
**Genetic operators:** The package `net.sf.jclec.binarray` contains several mutation and crossover operators ready to use, which are described briefly below. Mutation operators `SeveralLociMutator` and `OneLocusMutator` perform the mutation of the genes of individuals of the population. `OneLocusMutator` selects only one gene at random while `SeveralLociMutator` operates on several genes. Finally, `UniformMutator` mutates the genes according to a given mutation probability value.

`OnePointCrossover` crossover operator performs the crossover of two individuals using a single point of intersection. In contrast, the `TwoPointsCrossover` operator performs the exchange of genes between two individuals using two points of intersection. `UniformCrossover` operator generates a random value (between 0 and 1) for each gene. If the value exceeds the locus crossover probability parameter, parent genes are exchanged in the offspring; otherwise, the genes are copied from the corresponding parent.

```
<recombinator type="net.sf.jclec.binarray.rec.UniformCrossover" rec-prob="0.75" />
<mutator type="net.sf.jclec.binarray.mut.OneLocusMutator" mut-prob="0.1" />
```

**Parameters:** with respect to other parameters, it is necessary to define the size of the population and the seed for the random number generator:

```
<rand-gen-factory type="net.sf.jclec.util.random.RanecuFactory" seed="987328938"/>
<population-size>100</population-size>
```

Finally, regardless which algorithm is used, the listener used to display the results must to be determined from the `net.sf.jclec.listener` package. In this example it is used a basic report generator that produces a file with the best, the worst and the average individual and the fitness mean and variance every 50 generations.

```
<listener type="net.sf.jclec.listener.PopulationReporter">
  <report-frequency>50</report-frequency>
</listener>
```

Below it is shown the whole XML configuration file that serves as the basis for the development of the tutorial.
The KnapsackEvaluator class is for general purposes and it can solve any knapsack problem. Before running the experiment, the weights and profits values must be set as well as the maximum value that the bag supports.

Once parameters are set in the the configuration file, the RunExperiment class can be run giving the configuration file as an argument. This method will run the genetic algorithm and will output a report with all the information requested. This information is usually the best individual, the worst individual found and the average fitness of individuals every 50 generations.
2.5 Results

Finally, analyzing the results obtained using the evaluator \textit{KnapsackEvaluator} when changing the combination of the crossover and mutation operators, leaving in this case the probability value constant (using the default values, the probability of crossover is 0.75 and the mutation probability is 0.1). For all the tests a population of 100 individuals has been used and the number of generations is 1000. Tables 2.2, 2.3 and 2.4 show the fitness obtained by combining the crossover (\textit{OnePointCrossover} and \textit{UniformCrossover}) with the mutation operators (\textit{OneLocusMutator}, \textit{SeveralLociMutator} and \textit{UniformMutator}).

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
\textbf{OneLocusMutator} & \textbf{SeveralLociMutator} \\
\hline
\textbf{OnePointCrossover} & \textbf{UniformCrossover} \\
\hline
Execution 1: 487.0 & Execution 1: 596.0 \\
Execution 2: 501.0 & Execution 2: 608.0 \\
Execution 3: 508.0 & Execution 3: 580.0 \\
Execution 4: 501.0 & Execution 4: 552.0 \\
Execution 5: 533.0 & Execution 5: 575.0 \\
Execution 6: 502.0 & Execution 6: 568.0 \\
Execution 7: 425.0 & Execution 7: 622.0 \\
Execution 8: 449.0 & Execution 8: 580.0 \\
Execution 9: 417.0 & Execution 9: 596.0 \\
Execution 10: 402.0 & Execution 10: 563.0 \\
\hline
Average: 472.5 & Average: 584 \\
\end{tabular}
\caption{OneLocusMutator results.}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
\textbf{OnePointCrossover} & \textbf{UniformCrossover} \\
\hline
Execution 1: 544.0 & Execution 1: 592.0 \\
Execution 2: 546.0 & Execution 2: 622.0 \\
Execution 3: 562.0 & Execution 3: 622.0 \\
Execution 4: 564.0 & Execution 4: 622.0 \\
Execution 5: 562.0 & Execution 5: 622.0 \\
Execution 6: 592.0 & Execution 6: 622.0 \\
Execution 7: 534.0 & Execution 7: 592.0 \\
Execution 8: 530.0 & Execution 8: 592.0 \\
Execution 9: 534.0 & Execution 9: 576.0 \\
Execution 10: 580.0 & Execution 10: 622.0 \\
\hline
Average: 554.8 & Average: 608.4 \\
\end{tabular}
\caption{SeveralLociMutator results.}
\end{table}
2.5 Results

<table>
<thead>
<tr>
<th>UniformMutator</th>
<th>OnePointCrossover</th>
<th>UniformCrossover</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution 1:</td>
<td>-543.0</td>
<td>548.0</td>
</tr>
<tr>
<td>Execution 2:</td>
<td>338.0</td>
<td>491.0</td>
</tr>
<tr>
<td>Execution 3:</td>
<td>354.0</td>
<td>520.0</td>
</tr>
<tr>
<td>Execution 4:</td>
<td>-292.0</td>
<td>487.0</td>
</tr>
<tr>
<td>Execution 5:</td>
<td>404.0</td>
<td>550.0</td>
</tr>
<tr>
<td>Execution 6:</td>
<td>-358.0</td>
<td>529.0</td>
</tr>
<tr>
<td>Execution 7:</td>
<td>250.0</td>
<td>513.0</td>
</tr>
<tr>
<td>Execution 8:</td>
<td>359.0</td>
<td>529.0</td>
</tr>
<tr>
<td>Execution 9:</td>
<td>389.0</td>
<td>554.0</td>
</tr>
<tr>
<td>Execution 10:</td>
<td>-295.0</td>
<td>534.0</td>
</tr>
<tr>
<td>Average:</td>
<td>60.6</td>
<td>525.5</td>
</tr>
</tbody>
</table>

Table 2.4: UniformMutator results.

As can be seen, the combination of SeveralLociMutator and UniformCrossover produces better individuals, obtaining an average fitness equal to 608.4. It is also interesting to see how the fitness evolves as the population size increases. These results are shown in Table 2.5 and Figure 2.1.

<table>
<thead>
<tr>
<th>P. Size</th>
<th>Exec. 1</th>
<th>Exec. 2</th>
<th>Exec. 3</th>
<th>Exec. 4</th>
<th>Exec. 5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>488</td>
<td>488</td>
<td>484</td>
<td>484</td>
<td>512</td>
<td>491.2</td>
</tr>
<tr>
<td>30</td>
<td>576</td>
<td>576</td>
<td>592</td>
<td>530</td>
<td>562</td>
<td>567.2</td>
</tr>
<tr>
<td>45</td>
<td>592</td>
<td>592</td>
<td>576</td>
<td>576</td>
<td>622</td>
<td>591.6</td>
</tr>
<tr>
<td>60</td>
<td>558</td>
<td>558</td>
<td>580</td>
<td>592</td>
<td>622</td>
<td>582</td>
</tr>
<tr>
<td>75</td>
<td>546</td>
<td>546</td>
<td>620</td>
<td>610</td>
<td>562</td>
<td>576.8</td>
</tr>
<tr>
<td>90</td>
<td>564</td>
<td>564</td>
<td>622</td>
<td>622</td>
<td>622</td>
<td>598.8</td>
</tr>
<tr>
<td>105</td>
<td>576</td>
<td>576</td>
<td>622</td>
<td>610</td>
<td>610</td>
<td>598.8</td>
</tr>
<tr>
<td>120</td>
<td>622</td>
<td>622</td>
<td>610</td>
<td>622</td>
<td>610</td>
<td>617.2</td>
</tr>
<tr>
<td>135</td>
<td>622</td>
<td>622</td>
<td>622</td>
<td>622</td>
<td>622</td>
<td>622</td>
</tr>
<tr>
<td>150</td>
<td>622</td>
<td>622</td>
<td>622</td>
<td>622</td>
<td>622</td>
<td>622</td>
</tr>
</tbody>
</table>

Table 2.5: Population size vs Performance.
As noted, the best result is obtained with a population size between 135 and 150 individuals. Generally, the fitness improves as the population size increases. Now, the mutation probability is set to 0.1 and the crossover probability varies. These results are shown in Table 2.6 and Figure 2.2.

<table>
<thead>
<tr>
<th>Probability</th>
<th>Exec. 1</th>
<th>Exec. 2</th>
<th>Exec. 3</th>
<th>Exec. 4</th>
<th>Exec. 5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>530</td>
<td>548</td>
<td>472</td>
<td>542</td>
<td>576</td>
<td>533.6</td>
</tr>
<tr>
<td>0.3</td>
<td>576</td>
<td>592</td>
<td>560</td>
<td>576</td>
<td>576</td>
<td>576</td>
</tr>
<tr>
<td>0.5</td>
<td>530</td>
<td>546</td>
<td>610</td>
<td>592</td>
<td>558</td>
<td>567.2</td>
</tr>
<tr>
<td>0.7</td>
<td>546</td>
<td>610</td>
<td>622</td>
<td>610</td>
<td>580</td>
<td>593.6</td>
</tr>
<tr>
<td>0.9</td>
<td>622</td>
<td>622</td>
<td>610</td>
<td>604</td>
<td>622</td>
<td>616.0</td>
</tr>
</tbody>
</table>

Table 2.6: Crossover probability results.
The best result is obtained with a crossover probability of 90%.

Now, the crossover probability is set to 0.9 and the mutation probability varies in order to study how this probability affects the results. As noted, best results are obtained with a mutation probability of 30%. These results are shown in Table 2.7 and Figure 2.3.

**Table 2.7: Mutation probability results.**
Figure 2.3: Mutation probability results.
3. GENETIC ALGORITHMS USING INTEGER ENCODING: TRAVELING SALESMAN PROBLEM

3.1 Introduction

The Traveling Salesman Problem (TSP) is a problem in combinatorial optimization and fits within known as transportation problems. The TSP has several applications for personal services, goods or other services (messaging, garbage collection, etc). The study of how to make these services costs profitable either minimizing consumption or obtaining the shortest time, make this an exploited area because of its applications. These problems generally require to consider simultaneously a number of restrictions, conditions and factors affecting the efficiency and the quality of the solutions provided. The variety of objectives, resources and constraints that often have real transportation problems make it very difficult to treat with exact optimization methods.

3.2 Problem definition

The Salesman Problem can be stated as follows: "A salesman must visit \( n \) cities, starting and finishing in his own city. Knowing the cost of going from one town to another, decide the path of minimal cost."

The solution to be decided, given a road map, is the order that the salesman must follow to visit \( n \) cities starting whichever, driving fewer miles and returning to the beginning after visiting each city only once. In this case, instead of having to select a subset of items, it is about choosing a permutation of \( n \) cities so that the distances traveled are minimal.

The Traveling Salesman Problem has the following properties:

- It is one of the most interesting which has arisen in Operations Research and for which extensive material has been published.
- It is very intuitive and easy to understand.
• Most of the techniques that have been appearing in the area of Combinatorial Optimization have been tested on it because its resolution is very complex.

When the theory of algorithmic complexity was stated, the TSP was one of the first problems to be studied, Karp proved in 1972 that it belongs to the class of NP-hard problems. Sahni and Gonzalez (1976) proved that, unless P = NP, there is no heuristic algorithm for the TSP to work in polynomial time on the size of the problem and is ε- Average.

This problem can be formally represented using a graph, where nodes represent cities, and edges the connection between these cities. It would be in addition a weighted graph, so each edge would have a value representing the euclidean distance from one city to another. The objective is to find a path in such a way that every city is visited only once, being the starting and the ending city the same. The Figure 3.1 shows a sample graph for a TSP problem with 7 cities:

Figure 3.1: 7 cities TSP graph.

