EvolutionarySearch: a brief guide for developers

Sebastián Luna Valero

This document explains the design of the package **EvolutionarySearch** in order ease its extension with new operators by the developers interested in doing so. That is why this document is intended for people with, at least a little, previous knowledge about *Evolutionary Computation (EC)*.

EvolutionarySearch is an official WEKA package that performs *attribute selection*. This new package implements an *Evolutionary Algorithm (EA)* that looks for a good subset of attributes in order to improve the classification accuracy when combined with supervised learning methods.

The EA encodes individuals (solutions to the problem) with a constant length vector of bits x (see Figure 1) where: $x(i) = 1$ denotes that the i-th attribute has been selected and $x(i) = 0$ means the opposite.

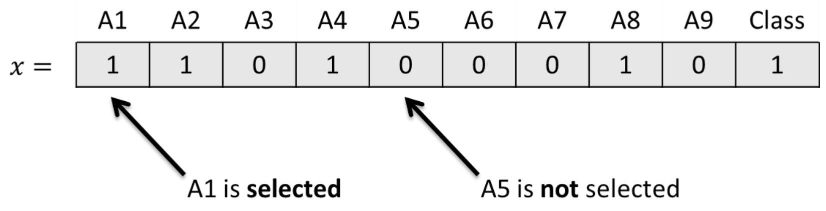

Figure 1. Binary encoding for individuals in the EA.

The version 1.0.0 of EvolutionarySearch presents the following basic operators:

- Random initialization
- Binary tournament selection
- Single point crossover
- Bit flip mutation, and
- Generational replacement with elitism (i.e., the best individual is always kept)

EvolutionarySearch: A basic UML class diagram

Figure 2 presents a simplified UML class diagram of EvolutionarySearch. Let us now describe each class separately and the relation among them.

EvolutionarySearch.java is the main class implementing the Evolutionary Algorithm. It uses all the available operators to evolve the population through generations. That is why all the remaining classes are (directly or indirectly) associated with this one.

The class Individual.java models the individuals in the population. Inspired in the GeneticSearch.java (implemented by Mark Hall) this class uses the BitSet.java provided by Java. BitSet is a vector of bits (0s or 1s) representing a solution in a Feature Selection Problem. In the same way, Population.java models a population of individuals that is used by the Evolutionary Algorithm.

Every operator inherits from the class Operator.java. However, depending on the type of operator each one inherits from a specific class: Initialization.java, Selection.java, Crossover.java, Mutation.java and Replacement.java. Figure 2 depicts the operators available in the version 1.0.0 of EvolutionarySearch: random initialization, binary tournament selection, single point crossover, bit flip mutation and generational replacement with elitism. In the next section we will see how to add a new mutation operator by using this class hierarchy.

Finally, EvolutionaryStatistics.java is an auxiliary class to gather all the statistics related to the search process and Random.java is the class provided by Java to generate random number sequences (located in the package java.util)