To solve the problem it is supposed that the graph is complete, in which every pair of distinct vertices is connected by an edge, so that all cities will be connected with all others. The distance considered is the euclidean distance.

This tutorial will practise with 5 problems:

• berlin52: Number of cities: 52. Optimal solution cost: 7.542.
• d198: Number of cities: 198. Optimal solution cost: 15.780.
• lin318: Number of cities: 318. Optimal solution cost: 42.029.
3.3 Genetic algorithm

These instances have been obtained from TSPLIB library, a web site that collects various cases of this problem, along with the cost of its optimal solution or the best solution obtained so far. All files have the same format, a list of two values for each city that represent its coordinates in the map. To build the matrix of costs, it must be calculated the euclidean distance between each pair of cities.

To solve the problem, a genetic algorithm is used. This technique does not guarantee to achieve the optimal solution but an acceptable solution; furthermore, it requires less computational time and therefore it is more efficient.

3.3 Genetic algorithm

Two genetic algorithms will be employed, the simple generational elitist and the steady state. The former creates new offsprings from the individuals of an old population, by using the genetic operators, placing these individuals in a new population and always maintaining the best individual found so far in the new population. In contrast, the latter considers a continuous flow of individuals in the population, where new individuals are added while others are discarded. Hereafter, their main features are outlined:

- **Encoding criterion**: the traveling salesman problem is an order representation problem. This representation consist of as many genes as cities there are, and its values will be integer numbers. The chromosome will indicate the order in which to travel the graph, so that if there are have 5 cities (1,2,3,4,5), the code of the chromosome of an individual could be any permutation of that set, so a chromosome representation could be the following:

  \[
  \begin{array}{c}
  2 \\
  3 \\
  5 \\
  1 \\
  4 \\
  \end{array}
  \]

  which indicates that the cities will be visited in the following order: from city 2 to city 3, from city 3 to city 5, from city 5 to city 1, from city 1 to city 4 and finally as a cycle from city 4 to city 2.

- **Not feasible individual treatment**: it determines how to treat individuals that do not meet the restrictions. In this case, the control of invalid individuals will be performed at the creation and reproduction stages, blocking them to generate invalid offsprings.

  This way, the encoding used will allow us to generate all possible solutions. However, when using a representation order it is necessary to ensure that after applying any operator, the solution generated remains valid; otherwise, invalid solutions would be generated.
• **Initialization criterion**: the initialization criterion will generate an initial population randomly. Thus, each individual’s chromosome is randomly coded. The mechanism would be: “it will begin with a randomly chosen city, the next city is also randomly chosen among all but the one initially chosen, and thus will be selected randomly from the remaining until a solution is complete”. This process is repeated as many times as individuals are.

• **Stopping criterion**: 1000 iterations for the simple generational algorithm and 10000 for the steady state algorithm.

• **Fitness function**: the fitness function is the total distance traveled to visit the cities in the order determined by the solution to be evaluated. The compute of the fitness function of each neighbor solution can be optimally performed.  

  - Unoptimized way:
    \[
    \text{fitness function} = \sum_{i=1}^{n} D(i, j)
    \]
    Where \(d(i, j)\) is the distance between the cities \(i\) and \(j\) traveling in the order established.

  - Optimized way: This way uses the new solution generated, removing from the previous one the changes with respect to the new solution and adding those of the new, thus it will not depend on the number of cities, the formula would be:
    \[
    \text{new_f}itness = \text{old_f}itness - D(S[i - 1], S[i]) - D(S[i], S[i + 1]) \\
    - D(S[j - 1], S[j]) - D(S[j], S[j + 1]) + D(S[i - 1], S[j]) \\
    + D(S[j], S[i + 1]) + D(S[j - 1], S[i]) + D(S[i], S[j + 1])
    \]
    Where \((i, j)\) are the positions of the vector which have been interchanged from a solution to another. Moreover, it should be fulfilled the following cases: in case of being exchanged the first or the last city, it must be known that the previous of the first city is the last one and the following of the last is the first. It should also be borne in mind when exchanging two consecutive cities, i.e. \(i = j + 1\). Considering the first city is \(i\) and the second is \(j\).

• **Evolution outline**: a generational algorithm with elitism and a steady state algorithm in which each iteration will select two parents.

• **Criterion for selecting parents**: a binary tournament size equal to 2, consisting of two individuals chosen randomly from the population and selecting the best.

• **Replacing criterion**: in the generational algorithm, the offsprings automatically replace their parents. There will be elitism, so if the best solution from the older generation does not survive then the worst solution from the new population is replaced by the best solution from the older generation. In the steady state algorithm, the offsprings and the parents compete and the two best are incorporated into the new population.
- **Crossover operators**: the following operators will be used:
  
  * **PMX Operator**: operates by exchanging parts among two breakpoints, but it fixes the solutions based on the correspondence between parents. Example:

    - \( A = 1 \ 2 \ 3 | 4 \ 5 \ 6 | 7 \ 8 \ 9 \)
    - \( B = 8 \ 6 \ 4 | 2 \ 9 \ 7 | 5 \ 3 \ 1 \)

    The section exchanged remains the same. The rest of the permutation is fixed using the mapping exchanged:

    \[
    \begin{array}{c}
    \uparrow \ \uparrow \ \uparrow \\
    2 \ 9 \ 7 \\
    \end{array}
    \]

    The new individuals would be:

    - \( A = 1 \ 4 \ 3 | 2 \ 9 \ 7 | 6 \ 8 \ 5 \)
    - \( B = 8 \ 7 \ 2 | 4 \ 5 \ 6 | 9 \ 3 \ 1 \)

  * **OX operator**: similar to PMX, it does not fix by exchanging but creates gaps that are completed in order to move from the second breakpoint. Example:

    - \( A = 1 \ 2 \ 3 | 4 \ 5 \ 6 | 7 \ 8 \ 9 \)
    - \( B = 8 \ 6 \ 4 | 2 \ 9 \ 7 | 5 \ 3 \ 1 \)

    The first individual is:

    \[
    \begin{array}{c}
    1 \ H \ 3 | 4 \ 5 \ 6 | H \ 8 \ H \\
    4 \ 5 \ 6 | H \ H \ H | 8 \ 1 \ 3 \\
    4 \ 5 \ 6 | 2 \ 9 \ 7 | 8 \ 1 \ 3 \\
    \end{array}
    \]

    The second individual is:

    \[
    \begin{array}{c}
    8 \ H \ H | 2 \ 9 \ 7 | H \ 3 \ 1 \\
    2 \ 9 \ 7 | H \ H \ H | 3 \ 1 \ 8 \\
    2 \ 9 \ 7 | 4 \ 5 \ 6 | 3 \ 1 \ 8 \\
    \end{array}
    \]

    PMX and OX operators will be used with a 75% probability of occurrence.
Mutation operators

* 2-opt operator: it is an exchange operator which takes two elements and interchanges their positions. Thus, it changes the arrival and the departure of two cities. \((1 \ 2 \ 4 \ 3 \ 8 \ 5 \ 7 \ 6) \Rightarrow (1 \ 2 \ 5 \ 3 \ 8 \ 4 \ 7 \ 6)\) swaps city 4 and 5.

* Random sublist operator: a sublist is shuffled, so any combination is different and the only condition is that it differs from the previous ones. This operator produces a major change of the current solution regarding to the neighbor solution to be generated. Here the number of cities involved changes depending on the interval chosen, so that it will no be applied the optimized fitness function because it would be almost as expensive as determining the cities that have been modified and would require many more changes.

2-opt and random sublist operators will be used with a 10% probability of occurrence.

Operating parameters: the population size is 100 individuals.

### 3.4 Implementation

This section describes how to encode the configuration file required to run the algorithm in the library JCLEC and how to implement basic methods for evaluating individuals in the population.

This tutorial uses `net.sf.jclec.orderarray` package which includes the codification of individuals, crossover and mutation operators and all those classes that are necessary for the resolution of the TSP problem.

- Firstly, it is necessary to determine which genetic algorithm will be used to solve the problem, using the package `net.sf.jclec.algorithm`. In this case, two different algorithms will be employed separately. When using the simple generational elitist, the following value will be selected for the attribute algorithm-type:

```xml
1 <process algorithm-type="net.sf.jclec.algorithm.classic.SGE"/>
```

In the case of the steady state, it is necessary to change this XML attribute as shown below:

```xml
1 <process algorithm-type="net.sf.jclec.algorithm.classic.SS"/>
```
3.4 Implementation

- **Coding:** the class to use for representing the solution must be indicated. Using binary representation, the JCLEC packages `net.jclec.orderarray` and `OrderArrayIndividualSpecies` are employed. It is not necessary to implement anything else because these individuals are directly valid from the representation that has been chosen, specifying the size, i.e. the number of genes, for each individual which depends on the number of cities to be considered.

```xml
<species type="net.sf.jclec.orderarray.OrderArrayIndividualSpecies" genotype-length="52"/>
```

- **Initialization:** the population is randomly initialized using binary genotypes employing the class `net.sf.jclec.orderarray.OrderArrayCreator`.

```xml
<provider type="net.sf.jclec.orderarray.OrderArrayCreator"/>
```

- **Stopping criterion:** This is set to a maximum number of generations.

```xml
<max-of-generations>1000</max-of-generations>
```

- **Fitness function:** This step involves implementing an appropriate evaluator, according to the characteristics that were determined to evaluate the population and treat individuals. This class will extend from `AbstractEvaluator`, located in the package `net.sf.jclec.base`. By extending it, those abstract methods are implemented. The method `evaluate()` evaluates individuals given as argument and sets their fitness values.

```xml
<evaluator type="tutorial.TSP" file-name="berlin52.tsp" number-cities="52"/>
```

Below is the method `evaluate()` included in the file `TSP.java` necessary for the evaluation of the individuals.

```java
protected void evaluate(IIndividual ind) {
    // Individual genotype
    int[] genotype = ((OrderArrayIndividual)ind).getGenotype();
    double distance = 0;

    for (int i=0; i<genotype.length-1; i++)
        distance += distancesMatrix[genotype[i]][genotype[i+1]];

    distance += distancesMatrix[genotype[genotype.length-1]][genotype[0]];
    ind.setFitness(new SimpleValueFitness(distance));
}
```
Genetic algorithms using integer encoding: Traveling Salesman Problem

- **Selection criterion** of the parents can be set by using the `net.sf.jclec.selector` package. The selector is implemented by a tournament:

```xml
<parents selector type="net.sf.jclec.selector.TournamentSelector" tournament size="2" />
```

- **Genetic operators**: The package `net.sf.jclec.orderarray` contains mutation and crossover operators ready to use, which are described briefly below. These are the mutation operators `Order2OptMutator` and `OrderSublistMutator` and crossover operators `OrderOXCrossover` and `OrderPMXCrossover`.

```xml
<mutator type="net.sf.jclec.orderarray.mut.Order2OptMutator" evaluate="true" mut prob="0.1" />
<recombinator type="net.sf.jclec.orderarray.rec.OrderPMXCrossover" evaluate="true" rec prob="0.9" />
```

- **Parameters**: with respect to other parameters, it is necessary to define the size of the population and the seed for the random number generator:

```xml
<rand gen factory type="net.sf.jclec.util.random.RanecuFactory" seed="987328938" />
<population size=100 /></population size>
```

Finally, whichever the algorithm is used, it is necessary to determine the listener to be used to display the results, `net.sf.jclec.listener` package. In this example, a basic report generator is used. It produces a file with the best, the worst, the average individual, the fitness mean and variance in 50 generations.

```xml
<listener type="net.sf.jclec.listener.PopulationReporter" />
<report frequency=50 /></report frequency>
</listener>
```

In order to speed up the problem, the distances matrix is calculated at first only once. To do this, the `configure()` method loads the coordinates of each city from a file problem indicated in the configuration XML file and then it performs the matrix calculations.

```java
public void configure(Configuration settings) {
    String nFile = settings.getString("[@file-name]");
    String line = null;
    int nCities = settings.getInt("[@number-cities]"), i=0;
    File f = new File(nFile);
    distancesMatrix = new double[nCities][nCities];
    double x[] = new double[nCities];
    double y[] = new double[nCities];
    try {
```
3.4 Implementation

```java
FileReader fr = new FileReader(f);
BufferedReader br = new BufferedReader(fr);

while (!((line = br.readLine()).contains("NODE_COORD_SECTION")))
    while (!((line = br.readLine()).contains("EOF")))
    {
        line = String.copyValueOf(line.toCharArray(), line.indexOf(" "), line.length() - line.indexOf(" "));
        line = line.trim();
        x[i] = Double.valueOf(line.split(" ")[0]).doubleValue();
        line = String.copyValueOf(line.toCharArray(), line.indexOf(" "), line.length() - line.indexOf(" "));
        line = line.trim();
        y[i] = Double.valueOf(line.split(" ")[0]).doubleValue();
        i++;
    }
    for (int j = i + 1; j < nCities - 1; j++)
    {
        distancesMatrix[i][j] = Math.sqrt(Math.pow(x[i] - x[j], 2) + Math.pow(y[i] - y[j], 2));
        distancesMatrix[j][i] = distancesMatrix[i][j];
    }
    catch (IOException e) {
        System.out.println(e);
    }
    // Maximize flag
    setMaximize(false);
```

Below it is shown the configuration XML file that serves as the basis for the development of the tutorial.

```xml
<experiment>
    <process algorithm-type="net.sf.jclec.algorithm.classic.SGE">
        <rand-gen factory-type="net.sf.jclec.util.random.RanecuFactory" seed="987328938" />
        <population size=100 / population-size>1000< /max-of-generations>1000< /max-of-generations>
        <species type="net.sf.jclec .orderarray.OrderArrayIndividualSpecies" genotype-length="52" />
        <evaluator type="tutorial.TSP" file-name="berlin52.tsp" number-cities="52" />
        <provider type="net.sf.jclec .orderarray.OrderArrayCreator" />
        <parents selector-type="net.sf.jclec .selector .TournamentSelector" tournament-size="2" />
        <mutator type="net.sf.jclec .orderarray .mut .Order2OptMutator" mut-prob="0.1" />
        <recombinator type="net.sf.jclec .orderarray .rec .OrderPMXCrossover" rec-prob="0.9" />
        <listener type="net.sf.jclec .listener .PopulationReporter" />
        <report frequency>50</report frequency>
    </process>
</experiment>
```
3.5 Results

Using the configuration file from above, the parameters can be adjusted. Below, results of different executions are shown in order to find out which combination of crossover and mutation operators performs best. It is recommended to use a low value for mutation probability, and a high one for crossover. It is also recommended to run several times the algorithm using different seed values.

<table>
<thead>
<tr>
<th>Execution 1</th>
<th>Seed</th>
<th>Berlin52</th>
<th>Bier127</th>
<th>D198</th>
<th>Lin318</th>
<th>Rat575</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>123456789</td>
<td>10226</td>
<td>264416</td>
<td>71731</td>
<td>278644</td>
<td>64199</td>
</tr>
<tr>
<td>Execution 2</td>
<td>234567891</td>
<td>10133</td>
<td>250683</td>
<td>74179</td>
<td>288129</td>
<td>64554</td>
</tr>
<tr>
<td>Execution 3</td>
<td>345678912</td>
<td>10480</td>
<td>251573</td>
<td>71446</td>
<td>277827</td>
<td>64265</td>
</tr>
<tr>
<td>Execution 4</td>
<td>456789123</td>
<td>11046</td>
<td>251992</td>
<td>68976</td>
<td>292310</td>
<td>64175</td>
</tr>
<tr>
<td>Execution 5</td>
<td>567891234</td>
<td>9544</td>
<td>244457</td>
<td>71105</td>
<td>283137</td>
<td>62892</td>
</tr>
<tr>
<td>Average</td>
<td>-</td>
<td>10286</td>
<td>252624</td>
<td>71487</td>
<td>284010</td>
<td>64017</td>
</tr>
<tr>
<td>Variance</td>
<td>-</td>
<td>298214</td>
<td>52754818</td>
<td>3438879</td>
<td>38409318</td>
<td>418098</td>
</tr>
<tr>
<td>Deviation</td>
<td>-</td>
<td>546</td>
<td>7263</td>
<td>1854</td>
<td>6198</td>
<td>647</td>
</tr>
</tbody>
</table>

Table 3.1: TwoOpt mutator + PMX crossover for generational elitist algorithm.

<table>
<thead>
<tr>
<th>Execution 1</th>
<th>Seed</th>
<th>Berlin52</th>
<th>Bier127</th>
<th>D198</th>
<th>Lin318</th>
<th>Rat575</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>123456789</td>
<td>15034</td>
<td>381030</td>
<td>99197</td>
<td>399753</td>
<td>83712</td>
</tr>
<tr>
<td>Execution 2</td>
<td>234567891</td>
<td>13758</td>
<td>361216</td>
<td>105071</td>
<td>397539</td>
<td>84920</td>
</tr>
<tr>
<td>Execution 3</td>
<td>345678912</td>
<td>13367</td>
<td>390990</td>
<td>103701</td>
<td>414450</td>
<td>84354</td>
</tr>
<tr>
<td>Execution 4</td>
<td>456789123</td>
<td>15563</td>
<td>395772</td>
<td>97607</td>
<td>400776</td>
<td>82497</td>
</tr>
<tr>
<td>Execution 5</td>
<td>567891234</td>
<td>15252</td>
<td>355574</td>
<td>102466</td>
<td>394863</td>
<td>81937</td>
</tr>
<tr>
<td>Average</td>
<td>-</td>
<td>14595</td>
<td>376916</td>
<td>101608</td>
<td>401476</td>
<td>83484</td>
</tr>
<tr>
<td>Variance</td>
<td>-</td>
<td>941854</td>
<td>318131965</td>
<td>9733987</td>
<td>57754501</td>
<td>1559829</td>
</tr>
<tr>
<td>Deviation</td>
<td>-</td>
<td>970</td>
<td>17836</td>
<td>3120</td>
<td>7600</td>
<td>1249</td>
</tr>
</tbody>
</table>

Table 3.2: SubList mutator + PMX crossover for generational elitist algorithm.
### 3.5 Results

<table>
<thead>
<tr>
<th>Execution</th>
<th>Seed</th>
<th>Berlin52</th>
<th>Bier127</th>
<th>D198</th>
<th>Lin318</th>
<th>Rat575</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution 1</td>
<td>123456789</td>
<td>8503</td>
<td>240238</td>
<td>49025</td>
<td>233947</td>
<td>51348</td>
</tr>
<tr>
<td>Execution 2</td>
<td>234567891</td>
<td>8396</td>
<td>223662</td>
<td>48541</td>
<td>252552</td>
<td>50606</td>
</tr>
<tr>
<td>Execution 3</td>
<td>345678912</td>
<td>8423</td>
<td>225100</td>
<td>45636</td>
<td>240126</td>
<td>51590</td>
</tr>
<tr>
<td>Execution 4</td>
<td>456789123</td>
<td>9018</td>
<td>228507</td>
<td>44779</td>
<td>249266</td>
<td>52577</td>
</tr>
<tr>
<td>Execution 5</td>
<td>567891234</td>
<td>8421</td>
<td>241740</td>
<td>45247</td>
<td>238722</td>
<td>52280</td>
</tr>
<tr>
<td>Average</td>
<td>-</td>
<td>8552</td>
<td>231849</td>
<td>46646</td>
<td>242922</td>
<td>51680</td>
</tr>
<tr>
<td>Variance</td>
<td>-</td>
<td>69382</td>
<td>72992077</td>
<td>3928558</td>
<td>59746376</td>
<td>609160</td>
</tr>
<tr>
<td>Deviation</td>
<td>-</td>
<td>263</td>
<td>8544</td>
<td>1982</td>
<td>7730</td>
<td>780</td>
</tr>
</tbody>
</table>

Table 3.3: TwoOpt mutator + OX crossover for generational elitist algorithm.

<table>
<thead>
<tr>
<th>Execution</th>
<th>Seed</th>
<th>Berlin52</th>
<th>Bier127</th>
<th>D198</th>
<th>Lin318</th>
<th>Rat575</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution 1</td>
<td>123456789</td>
<td>9301</td>
<td>269041</td>
<td>49208</td>
<td>271421</td>
<td>56626</td>
</tr>
<tr>
<td>Execution 2</td>
<td>234567891</td>
<td>8344</td>
<td>245012</td>
<td>50498</td>
<td>258746</td>
<td>55651</td>
</tr>
<tr>
<td>Execution 3</td>
<td>345678912</td>
<td>9401</td>
<td>260494</td>
<td>49681</td>
<td>251656</td>
<td>53973</td>
</tr>
<tr>
<td>Execution 4</td>
<td>456789123</td>
<td>9307</td>
<td>243747</td>
<td>49903</td>
<td>260643</td>
<td>55894</td>
</tr>
<tr>
<td>Execution 5</td>
<td>567891234</td>
<td>9012</td>
<td>240793</td>
<td>48196</td>
<td>253768</td>
<td>54133</td>
</tr>
<tr>
<td>Average</td>
<td>-</td>
<td>9073</td>
<td>251817</td>
<td>49497</td>
<td>259247</td>
<td>55256</td>
</tr>
<tr>
<td>Variance</td>
<td>-</td>
<td>187287</td>
<td>151229403</td>
<td>744256</td>
<td>59512940</td>
<td>1337105</td>
</tr>
<tr>
<td>Deviation</td>
<td>-</td>
<td>433</td>
<td>12298</td>
<td>863</td>
<td>7714</td>
<td>1156</td>
</tr>
</tbody>
</table>

Table 3.4: SubList mutator + OX crossover for generational elitist algorithm.

<table>
<thead>
<tr>
<th>Execution</th>
<th>Seed</th>
<th>Berlin52</th>
<th>Bier127</th>
<th>D198</th>
<th>Lin318</th>
<th>Rat575</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution 1</td>
<td>123456789</td>
<td>11327</td>
<td>312719</td>
<td>78848</td>
<td>358175</td>
<td>75083</td>
</tr>
<tr>
<td>Execution 2</td>
<td>234567891</td>
<td>11330</td>
<td>336741</td>
<td>84278</td>
<td>342543</td>
<td>74842</td>
</tr>
<tr>
<td>Execution 3</td>
<td>345678912</td>
<td>12729</td>
<td>311461</td>
<td>83626</td>
<td>338349</td>
<td>73961</td>
</tr>
<tr>
<td>Execution 4</td>
<td>456789123</td>
<td>11779</td>
<td>338736</td>
<td>81281</td>
<td>347450</td>
<td>75342</td>
</tr>
<tr>
<td>Execution 5</td>
<td>567891234</td>
<td>12459</td>
<td>310505</td>
<td>82339</td>
<td>348058</td>
<td>73547</td>
</tr>
<tr>
<td>Average</td>
<td>-</td>
<td>11925</td>
<td>322032</td>
<td>82074</td>
<td>346915</td>
<td>74555</td>
</tr>
<tr>
<td>Variance</td>
<td>-</td>
<td>416569</td>
<td>206682668</td>
<td>4593074</td>
<td>55217702</td>
<td>587340</td>
</tr>
<tr>
<td>Deviation</td>
<td>-</td>
<td>645</td>
<td>14376</td>
<td>2143</td>
<td>7431</td>
<td>766</td>
</tr>
</tbody>
</table>

Table 3.5: TwoOpt mutator + PMX crossover for Steady State algorithm.
Genetic algorithms using integer encoding: Traveling Salesman Problem

Table 3.6: SubList mutator + PMX crossover for Steady State algorithm.