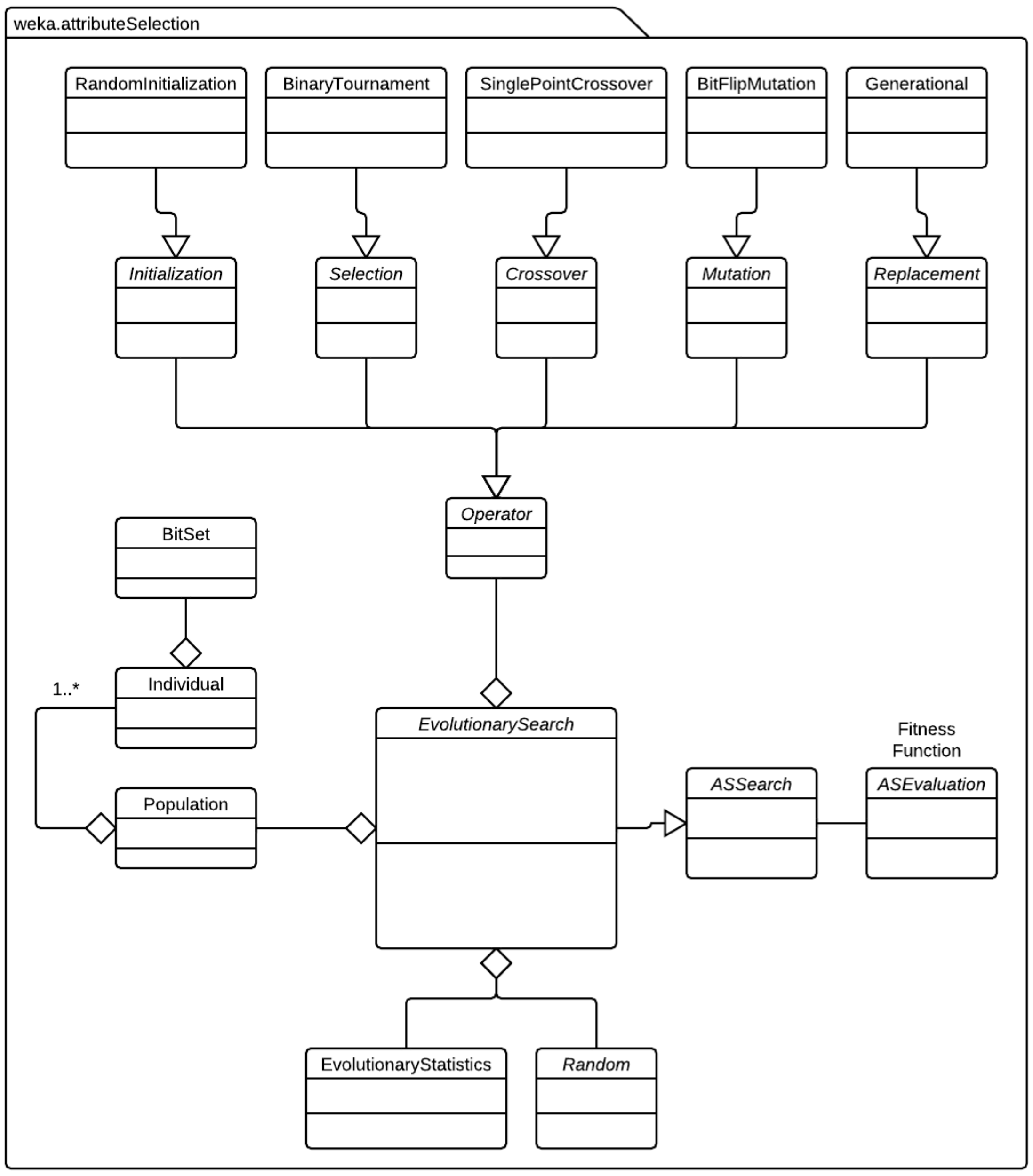

Figure 2: Simplified UML class diagram of EvolutionarySearch.

Adding new operators to EvolutionarySearch

If you want to add a new operator to perform a different initialization, selection, crossover, mutation or replacement you will need to extend the corresponding class.

Let us explain how to add a new operator by using an example. Suppose that we want to add a new mutation operator called *bit-off mutation*. This simple mutation works like the bit-flip mutation but, instead of flipping the value of a bit, it always reset the bit value to zero.

The first step is to create a new Java class (*BitOffMutation.java*) that *extends* the class called Mutation and then implement the new operator inside the *execute* method, inherited from the superclass (see Figure 3).

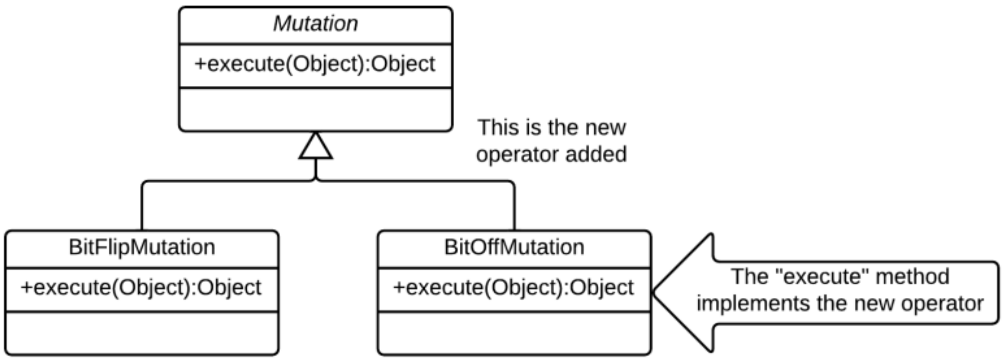

Figure 3: UML class diagram for the new mutation operator BitOffMutation.

The second step is to update the file *EvolutionarySearch.java*. First, include a new code for the new operator in the corresponding Tag. See the source code example below:

```
/** type of mutation */
private int mutationOp;

/** default operator */
protected static final int BIT_FLIP_MUTATION = 0;

/** new mutation operator */
protected static final int BIT_OFF_MUTATION = 1;

/** update the MUTATION_TAG */
public static final Tag [] MUTATION_TAG = {
        new Tag(BIT_FLIP_MUTATION, "bit-flip"),
        new Tag(BIT_OFF_MUTATION, “bit-off”)
};
```

Now it is time to update the method *initOperators* (also in EvolutionarySearch.java). Here, each operator is configured with the input parameters provided by the user. In the source code you have to include the following:

```
        // mutation operator
        parameters = new HashMap<String, Object>();

        if (mutationOp == BIT_OFF_MUTATION) {
                // configure your new operator
        } else {
                // bit-flip mutation is the default option
                // here you have how to configure it
                parameters.put("probability", mutationProbability);
                parameters.put("random", random);
                parameters.put("classIndex", classIndex);
                mutationOperator =
                  MutationFactory.getMutationOperator("BitFlipMutation",parameters);
        }

        parameters.clear();
```

Note that every time you add a new (mutation) operator you will need to add a new *if-then* clause to configure your new operator.

Although it is not mandatory, it is highly recommendable that you update the *toString* method in order to get your new operator printed out properly when necessary. To do that, the corresponding *switch* clause should be modified:

```
        result.append("\n\tMutation: ");
        switch (mutationOp) {
                case BIT_OFF_MUTATION: result.append("bit-off mutation");
                default: result.append("bit-flip mutation");
        }
```

In the same way, it is also recommendable to update the *javadoc* comments in the source code, so that the rest of users understand how the new operator works.

Finally, you have also to update the *getMutationOperator* method inside the class *MutationFactory.java* in the following way:

```
        if (name.equalsIgnoreCase("BitFlipMutation")) {
                return new BitFlipMutation(parameters);
        } else if (name.equalsIgnoreCase("BitOffMutation")) {
                return new BitOffMutation(parameters);
        } else {
                throw new EvolutionaryException("…");
        }
```