<table>
<thead>
<tr>
<th></th>
<th>Seed</th>
<th>Berlin52</th>
<th>Bier127</th>
<th>D198</th>
<th>Lin318</th>
<th>Rat575</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution 1</td>
<td>123456789</td>
<td>13995</td>
<td>435471</td>
<td>117187</td>
<td>450571</td>
<td>92893</td>
</tr>
<tr>
<td>Execution 2</td>
<td>234567891</td>
<td>15333</td>
<td>443931</td>
<td>118143</td>
<td>461033</td>
<td>92913</td>
</tr>
<tr>
<td>Execution 3</td>
<td>345678912</td>
<td>14355</td>
<td>430996</td>
<td>122662</td>
<td>457145</td>
<td>92525</td>
</tr>
<tr>
<td>Execution 4</td>
<td>456789123</td>
<td>15092</td>
<td>458541</td>
<td>113410</td>
<td>449225</td>
<td>92169</td>
</tr>
<tr>
<td>Execution 5</td>
<td>567891234</td>
<td>16016</td>
<td>425392</td>
<td>120642</td>
<td>454418</td>
<td>91678</td>
</tr>
<tr>
<td>Average</td>
<td>-</td>
<td>14958</td>
<td>438866</td>
<td>118409</td>
<td>454478</td>
<td>92436</td>
</tr>
<tr>
<td>Variance</td>
<td>-</td>
<td>642389</td>
<td>166942367</td>
<td>12407408</td>
<td>23234720</td>
<td>272651</td>
</tr>
<tr>
<td>Deviation</td>
<td>-</td>
<td>801</td>
<td>12921</td>
<td>3522</td>
<td>4820</td>
<td>522</td>
</tr>
</tbody>
</table>

Table 3.7: TwoOpt mutator + OX crossover for Steady State algorithm.

<table>
<thead>
<tr>
<th></th>
<th>Seed</th>
<th>Berlin52</th>
<th>Bier127</th>
<th>D198</th>
<th>Lin318</th>
<th>Rat575</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution 1</td>
<td>123456789</td>
<td>10774</td>
<td>344695</td>
<td>75952</td>
<td>355329</td>
<td>71500</td>
</tr>
<tr>
<td>Execution 2</td>
<td>234567891</td>
<td>12715</td>
<td>340033</td>
<td>80079</td>
<td>352832</td>
<td>74584</td>
</tr>
<tr>
<td>Execution 3</td>
<td>345678912</td>
<td>10986</td>
<td>345766</td>
<td>72978</td>
<td>340399</td>
<td>72881</td>
</tr>
<tr>
<td>Execution 4</td>
<td>456789123</td>
<td>12751</td>
<td>32146</td>
<td>70180</td>
<td>365730</td>
<td>71345</td>
</tr>
<tr>
<td>Execution 5</td>
<td>567891234</td>
<td>12401</td>
<td>32450</td>
<td>76363</td>
<td>344346</td>
<td>73788</td>
</tr>
<tr>
<td>Average</td>
<td>-</td>
<td>11925</td>
<td>337018</td>
<td>75110</td>
<td>351345</td>
<td>72820</td>
</tr>
<tr>
<td>Variance</td>
<td>-</td>
<td>934455</td>
<td>95129795</td>
<td>13955303</td>
<td>95149836</td>
<td>1992361</td>
</tr>
<tr>
<td>Deviation</td>
<td>-</td>
<td>967</td>
<td>9753</td>
<td>3736</td>
<td>9754</td>
<td>1412</td>
</tr>
</tbody>
</table>

Table 3.8: SubList mutator + OX crossover for Steady State algorithm.

<table>
<thead>
<tr>
<th></th>
<th>Seed</th>
<th>Berlin52</th>
<th>Bier127</th>
<th>D198</th>
<th>Lin318</th>
<th>Rat575</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution 1</td>
<td>123456789</td>
<td>13824</td>
<td>390015</td>
<td>107356</td>
<td>445057</td>
<td>91541</td>
</tr>
<tr>
<td>Execution 2</td>
<td>234567891</td>
<td>13182</td>
<td>414756</td>
<td>102935</td>
<td>449846</td>
<td>91478</td>
</tr>
<tr>
<td>Execution 3</td>
<td>345678912</td>
<td>12200</td>
<td>416680</td>
<td>102180</td>
<td>455326</td>
<td>94479</td>
</tr>
<tr>
<td>Execution 4</td>
<td>456789123</td>
<td>12948</td>
<td>424041</td>
<td>100649</td>
<td>452418</td>
<td>90448</td>
</tr>
<tr>
<td>Execution 5</td>
<td>567891234</td>
<td>12842</td>
<td>413197</td>
<td>96268</td>
<td>46493</td>
<td>92408</td>
</tr>
<tr>
<td>Average</td>
<td>-</td>
<td>12999</td>
<td>413538</td>
<td>101878</td>
<td>453468</td>
<td>92071</td>
</tr>
<tr>
<td>Variance</td>
<td>-</td>
<td>344878</td>
<td>83172020</td>
<td>16049497</td>
<td>53604678</td>
<td>2294458</td>
</tr>
<tr>
<td>Deviation</td>
<td>-</td>
<td>587</td>
<td>9120</td>
<td>4006</td>
<td>7322</td>
<td>1515</td>
</tr>
</tbody>
</table>
The results of the last eight tables indicates that the generational elitist algorithm always performs better than the steady state algorithm obtaining better fitness values from individuals. Figures 3.2 and 3.3 summarize these results.

Figure 3.2: Generational elitist algorithm fitness.

Figure 3.3: Steady state algorithm fitness.
4. GENETIC ALGORITHMS USING REAL ENCODING: OPTIMIZATION OF REAL FUNCTIONS

4.1 Introduction

This section aims to explain how to optimize real functions of real variable applying genetic algorithms. The purpose of this section is to study the performance of a set of real-coded evolutionary algorithms for the optimization of real functions of real variable, also analyzing the quality of the solutions obtained. These algorithms, that are available in JCLEC library, will be applied to a set of 3 different functions.

4.2 Problem definition

The problem proposed here is to find the global optimum of 3 functions. Each function will be minimized in two different domains for variable $x$. Their description and representation (Figure 4.2; first represents the Sphere function, then the Rastrigin function, and finally the Ackley function) are shown below:

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
<th>Domain 1</th>
<th>Domain 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sphere</td>
<td>$f_1(x) = \sum_{i=1}^{p} x_i^2$</td>
<td>[-5.12, 5.12]</td>
<td>[0, 5.12]</td>
</tr>
<tr>
<td>Rastrigin</td>
<td>$f_2(x) = \sum_{i=1}^{p} (x_i^2 - 10 \cos(2\pi x_i) + 10)$</td>
<td>[-5.12, 5.12]</td>
<td>[0, 5.12]</td>
</tr>
<tr>
<td>Ackley</td>
<td>$f_3(x) = e - 20 exp\left(-0.2 \sqrt{\frac{1}{p} \sum_{i=1}^{p} x_i^2}\right) - exp\left(\frac{1}{p} \sum_{t=1}^{p} \cos(2\pi x_t)\right)$</td>
<td>[-30, 30]</td>
<td>[0, 30]</td>
</tr>
</tbody>
</table>

4.3 Genetic algorithms

The functions selected will be resolved by different genetic algorithms, such as a generational genetic algorithm, steady state genetic algorithm and two multimodal algorithms, which are detailed now.
4.3.1 Common configuration

The following parameters have the same configuration values for all the algorithms:

- **Encoding criterion**: the representation will adopt the shape of a vector of real values. Thus, each solution is encoded using a real vector of size $n$, being $n$ equal to 5 for this problem. Each gene of a chromosome is a real number that takes a value within a range depending on the domain of the corresponding variable.

- **Initialization**: the population will be randomly initialized.

- **Fitness function**: the function to be minimized is directly used as objective function. Then, a specific evaluator is implemented to measure the quality of individuals through the function expression (the lower the fitness, the better the individual). To specify if the fitness function will be minimized or maximized, there is an object, `ValueFitnessComparator`, that has a configuration variable called `maximize`, which should be set to true or false (in this case, false) for maximizing or minimizing respectively.

- **Criterion for selecting parents**: a binary tournament will be used, which selects the best of two individuals chosen randomly from the population.
4.3 Genetic algorithms

- **Crossover operator**: the BLX-\(\alpha\) crossover operator will be used. This operator is available in `net.sf.jclec.realarray.rec.BLXAlphaCrossover`. This crossover depends on alpha parameter which decides the crossover interval that is shown in the following figure. If alpha value is zero, this crossover is equal than flat crossover.

![Figure 4.2: BLX-\(\alpha\) crossover operator.](image)

- **Mutation operator**: the non uniform mutation operator will be used. This operator is available in `net.sf.jclec.realarray.mut.NonUniformMutator`.

- **Other parameters**: the size of the population will be 100 individuals.

### 4.3.2 Generational genetic algorithm

The following parameters are configured specifically for the SGE algorithm:

- **Definition**: The generational genetic algorithm (the “Standard Genetic algorithm”) creates a new offspring from the members of an old population using the genetic operators and places these individuals in a new population which becomes the old population when the whole new population is created.

- **Stop criterion**: 1000 generations will be carried out.

- **Evolution scheme**: a generational scheme with elitism is considered. Specifically, the selected class will be: `net.sf.jclec.algorithm.classic.SGE`.

- **Criterion for replacing**: the offspring replace their parents automatically. For the elitism, if the best solution from the previous generation does not survive, the worst solution is replaced by it in the new population.

- **Other parameters**: the population size will be 100 individuals.

### 4.3.3 Steady state genetic algorithm

The following parameters are configured specifically for the SS algorithm:

- **Definition**: The generational genetic algorithm (the “Standard Genetic algorithm”) creates a new offspring from the members of an old population using the genetic operators and places these individuals in a new population which becomes the old population when the whole new population is created.

- **Stop criterion**: 1000 generations will be carried out.

- **Evolution scheme**: a generational scheme with elitism is considered. Specifically, the selected class will be: `net.sf.jclec.algorithm.classic.SGE`.

- **Criterion for replacing**: the offspring replace their parents automatically. For the elitism, if the best solution from the previous generation does not survive, the worst solution is replaced by it in the new population.

- **Other parameters**: the population size will be 100 individuals.
• **Definition**: The steady state genetic algorithm is different with respect to the generational model in that there is typically one single new member inserted into the new population at any one time. A replacement/deletion strategy defines which member of the population will be replaced by the new offspring.

• **Stop criterion**: 50000 generations will be carried out.

• **Evolution scheme**: a stationary schema is considered. Specifically, the selected class will be: `net.sf.jclec.algorithm.classic.SS`.

• **Criterion for replacing**: the two offsprings generated are compared with their parents. Then, the two best individuals will be kept in the population.

### 4.3.4 Multimodal algorithms

Sharing and Clearing versions will be also tested. Their main characteristics are:

• **Stop criterion**: 1000 generations will be carried out.

• **Evolution scheme**: a generational scheme with elitism is considered. Specifically, the class selected for the Clearing algorithm is `net.sf.jclec.algorithm.niching.Clearing`, and for the Sharing algorithm is `net.sf.jclec.algorithm.niching.Sharing`. This algorithm has a parameter, called $\alpha$, whose value is set at 1.0.

• **Criterion for replacing**: the offsprings replace their parents automatically. For elitism, the two offsprings generated are compared with their parents. Then, the two best individuals will be kept in the population.

### 4.4 Implementation

This section describes how to encode the configuration file required to run the algorithm in the library JCLEC and how to implement basic methods for evaluating individuals in the population.

This tutorial uses `net.sf.jclecREALARRAY` package which includes the codification of individuals, crossover and mutation operators and all those classes that are necessary for its resolution.

• The functions selected will be resolved by different genetic algorithms, such as a generational genetic algorithm, steady state genetic algorithm and two multimodal algorithms.
4.4 Implementation

- Generational genetic algorithm

```xml
<process algorithm-type="net.sf.jclec.algorithm.classic.SGE"/>
```

- Steady state genetic algorithm

```xml
<process algorithm-type="net.sf.jclec.algorithm.classic.SS"/>
```

- Sharing algorithm

```xml
<process algorithm-type="net.sf.jclec.algorithm.niching.Clearing"/>
```

- Clearing algorithm

```xml
<process algorithm-type="net.sf.jclec.algorithm.niching.Sharing"/>
```

- **Coding**: it is necessary to indicate the class to use for representing the solution. Using real representation, the JCLEC package `net.jclec.realarray` and `RealArrayIndividualSpecies` are used. It is necessary to specify the interval range for each gene from the genotype for the domains 1 and 2.

```xml
<species type="net.sf.jclec.realarray.RealArrayIndividualSpecies">
  <genotype-schema>
    <locus type="net.sf.jclec.util.range.Interval" left="0" right="5.12" closure="closed-closed"/>
    <locus type="net.sf.jclec.util.range.Interval" left="0" right="5.12" closure="closed-closed"/>
    <locus type="net.sf.jclec.util.range.Interval" left="0" right="5.12" closure="closed-closed"/>
    <locus type="net.sf.jclec.util.range.Interval" left="0" right="5.12" closure="closed-closed"/>
    <locus type="net.sf.jclec.util.range.Interval" left="0" right="5.12" closure="closed-closed"/>
    <locus type="net.sf.jclec.util.range.Interval" left="0" right="5.12" closure="closed-closed"/>
    <locus type="net.sf.jclec.util.range.Interval" left="0" right="5.12" closure="closed-closed"/>
    </genotype-schema>
  </species>
```

- **Initialization**: the population is randomly initialized using binary genotypes employing the class `net.sf.jclec.orderarray.RealArrayCreator`.

```xml
<provider type="net.sf.jclec.orderarray.RealArrayCreator"/>
```

- **Stopping criterion**: This is set to a maximum number of generations.

```xml
<max-of-generations>50000</max-of-generations>
```
Fitness function: This step involves implementing an appropriate evaluator, according to the characteristics that were determined to evaluate the population and treat infeasible individuals. This class will extend from AbstractEvaluator, located in the package net.sf.jclec.base. By extending it, those abstract methods are implemented. The method evaluate() evaluates individuals given as argument and sets their fitness values.

- F1. Sphere evaluator

The method that represents the evaluator for the Sphere function is shown below:

```java
protected void evaluate(IIndividual ind)
{
    // Individual genotype
    double[] genotype = ((RealArrayIndividual)ind).getGenotype();
    double fitness = 0.;

    for (int i = 0; i < genotype.length; i++)
        fitness += Math.pow(genotype[i], 2);

    ind.setFitness(new SimpleValueFitness(fitness));
}
```

- F2. Rastrigin evaluator

The method that represents the evaluator for the Rastrigin function is shown below:

```java
protected void evaluate(IIndividual ind)
{
    // Individual genotype
    double[] genotype = ((RealArrayIndividual)ind).getGenotype();
    double fitness = 0.;

    for (int i = 0; i < genotype.length; i++)
        fitness += Math.pow(genotype[i], 2) - Math.cos(2 * Math.PI * genotype[i]) + 10;

    ind.setFitness(new SimpleValueFitness(fitness));
}
```

- F3. Ackley evaluator

The method that represents the evaluator for the Ackley function is shown below:

```java
protected void evaluate(IIndividual ind)
{
    // Individual genotype
    double[] genotype = ((RealArrayIndividual)ind).getGenotype();
    double fitness = 0., sum = 0., sum2 = 0;

    for (int i = 0; i < genotype.length; i++)
    {
    }
```
4.4 Implementation

```java
sum += Math.pow(genotype[i], 2);
sum2 += Math.cos(2.0 * Math.PI * genotype[i]);
}
for (int i = 0; i < genotype.length; i++)
    fitness += 20.0 + Math.E - 20.0 * Math.exp(-0.2 * Math.sqrt(sum / genotype.length)) - Math.exp((sum2 / genotype.length));
ind.setFitness(new SimpleValueFitness(fitness));
```

- **Selection criterion** of the parents can be set by using the `net.sf.jclec.selector` package. The selector is implemented by a tournament of size two:

```xml
<parents selector type="net.sf.jclec.selector.TournamentSelector" tournament=size="2"/>
```

- **Genetic operators**: The package `net.sf.jclec.reعالarray` contains mutation and crossover operators ready to use, `NonUniformMutator` and `BLXAlphaCrossover` are used.

```xml
<mutator type="net.sf.jclec.reعالarray.mut.NonUniformMutator" mut=prob="0.15"/>
<recombinator type="net.sf.jclec.reعالarray.rec.BLXAlphaCrossover" rec=prob="0.9" alpha="0.3"/>
```

- **Parameters**: with respect to other parameters, it is necessary to define the size of the population and the seed for the random number generator:

```xml
<rand gen factory type="net.sf.jclec.util.random.RanecuFactory" seed="987328938"/>
<population size=100/>
```

Finally, whichever the algorithm is used, the listener displays the results. In this example, a basic report generator is used. It produces a file with the best, the worst, the average individual, the fitness mean and variance in 50 generations.

```xml
<listener type="net.sf.jclec.listener.PopulationReporter">
<report frequency=50/>
</listener>
```

Below it is shown the configuration XML file that serves as the basis for the development of the tutorial.

```xml
<experiment>
<process algorithm type="net.sf.jclec.algorithm.classic.SGE">
<population size=100/>
<max of generations=1000/>
<rand gen factory type="net.sf.jclec.util.random.RanecuFactory" seed="124321453"/>
<species type="net.sf.jclec.reعالarray.RealArrayIndividualSpecies">
<genotype schema>
```
4.5 Results

In this section, the results obtained for each function and for each algorithm (generational, steady state and multimodal) are compared. For each algorithm and each function, the different individuals will take values from two different domains.

4.5.1 Generational genetic algorithm

The results obtained for the first domain are shown in Table 4.1. Analyzing these results, it can be noticed that the Sphere function obtains the best average value (closest to zero). The Y axis represents the values, whereas the X axis states for the different seeds used.

The results obtained for the second domain are shown in Table 4.2. Analyzing these results, it can be noticed that the Sphere function obtains the best average value (closest to zero). The Y axis represents the values, whereas the X axis states for the different seeds used.
### 4.5 Results

<table>
<thead>
<tr>
<th>Execution</th>
<th>Seed</th>
<th>Sphere</th>
<th>Rastrigin</th>
<th>Ackley</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>124321453</td>
<td>3.979</td>
<td>30.6845</td>
<td>66.6672</td>
</tr>
<tr>
<td>2</td>
<td>983746234</td>
<td>3.2828</td>
<td>23.6429</td>
<td>69.0209</td>
</tr>
<tr>
<td>3</td>
<td>139548473</td>
<td>6.0807</td>
<td>41.5819</td>
<td>61.9542</td>
</tr>
<tr>
<td>4</td>
<td>394827384</td>
<td>2.4208</td>
<td>30.9361</td>
<td>65.2977</td>
</tr>
<tr>
<td>5</td>
<td>349283745</td>
<td>3.2161</td>
<td>16.9305</td>
<td>73.3726</td>
</tr>
<tr>
<td>6</td>
<td>239485723</td>
<td>4.9361</td>
<td>33.5381</td>
<td>75.0014</td>
</tr>
<tr>
<td>7</td>
<td>849372834</td>
<td>1.9047</td>
<td>34.7731</td>
<td>58.7823</td>
</tr>
<tr>
<td>8</td>
<td>666777543</td>
<td>2.0539</td>
<td>20.3559</td>
<td>62.6038</td>
</tr>
<tr>
<td>9</td>
<td>987789878</td>
<td>9.1623</td>
<td>30.2585</td>
<td>58.3713</td>
</tr>
<tr>
<td>10</td>
<td>999666773</td>
<td>2.6995</td>
<td>19.2077</td>
<td>68.5982</td>
</tr>
<tr>
<td>Average</td>
<td>-</td>
<td>3.97359</td>
<td>28.19092</td>
<td>65.96966</td>
</tr>
<tr>
<td>Variance</td>
<td>-</td>
<td>4.534003835</td>
<td>55.88319388</td>
<td>29.03793065</td>
</tr>
<tr>
<td>Deviation</td>
<td>-</td>
<td>2.129320041</td>
<td>7.475506263</td>
<td>5.388685429</td>
</tr>
</tbody>
</table>

Table 4.1: Generational genetic algorithm first domain results.

![Figure 4.3: Generational genetic algorithm first domain results.](image-url)
Table 4.2: Generational genetic algorithm second domain results.

<table>
<thead>
<tr>
<th>Execution</th>
<th>Seed</th>
<th>Sphere</th>
<th>Rastrigin</th>
<th>Ackley</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>124321453</td>
<td>0.3764</td>
<td>15.6112</td>
<td>29.9994</td>
</tr>
<tr>
<td>2</td>
<td>983746234</td>
<td>0.2123</td>
<td>4.7917</td>
<td>26.3153</td>
</tr>
<tr>
<td>3</td>
<td>139548473</td>
<td>3.4899</td>
<td>9.4739</td>
<td>71.9414</td>
</tr>
<tr>
<td>4</td>
<td>394827384</td>
<td>1.7103</td>
<td>5.1283</td>
<td>53.9711</td>
</tr>
<tr>
<td>5</td>
<td>349283745</td>
<td>1.1381</td>
<td>5.5829</td>
<td>26.5755</td>
</tr>
<tr>
<td>6</td>
<td>239485723</td>
<td>1.1858</td>
<td>16.2791</td>
<td>46.7165</td>
</tr>
<tr>
<td>7</td>
<td>849372834</td>
<td>1.1675</td>
<td>15.4209</td>
<td>46.0527</td>
</tr>
<tr>
<td>8</td>
<td>66677543</td>
<td>2.3177</td>
<td>1.2838</td>
<td>36.1038</td>
</tr>
<tr>
<td>9</td>
<td>987789878</td>
<td>2.6048</td>
<td>8.3455</td>
<td>48.9161</td>
</tr>
<tr>
<td>10</td>
<td>999666773</td>
<td>0.4044</td>
<td>3.9908</td>
<td>-4.4409E-15</td>
</tr>
<tr>
<td>Average</td>
<td>-</td>
<td>1.46072</td>
<td>8.59081</td>
<td>38.65918</td>
</tr>
<tr>
<td>Variance</td>
<td>-</td>
<td>1.033895236</td>
<td>26.59080552</td>
<td>344.1390664</td>
</tr>
<tr>
<td>Deviation</td>
<td>-</td>
<td>1.01680639</td>
<td>5.15662734</td>
<td>18.55098559</td>
</tr>
</tbody>
</table>

Figure 4.4: Generational genetic algorithm second domain results.
### 4.5 Results

#### 4.5.2 Steady state genetic algorithm

The results obtained for the first domain are shown in Table 4.3. Analyzing these results, it can be noticed that the algorithm obtains the best average value for the Sphere function (closest to zero). The Y axis represents the values, whereas the X axis states for the different seeds used. The steady state genetic algorithm obtains better results than those obtained by the generational genetic algorithm.

<table>
<thead>
<tr>
<th>Execution</th>
<th>Seed</th>
<th>Sphere</th>
<th>Rastrigin</th>
<th>Ackley</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>124321453</td>
<td>8.342E-2</td>
<td>3.881E-1</td>
<td>5.220E-1</td>
</tr>
<tr>
<td>2</td>
<td>983746234</td>
<td>1.644E-1</td>
<td>1.5277</td>
<td>7.048E-1</td>
</tr>
<tr>
<td>3</td>
<td>139548473</td>
<td>1.308E-1</td>
<td>3.2756</td>
<td>3.709E-1</td>
</tr>
<tr>
<td>4</td>
<td>394827384</td>
<td>2.1666E-1</td>
<td>5.3007E-1</td>
<td>3.7935E-1</td>
</tr>
<tr>
<td>5</td>
<td>349283745</td>
<td>2.018E-1</td>
<td>4.9019E-1</td>
<td>5.8337E-1</td>
</tr>
<tr>
<td>6</td>
<td>239485723</td>
<td>1.734E-1</td>
<td>3.4615E-1</td>
<td>5.1815E-1</td>
</tr>
<tr>
<td>7</td>
<td>849372843</td>
<td>1.8619E-1</td>
<td>4.0313E-1</td>
<td>4.2402E-1</td>
</tr>
<tr>
<td>8</td>
<td>666777543</td>
<td>6.8373E-2</td>
<td>5.4299E-1</td>
<td>7.5765E-1</td>
</tr>
<tr>
<td>9</td>
<td>987789878</td>
<td>3.1667E-1</td>
<td>7.5286E-1</td>
<td>3.6992E-1</td>
</tr>
<tr>
<td>10</td>
<td>999666773</td>
<td>2.5608E-1</td>
<td>1.5786</td>
<td>5.9038E-1</td>
</tr>
<tr>
<td>Average</td>
<td>-</td>
<td>0.1797944</td>
<td>0.983545</td>
<td>0.522065</td>
</tr>
<tr>
<td>Variance</td>
<td>-</td>
<td>0.005081487</td>
<td>0.769769151</td>
<td>0.017331785</td>
</tr>
<tr>
<td>Deviation</td>
<td>-</td>
<td>0.071284548</td>
<td>0.87736489</td>
<td>0.131650236</td>
</tr>
</tbody>
</table>

Table 4.3: Steady state genetic algorithm first domain results.

![Figure 4.5: Steady state genetic algorithm first domain results.](image)

The results obtained for the second domain are shown in Table 4.4. Analyzing these results, it can be noticed that the Rastrigin function obtains the best average value (closest
to zero). In this domain, the Sphere function obtains the worst average value. The Y axis represents value, whereas the X axis states for the different seeds used.

<table>
<thead>
<tr>
<th>Seed</th>
<th>Sphere</th>
<th>Rastrigin</th>
<th>Ackley</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution 1</td>
<td>124321453</td>
<td>1.2654E-3</td>
<td>1.9723E-1</td>
</tr>
<tr>
<td>Execution 2</td>
<td>983746234</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Execution 3</td>
<td>139548473</td>
<td>1.9305E-2</td>
<td>2.1723E-1</td>
</tr>
<tr>
<td>Execution 4</td>
<td>394827384</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Execution 5</td>
<td>349283745</td>
<td>0.0</td>
<td>3.5139E-1</td>
</tr>
<tr>
<td>Execution 6</td>
<td>239485723</td>
<td>1.0798E-3</td>
<td>0.0</td>
</tr>
<tr>
<td>Execution 7</td>
<td>849372834</td>
<td>0.0</td>
<td>2.6467E-1</td>
</tr>
<tr>
<td>Execution 8</td>
<td>666775437</td>
<td>2.5699E-3</td>
<td>0.0</td>
</tr>
<tr>
<td>Execution 9</td>
<td>987789878</td>
<td>2.31447E-1</td>
<td>0.0</td>
</tr>
<tr>
<td>Execution 10</td>
<td>999666773</td>
<td>1.6122E-1</td>
<td>3.5779E-1</td>
</tr>
<tr>
<td>Average</td>
<td>-</td>
<td>0.04168871</td>
<td>0.137667</td>
</tr>
<tr>
<td>Variance</td>
<td>-</td>
<td>10.40784841</td>
<td>0.020991148</td>
</tr>
<tr>
<td>Deviation</td>
<td>-</td>
<td>3.226119715</td>
<td>0.144883222</td>
</tr>
</tbody>
</table>

Table 4.4: Steady state genetic algorithm second domain results.

![Graph showing Fitness vs Execution with three functions: Sphere, Rastrigin, and Ackley](image)

Figure 4.6: Steady state genetic algorithm second domain results.

### 4.5.3 Multimodal algorithms

In this section, the evaluation of two multimodal algorithms (Sharing and Clearing) is carried out.
4.5 Results

- **Sharing algorithm**

The results obtained for the first domain are shown in the Table 4.5. Analyzing these results, it can be noticed that the Sphere function obtains the best average value (closest to zero). The Y axis represents the values, whereas the X axis states for the different seeds used.

<table>
<thead>
<tr>
<th>Execution</th>
<th>Seed</th>
<th>Sphere</th>
<th>Rastrigin</th>
<th>Ackley</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution 1</td>
<td>124321453</td>
<td>1.0496</td>
<td>2.1321</td>
<td>14.3836</td>
</tr>
<tr>
<td>Execution 2</td>
<td>983746234</td>
<td>7.8644E-1</td>
<td>1.0451</td>
<td>2.8105</td>
</tr>
<tr>
<td>Execution 3</td>
<td>139548473</td>
<td>1.3113</td>
<td>6.4643</td>
<td>19.2538</td>
</tr>
<tr>
<td>Execution 4</td>
<td>394827384</td>
<td>5.993E-1</td>
<td>4.4826</td>
<td>10.0077</td>
</tr>
<tr>
<td>Execution 5</td>
<td>349283745</td>
<td>2.6441E-1</td>
<td>9.2055</td>
<td>12.6129</td>
</tr>
<tr>
<td>Execution 6</td>
<td>239485723</td>
<td>2.7920E-1</td>
<td>6.2493</td>
<td>9.3918E-1</td>
</tr>
<tr>
<td>Execution 7</td>
<td>849372834</td>
<td>2.6345E-1</td>
<td>4.2619</td>
<td>9.1324</td>
</tr>
<tr>
<td>Execution 8</td>
<td>66677543</td>
<td>2.6279E-1</td>
<td>2.3551</td>
<td>1.1368</td>
</tr>
<tr>
<td>Execution 9</td>
<td>987789878</td>
<td>5.2489E-1</td>
<td>3.5648</td>
<td>25.8075</td>
</tr>
<tr>
<td>Execution 10</td>
<td>999666773</td>
<td>7.8863E-1</td>
<td>1.8636</td>
<td>14.1554</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Average</th>
<th>Sphere</th>
<th>Rastrigin</th>
<th>Ackley</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-</td>
<td>0.613001</td>
<td>4.1624</td>
<td>11.023978</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>0.122489052</td>
<td>5.794679898</td>
<td>58.14315784</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>0.349984359</td>
<td>2.407214136</td>
<td>7.625166086</td>
</tr>
</tbody>
</table>

Table 4.5: Sharing algorithm first domain results.

![Figure 4.7: Sharing algorithm first domain results.](image)

The results obtained for the second domain are shown in Table 4.6. Analyzing these results, it can be noticed that the Sphere function obtains the best average value (closest to zero). Furthermore, the Rastrigin function obtains good results. In this domain, the
Genetic algorithms using real encoding: Optimization of real functions

Ackley function obtains the worst average value. The Y axis represents value, whereas the X axis states for the different seeds used.

<table>
<thead>
<tr>
<th>Execution</th>
<th>Seed</th>
<th>Sphere</th>
<th>Rastrigin</th>
<th>Ackley</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution 1</td>
<td>124321453</td>
<td>1.0486</td>
<td>1.1569</td>
<td>2.8408</td>
</tr>
<tr>
<td>Execution 2</td>
<td>983746234</td>
<td>7.8643E-1</td>
<td>8.6774E-1</td>
<td>2.7949</td>
</tr>
<tr>
<td>Execution 3</td>
<td>139548473</td>
<td>1.3107</td>
<td>1.4462</td>
<td>3.7725</td>
</tr>
<tr>
<td>Execution 4</td>
<td>394827384</td>
<td>5.2429E-1</td>
<td>5.7849E-1</td>
<td>1.8633</td>
</tr>
<tr>
<td>Execution 5</td>
<td>349283745</td>
<td>2.6214E-1</td>
<td>2.8925E-1</td>
<td>9.3166E-1</td>
</tr>
<tr>
<td>Execution 6</td>
<td>239485723</td>
<td>2.6214E-1</td>
<td>2.8925E-1</td>
<td>9.3166E-1</td>
</tr>
<tr>
<td>Execution 7</td>
<td>849372834</td>
<td>2.6214E-1</td>
<td>2.8925E-1</td>
<td>9.3166E-1</td>
</tr>
<tr>
<td>Execution 8</td>
<td>666777543</td>
<td>2.6214E-1</td>
<td>2.8925E-1</td>
<td>9.3166E-1</td>
</tr>
<tr>
<td>Execution 9</td>
<td>987789878</td>
<td>5.2429E-1</td>
<td>5.7849E-1</td>
<td>2.7949</td>
</tr>
<tr>
<td>Execution 10</td>
<td>999666773</td>
<td>7.86432E-1</td>
<td>8.6774E-1</td>
<td>14.1554</td>
</tr>
</tbody>
</table>

Average: -
Variance: 0.6029302
Deviation: 0.665256

Table 4.6: Sharing algorithm second domain results.

![Figure 4.8: Sharing algorithm second domain results.](image)

- Clearing algorithm

The results obtained for the first domain are shown in the next table. Analyzing these results, it can be noticed that the Sphere function obtains the best average value (closest to zero). The Y axis represents the values, whereas the X axis states for the different seeds used.
4.5 Results

<table>
<thead>
<tr>
<th>Seed</th>
<th>Sphere</th>
<th>Rastrigin</th>
<th>Ackley</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution 1</td>
<td>124321453</td>
<td>1.0486</td>
<td>4.1120</td>
</tr>
<tr>
<td>Execution 2</td>
<td>983746234</td>
<td>1.0486</td>
<td>1.1569</td>
</tr>
<tr>
<td>Execution 3</td>
<td>139548473</td>
<td>1.3107</td>
<td>3.3864</td>
</tr>
<tr>
<td>Execution 4</td>
<td>394827384</td>
<td>5.2429E-1</td>
<td>5.7849E-1</td>
</tr>
<tr>
<td>Execution 5</td>
<td>349283745</td>
<td>2.6214E-1</td>
<td>2.2692</td>
</tr>
<tr>
<td>Execution 6</td>
<td>239485723</td>
<td>2.6214E-1</td>
<td>1.2842</td>
</tr>
<tr>
<td>Execution 7</td>
<td>849372834</td>
<td>2.6214E-1</td>
<td>3.2642</td>
</tr>
<tr>
<td>Execution 8</td>
<td>666777543</td>
<td>2.7929E-1</td>
<td>2.4327</td>
</tr>
<tr>
<td>Execution 9</td>
<td>987789878</td>
<td>5.2429E-1</td>
<td>3.5534</td>
</tr>
<tr>
<td>Execution 10</td>
<td>999666773</td>
<td>7.8643E-1</td>
<td>1.8628</td>
</tr>
</tbody>
</table>

| Average | 0.630862 | 2.390029 | 2.113443 |
| Variance | 0.138957698 | 1.23950756 | 1.108405319 |
| Deviation | 0.372770301 | 1.113331741 | 1.052808301 |

Table 4.7: Clearing algorithm first domain results.

The results obtained for the second domain are shown in the next table. Analyzing these results, it can be noticed that the Sphere function obtains the best average value. The Y axis represents value, whereas the X axis states for the different seeds used.

4.5.4 Global results

In this section, several summary tables that allow us to make a comparative study are presented. The results obtained with the Sphere function for the first domain are shown in Table 4.9:
Table 4.8: Clearing algorithm second domain results.

![Figure 4.10: Clearing algorithm second domain results.](image)

Table 4.9: Sphere results summary table for the first domain.

The results obtained using the Sphere function for the first domain show that the best method, the one with the minimal average and deviation, is the Steady State algorithm.
This algorithm obtains the best individual and the best worst individual.

The results obtained with Sphere function and the second domain are in Table 4.10:

<table>
<thead>
<tr>
<th>Method</th>
<th>Average</th>
<th>Deviation</th>
<th>Best</th>
<th>Worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generational</td>
<td>1.46072</td>
<td>1.01680639</td>
<td>0.2123</td>
<td>3.4899</td>
</tr>
<tr>
<td>S. State</td>
<td>1.12260173</td>
<td>3.226119715</td>
<td>0.0</td>
<td>1.08E-01</td>
</tr>
<tr>
<td>Sharing</td>
<td>0.6029302</td>
<td>0.352679054</td>
<td>5.2429E-1</td>
<td>1.3107</td>
</tr>
<tr>
<td>Clearing</td>
<td>0.630862</td>
<td>0.372770301</td>
<td>2.6214E-1</td>
<td>1.3107</td>
</tr>
</tbody>
</table>

Table 4.10: Sphere results summary table for the second domain.

The results obtained using the Sphere function (see Table 4.10) for the second domain show that the best method, the one with the minimal average and deviation, is the Sharing multimodal algorithm. However, the algorithm with the best individual and the best worst individual is the Steady State.

<table>
<thead>
<tr>
<th>Method</th>
<th>Average</th>
<th>Deviation</th>
<th>Best</th>
<th>Worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generational</td>
<td>28.19092</td>
<td>7.475506263</td>
<td>16.9305</td>
<td>41.5819</td>
</tr>
<tr>
<td>S. State</td>
<td>0.983545</td>
<td>0.87736489</td>
<td>3.4615E-1</td>
<td>3.2756</td>
</tr>
<tr>
<td>Sharing</td>
<td>4.16243</td>
<td>2.407214136</td>
<td>1.0451</td>
<td>9.2055</td>
</tr>
<tr>
<td>Clearing</td>
<td>2.390029</td>
<td>1.113331741</td>
<td>5.7849E-1</td>
<td>4.1120</td>
</tr>
</tbody>
</table>

Table 4.11: Rastrigin results summary table for the first domain.

The results obtained using the Rastrigin function (see Table 4.11) for the first domain show that the best method, the one with the minimal average and deviation, is the Steady State algorithm. This algorithm obtains the best individual and the best worst individual.

<table>
<thead>
<tr>
<th>Method</th>
<th>Average</th>
<th>Deviation</th>
<th>Best</th>
<th>Worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generational</td>
<td>8.59081</td>
<td>5.15662734</td>
<td>1.2838</td>
<td>16.2791</td>
</tr>
<tr>
<td>S. State</td>
<td>0.137667</td>
<td>0.144883222</td>
<td>0.0</td>
<td>3.5139E-1</td>
</tr>
<tr>
<td>Sharing</td>
<td>0.665256</td>
<td>0.389122811</td>
<td>2.8925E-1</td>
<td>1.4462</td>
</tr>
<tr>
<td>Clearing</td>
<td>0.709527</td>
<td>0.400423103</td>
<td>2.8925E-1</td>
<td>1.1569</td>
</tr>
</tbody>
</table>

Table 4.12: Rastrigin results summary table for the second domain.

The results obtained using the Rastrigin function for the second domain (see Table 4.12) show that the best method, the one with the minimal average and deviation, is the Steady State algorithm. This algorithm obtains the best individual and the best worst individual.

The results obtained using the Ackley function for the first domain (see Table 4.13)
show that the best method, the one with the minimal average and deviation, is the Steady State algorithm. This algorithm obtains the best individual and the best worst individual.

Table 4.14: Ackley results summary table for the second domain.

The results obtained using the Ackley function for the second domain (see Table 4.14) show that the best method, the one with the minimal average and deviation, is the Steady State algorithm. This algorithm obtains the best individual and the best worst individual.

### 4.5.5 Generational vs Steady State

In this section, a study between the Generational and the Steady State algorithms is presented. The results shown in the next table are the average results for 10 seeds.

Table 4.15: Generational vs Steady State results.

Analyzing the results, notice that for all the functions and domains studied, the steady state algorithm obtains better results than the generational algorithm. Remember that the generational algorithm replaces completely the individuals in the population, obtaining different values.


4.5 Results

4.5.6 Sharing vs Clearing

In this section, a study between the Sharing and Clearing multimodal algorithms is presented. The results shown in the next table are the average results for 10 seeds.

<table>
<thead>
<tr>
<th>Function</th>
<th>Sharing</th>
<th>Clearing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sphere-domain 1</td>
<td>0.613001</td>
<td>0.630862</td>
</tr>
<tr>
<td>Sphere-domain 2</td>
<td>0.602930</td>
<td>0.630862</td>
</tr>
<tr>
<td>Rastrigin-domain 1</td>
<td>4.162430</td>
<td>2.390029</td>
</tr>
<tr>
<td>Rastrigin-domain 2</td>
<td>0.665256</td>
<td>0.709527</td>
</tr>
<tr>
<td>Ackley-domain 1</td>
<td>11.023978</td>
<td>2.113443</td>
</tr>
<tr>
<td>Ackley-domain 2</td>
<td>3.194844</td>
<td>2.111918</td>
</tr>
</tbody>
</table>

Table 4.16: Sharing vs Clearing results.

Analyzing these results, notice that Clearing algorithm is better than Sharing algorithm. This is because, in this algorithm, the selection process is much more powerful since individuals dominated by others are eliminated. However, the Sharing algorithm, makes a comparison between all individuals in the population without eliminating anyone for selection.

4.5.7 General comparison

In this section, a general study with the different algorithms (Generational, Steady State, Sharing and Clearing) is presented. The results shown in the next table are the average results for 10 seeds.

<table>
<thead>
<tr>
<th>Function</th>
<th>Generational</th>
<th>S. State</th>
<th>Sharing</th>
<th>Clearing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sphere-domain 1</td>
<td>3.97359</td>
<td>0.1797944</td>
<td>3.97359</td>
<td>0.1797944</td>
</tr>
<tr>
<td>Sphere-domain 2</td>
<td>1.46072</td>
<td>1.12260173</td>
<td>1.46072</td>
<td>1.12260173</td>
</tr>
<tr>
<td>Rastrigin-domain 1</td>
<td>28.19092</td>
<td>0.983545</td>
<td>28.19092</td>
<td>0.983545</td>
</tr>
<tr>
<td>Rastrigin-domain 2</td>
<td>8.59081</td>
<td>0.137667</td>
<td>8.59081</td>
<td>0.137667</td>
</tr>
<tr>
<td>Ackley-domain 1</td>
<td>65.96696</td>
<td>0.522065</td>
<td>65.96696</td>
<td>0.522065</td>
</tr>
<tr>
<td>Ackley-domain 2</td>
<td>38.65918</td>
<td>0.2208647</td>
<td>38.65918</td>
<td>0.2208647</td>
</tr>
</tbody>
</table>

Table 4.17: General comparison summary.
4.6 Study of effect of the alpha value in BLX-\(\alpha\) crossover operator

This section studies how the \(\alpha\) value modification affects in the BLX-\(\alpha\) crossover operator. The results using a simple generational algorithm and the sphere function are depicted in Figure 4.11, notice that the higher the \(\alpha\) value, the worse the results. The \(Y\) axis represents the values, whereas the \(X\) axis states for the different \(\alpha\) values used.

Figure 4.11: Alpha BLX-\(\alpha\) value study results.
5. GENETIC PROGRAMMING: A SYMBOLIC REGRESSION PROBLEM

5.1 Introduction

This section aims to explain how to solve symbolic regression problems applying Genetic Programming (GP) algorithms, and it also studies the performance of these algorithms according to their parameterization setup. The results obtained can be used as guidelines to suggest the most effective way to set up the parameters for other GP-based experiments.

The main difference with respect to GAs lies in the fact that in GP it must be shown how solutions are represented, and therefore, the user has to define a suitable terminal and a function set (notice that expressions in GP are not typed, and any function can take any terminal or any result of another function as argument). Thus, the purpose of a GP algorithm is to evolve a measurable expression or a program that fits such representation.

Grammar Guided Genetic Programming (GGGP) differs from conventional GP in the use of a context-free grammar that guides the search of expressions, restricting the search space and ensuring that any expression found will be valid regarding this grammar. However, this encoding will not be used to solve symbolic regression problems, because conventional GP is more appropriate for solving this kind of problems.

JCLEC library can deal both with conventional GP representation –expression tree encoding– and GGGP –syntax tree encoding–. The former is supported in the use of the net.sf.jclec.exprtree package, which defines the encoding of the individuals, their structure and operators to manipulate them. The latter is implemented in the net.sf.jclec.syntaxtree package of the library.

5.2 Problem definition

The goal of symbolic regression is to obtain a symbolic expression that represents a mathematical function. In particular, this tutorial explains how to approximate the following functions: \( f(x) = x^4 + x^3 + x^2 + x \) and \( f(x) = x^4 + x^3 + x^2 + x + 1 \). For this purpose, the evaluator to implement will employ a set of pairs (input,output) for both functions, as shown in Table 5.1:
56 Genetic programming: a symbolic regression problem

\[
f(x) = x^4 + x^3 + x^2 + x
\]

<table>
<thead>
<tr>
<th>x</th>
<th>f(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
<td>10</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>30</td>
</tr>
</tbody>
</table>

\[
f(x) = x^4 + x^3 + x^2 + x + 1
\]

<table>
<thead>
<tr>
<th>x</th>
<th>f(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
<td>9</td>
</tr>
<tr>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>31</td>
</tr>
</tbody>
</table>

Table 5.1: Functions to approximate

5.3 Implementation

This tutorial uses the \textit{net.sf.jclec.exprtree} package. The efficiency of this package is supported in the use of a virtual machine that uses a stack to calculate the results of intermediate operations:

- Instead of using a preorder route to evaluate each node of the tree, nodes are added to the stack (from right to left and from bottom to top, which is not exactly a postfix tree-traversal).

- When the last node added is a non-terminal node, the stack evaluates the function, which will affect to the two elements that were introduced previously in the stack (these argument nodes can be either terminals or evaluation results from other functions). For example, the stack contains the values 3.2453 and 2.4532. The next symbol to analyze is the non-terminal node “+”. The stack values 3.2453 and 2.4532 are popped. These values are added and then the result 5.6985 is pushed into the stack.

First, it is necessary to select the terminal and function set necessary to solve the problem. The operators \(+\), \(-\), \(*\) will be used as the function set and the terminal \(x\) as the terminal set. Thus, one class per symbol is implemented in a new sub-package, e.g. \textit{net.sf.jclec.tutorial.symreg}. Next it is shown the code of these classes:

- Class \textit{X}. This class represents a double terminal symbol.

```java
public class X extends Argument<Double>
{
  public X()
  {
    super(Double.class, 0);
  }
}
```
5.3 Implementation

```java
// java.lang.Object methods
public boolean equals(Object other)
{
    return other instanceof X;
}

public String toString()
{
    return "X";
}
```

- Class `Add`. This class sums two arguments. Its `evaluate()` method pops the two top arguments of the stack, sums them and pushes the result back to the stack.

```java
public class Add extends AbstractPrimitive
{
    /**
     * This operator receives two double arrays as arguments and return
     * a double array as result.
     */
    public Add()
    {
        super(new Class<?>[] {Double.class, Double.class}, Double.class);
    }

    @Override
    protected void evaluate(ExprTreeFunction context)
    {
        // Get arguments (in context stack)
        Double arg1 = pop(context);
        Double arg2 = pop(context);
        // Push result in context stack
        push(context, arg1+arg2);
    }

    // java.lang.Object methods
    public boolean equals(Object other)
    {
        return other instanceof Add;
    }

    public String toString()
    {
        return "+";
    }
}
```
• Class Sub. This class subtracts two arguments. Its evaluate() method pops the two top arguments of the stack, subtracks them ant pushes the result back to the stack.

```java
public class Sub extends AbstractPrimitive {
    /**
     * This operator receives two double arrays as arguments and return
     * a double array as result.
     */
    public Sub()
    {
        super(new Class<?>[] {Double.class, Double.class}, Double.class);
    }

    @Override
    protected void evaluate(ExprTreeFunction context)
    {
        // Get arguments (in context stack)
        Double arg1 = pop(context);
        Double arg2 = pop(context);
        // Push result in context stack
        push(context, arg1 - arg2);
    }

    // java.lang.Object methods
    public boolean equals(Object other)
    {
        return other instanceof Sub;
    }

    public String toString()
    {
        return "-";
    }
}
```

• Class Mul. This class multiplies two arguments. Its evaluate() method pops the two top arguments of the stack, multiplies them ant pushes the result back to the stack.

```java
package tutorial.symr
public class Mul extends AbstractPrimitive {
    /**
     * This operator receives two double arrays as arguments and return
     * a double array as result.
     */
    public Mul()
    {
    }
}
```
super(new Class<?>[] {Double.class, Double.class, Double.class});

@Override
protected void evaluate(ExprTreeFunction context)
{
    // Get arguments (in context stack)
    Double arg1 = pop(context);
    Double arg2 = pop(context);
    // Push result in context stack
    push(context, arg1*arg2);
}

// java.lang.Object methods
public boolean equals(Object other)
{
    return other instanceof Mul;
}

public String toString()
{
    return "*";
}

Notice that the previous classes are primitives, and therefore, they must implement the `evaluate()` method. In case of the class Terminal, the `evaluate()` method just push in the stack the terminal. In contrast, in the last three classes, this method is responsible for putting in the evaluation stack the value returned by the primitive when applying the function to the last two arguments of the stack.

Once the aforementioned classes have been defined, it is necessary to design the evaluator to compute the fitness of each individual. Thus, a new class is created, `SymRegEvaluator`, and this class is included in the package `net.sf.jclec.tutorial.symreg`. In the case of the first function, a training set composed of two arrays is defined, `xvalues` and `yvalues`. Then, for each `xvalues[i]` the method `execute()` of `ExprTreeFunction` is called, finding out the difference between this result and the real one (`yvalues[i]`). The squared difference of each element as accumulated as error, and the square root of the final value will be the fitness of the individual (notice that the objective is to minimize the error, i.e., the smaller is the value, the better is the individual).
Finally, this section describes how to encode the configuration file required to run the algorithm in the JCLEC library.
• Firstly, it is necessary to select a genetic algorithm to solve the problem.

```xml
<process algorithm-type="net.sf.jclec.algorithm.classic.SGE"/>
```

• **Coding**: in GP, individuals have an specific representation, adopting the shape of expression trees. Thus, the package `net.sf.jclec.exprtree` must be used, establishing the minimum tree size, the maximum tree size and the list of terminal symbols and functions.

```xml
<species type="net.sf.jclec.exprtree.ExprTreeIndividualSpecies" number-of-trees="1">
  <expression-tree>
    <min-tree-size>3</min-tree-size>
    <max-tree-size>25</max-tree-size>
    <root-type>java.lang.Double</root-type>
    <terminals>
      <terminal class="tutorial.symreg.X"/>
    </terminals>
    <functions>
      <function class="tutorial.symreg.Add"/>
      <function class="tutorial.symreg.Sub"/>
      <function class="tutorial.symreg.Mul"/>
    </functions>
  </expression-tree>
</species>
```

• **Initialization**: the population is randomly initialized by using expression trees and employing the class `net.sf.jclec.exprtree.ExprTreeCreator`.

```xml
<provider type="net.sf.jclec.exprtree.ExprTreeCreator"/>
```

• **Stopping criterion**: this is set to the maximum number of generations.

```xml
<max-of-generations>100</max-of-generations>
```

• **Fitness function**: This step involves implementing an appropriate evaluator, according to the characteristics determined to evaluate the population and treat unfeasible individuals. This class will extend from `AbstractEvaluator`, located in the package `net.sf.jclec.base`.

```xml
<evaluator type="tutorial.symreg.SymregEvaluator"/>
```

• **Selection criterion** of the parents can be set by using the `net.sf.jclec.selector` package. The selector is implemented by a tournament of size two.

```xml
<parents-selector type="net.sf.jclec.selector.TournamentSelector" tournament-size="2"/>
```

• **Genetic operators**. The package `net.sf.jclec.exprtree` contains mutation and crossover operators ready to use. `SubtreeCrossover` and `SubtreeMutator` will be selected.
Genetic programming: a symbolic regression problem

- **Parameters.** With respect to other parameters, it is necessary to define the size of the population and the seed for the random number generator.

```
<population-size>200</population-size>
<rand-gen-factory type="net.sf.jclec.util.random.RanecuFactory" seed="234567895"/>
```

Finally, it is necessary to determine the listener to be used for displaying results.

```
<listener type="net.sf.jclec.tutorial.PopulationReporter">
<report-title>SymReg</report-title>
<report-frequency>1</report-frequency>
<report-on-console>true</report-on-console>
<report-on-file>false</report-on-file>
<save-complete-population>false</save-complete-population>
</listener>
```

Below it is shown the whole XML configuration file.

```
<experiment>
<process alg="tutorial.symreg">
<process algorithm="net.sf.jclec.algorithm.classic.SGE">
<population-size>200</population-size>
<max-of-generations>100</max-of-generations>
<rand-gen-factory type="net.sf.jclec.util.random.RanecuFactory" seed="234567895"/>
<species type="net.sf.jclec.exprtree.ExprTreeIndividualSpecies" number-of-trees="1">
<expression-tree>
<min-tree-size>1</min-tree-size>
<max-tree-size>25</max-tree-size>
<root-type>java.lang.Double</root-type>
<terminals>
<terminal class="tutorial.symreg.X"/>
</terminals>
</expression-tree>
</species>
<evaluator type="tutorial.symreg.SymRegEvaluator"/>
<provider type="net.sf.jclec.exprtree.ExprTreeCreator"/>
<parents-selector type="tutorial.symreg.TournamentSelector" tournament-size="2"/>
<recombinator type="net.sf.jclec.exprtree.ExprTreeRecombinator" rec-prob="0.8"/>
```

```
5.4 Results

Using the configuration file from above, the parameters can be adjusted, to find out which crossover and mutation operators provide the best results. It is recommended to use a low value for mutation probability, and a high one for crossover. One can play with these values and check out the fitness variance. Because they are stochastic algorithms, it is necessary to run the algorithm several times using different seed values.

Tables 5.1 and 5.2 shows the average fitness results obtained for the two crossover operators considered with several probabilities, testing each probability with ten different seeds and also considering the SubTreeMutator operator with a mutation probability of 0.1. It can be seen that for both operators the results obtained are closely to the optimal values, and the best ones are reached with a crossover probability value comprised between 0.7 and 0.9. However, the SubTreeCrossover operator generally performs better than the TreeCrossover operator.

Experiments were also performed to figure out which mutation operator performs better, considering a fixed mutation probability of 0.1, ten different seeds, and now taking as crossover operator the SubTreeCrossover with a probability of 0.7. The decision of setting the mutation probability to 0.1 is based in the fact that removing the mutation in the evolutive process leads to poor results, because no genetic diversity is introduced in the new populations. However, setting this parameter to a value greater than 0.1 makes the algorithm to behave in a similar way as a random search.

The average results obtained for each mutator are summarized in Table 5.4.
### Table 5.2: SubTreeCrossover probabilities results.

<table>
<thead>
<tr>
<th>Execution</th>
<th>Seed</th>
<th>0.1</th>
<th>0.3</th>
<th>0.5</th>
<th>0.7</th>
<th>0.9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution 1</td>
<td>123456789</td>
<td>13.0384</td>
<td>15.5563</td>
<td>5.2915</td>
<td>3.1623</td>
<td>5.8310</td>
</tr>
<tr>
<td>Execution 2</td>
<td>234567891</td>
<td>10.2956</td>
<td>6.6332</td>
<td>3.1623</td>
<td>5.2915</td>
<td>3.1623</td>
</tr>
<tr>
<td>Execution 3</td>
<td>345678912</td>
<td>14.4222</td>
<td>6.6332</td>
<td>6.6332</td>
<td>3.1623</td>
<td>0.0000</td>
</tr>
<tr>
<td>Execution 4</td>
<td>456789123</td>
<td>10.2956</td>
<td>6.6332</td>
<td>3.1623</td>
<td>3.1623</td>
<td>3.1623</td>
</tr>
<tr>
<td>Execution 5</td>
<td>567891234</td>
<td>10.2956</td>
<td>4.0000</td>
<td>3.1623</td>
<td>6.6332</td>
<td>0.0000</td>
</tr>
<tr>
<td>Execution 6</td>
<td>678912345</td>
<td>8.6023</td>
<td>7.2111</td>
<td>5.8310</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Execution 9</td>
<td>912345678</td>
<td>10.2956</td>
<td>3.1623</td>
<td>0.0000</td>
<td>3.1623</td>
<td>4.2426</td>
</tr>
<tr>
<td>Execution 10</td>
<td>123571113</td>
<td>10.2956</td>
<td>8.6023</td>
<td>5.8310</td>
<td>7.4833</td>
<td>7.4833</td>
</tr>
<tr>
<td>Average</td>
<td>-</td>
<td>10.4470</td>
<td>7.1698</td>
<td>4.5538</td>
<td>3.8382</td>
<td>2.7044</td>
</tr>
</tbody>
</table>

### Table 5.3: TreeCrossover probabilities results.

<table>
<thead>
<tr>
<th>Execution</th>
<th>Seed</th>
<th>0.1</th>
<th>0.3</th>
<th>0.5</th>
<th>0.7</th>
<th>0.9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution 1</td>
<td>123456789</td>
<td>13.0384</td>
<td>6.6332</td>
<td>6.6332</td>
<td>6.6332</td>
<td>6.6332</td>
</tr>
<tr>
<td>Execution 6</td>
<td>678912345</td>
<td>10.2956</td>
<td>10.2956</td>
<td>6.6332</td>
<td>6.6332</td>
<td>7.0711</td>
</tr>
<tr>
<td>Execution 7</td>
<td>789123456</td>
<td>8.2462</td>
<td>7.2111</td>
<td>7.2111</td>
<td>7.0711</td>
<td>6.6332</td>
</tr>
<tr>
<td>Execution 8</td>
<td>891234567</td>
<td>6.6332</td>
<td>6.6332</td>
<td>5.2915</td>
<td>15.5563</td>
<td>10.5830</td>
</tr>
<tr>
<td>Execution 9</td>
<td>912345678</td>
<td>10.2956</td>
<td>6.6332</td>
<td>6.6332</td>
<td>5.8310</td>
<td>7.0711</td>
</tr>
<tr>
<td>Execution 10</td>
<td>123571113</td>
<td>10.2956</td>
<td>7.4833</td>
<td>7.4833</td>
<td>6.6332</td>
<td>7.4833</td>
</tr>
<tr>
<td>Average</td>
<td>-</td>
<td>10.0451</td>
<td>7.5085</td>
<td>6.6856</td>
<td>7.8553</td>
<td>7.1699</td>
</tr>
</tbody>
</table>

As shown, the best results are those obtained when using the SubTreeCrossover and the AllNodesMutator operators.
Table 5.4: Mutators average results.