jmetal

Release 6.0-SNAPSHOT

Antonio J. Nebro

Sep 23, 2019

CONTENTS:

Section author: Antonio J. Nebro <ajnebro@uma.es>

jMetal is a Java-based framework for multi-objective optimization with metaheuristics. It is an open-source project released under MIT license, and it is hosted in GitHub: https://github.com/jMetal/jMetal. The current version is jMetal 5.8. The next major release of jMetal will be version 6.0, which is being developed in the `jmetal6` branch of the repository as version 6.0-SNAPSHOT.

This Web site is devoted to describing the new features of jMetal 6.0. Please, note that **this is an active development branch**, where some ideas are still not completely defined and many changes will take place until the version is closed. In its current state, this branch is not intended to newcomers but to users already familiarized with jMetal. It is worth mentioning that we have taken as starting point all the code included in jMetal 5.8, so this branch is fully functional and in fact we are currently using it in our research work. Comments and suggestions are welcome; we encourage interested users in contributing to open new issues in the project repository.

A summary of the incoming features of jMetal 6.0 are listed next:

- Use of Sphinx for the documentation. In jMetal 5 the documentation is based on Markdown files; it not complete and some parts are outdated.
- Support for automatic configuration of metaheuristics. We include a sub-project called `jmetal-auto`, which currently contains a version of NSGA-II that can be fully auto tuned using `irace`, as is described in the paper "Automatic Configuration of NSGA-II with jMetal and irace", presented at GECCO 2019 (DOI: https://doi.org/10.1145/3319619.3326832).
- Improved experimentation. The output of an experiment (i.e., the execution of a number of algorithms on a set of problems) is an CSV file which can be further analyzed to produce Latex tables and graphics with statistical information. We plan to use Tablesaw (https://github.com/jtablesaw/tablesaw) and Smile (http://haifengl.github.io/smile/) for the analysis tasks. All the experimentation code is located in sub-package `jmetal-lab`.
- All the core packages in `jmetal-core` (`solution`, `problem`, `algorithm`, `operator`, `quality indicator`), are being reviewed, tested, and refactored. Some relevant changes already addressed are:
  - All the classes related to solutions encodings depended on a problem (i.e., a sub-class of class `Problem`). This dependence has been removed, so a solution can be created without the need of a problem to do that.
- The implementation of most of the algorithms in `jmetal-algorithm` will be revised. A major issue here is that the current base classes for evolutionary algorithms were designed to be extended by using inheritance, while for the autoconfiguration of algorithms we have designed a new template based on delegation, which is more flexible and allows to fully configure an algorithm with a string of arguments. As for now, the only algorithm we are woking with is NSGA-II, so we have now different versions of it (the original one in sub-project `jmetal-algorithm` and two new versions in sub-project `jmetal-auto`), what can be confusing.

jMetal 6 is implemented in Java 8 and it is a Maven project structured in six sub-projects:

Sub-project	Contents
jmetal-core	Core classes
jmetal-solution	Solution encodings
jmetal-algorithm	Algorithm implementations
jmetal-problem	Bencharmk problems
jmetal-exec	Examples
jmetal-lab	Experimentation
jmetal-auto	Auto configuration

CHAPTER

ONE

INSTALLATION

jMetal is a Maven project hosted in GitHub, so there are two ways of getting the software: adding it as a dependence in your own Maven project, or getting the source code from https://github.com/jMetal/jMetal.

The software requirements to use jMetal 6 are:

- Java 8 JDK
- Maven
- R (optional)
- Latex (optional)

To work with the jMetal 6.0-SNAPSHOT version, just clone the repository in https://github.com/jMetal/jMetal and checkout the `jmetal6` branch.

CHAPTER

TWO

AUTO-CONFIGURATION OF EVOLUTIONARY ALGORITHMS: NSGA-II

Before reading this section, readers are referred to the paper "Automatic configuration of NSGA-II with jMetal and irace", presented in GECCO 2019 (DOI: https://doi.org/10.1145/3319619.3326832)

2.1 Motivation

A current trend in multi-objective optimization is to use automatic parameter configuration tools to find accurate settings of metaheuristics to effectively solve a number of problems. The idea is to avoid the traditional approach of carrying out a number of pilot tests, which are typically conducted without following a systematic strategy. In this context, an algorithm configuration is a complete assignment of values to all required parameters of the algorithm.

The auto-configuration tool we have selected is irace, an R package that implements an elitist iterated racing algorithm, where algorithm configurations are sampled from a sampling distribution, uniformly at random at the beginning, but biased towards the best configurations found in later iterations. At each iteration, the generated configurations and the "elite" ones from previous iterations are raced by evaluating them on training problem instances. A statistical test is used to decide which configurations should be eliminated from the race. When the race terminates, the surviving configurations become elites for the next iteration.

The issue we studied in the aforementioned paper is how to use jMetal combined with irace to allow the automatic configuration of multiobjective metaheuristics. As this is our first approximation to this matter, we decided to focus on NSGA-II and its use to solve continuous problems.

2.2 NSGA-II parameters

The standard NSGA-II is a generational evolutionary algorithm featured by using a ranking method based on Pareto ranking and the crowding distance density estimator, both in the selection and replacement steps. When it is used to solve continuous problems, NSGA-II adopts the simulated binary crossover (SBX) and the polynomial mutation. No external archive is included in NSGA-II. However, as we intend to configure NSGA-II in an automatic way, we need to relax the aforementioned features in order to have enough flexibility to modify the search capabilities of the algorithm. This way, we are going to consider that any multi-objective evolutionary algorithm with the typical parameters (selection, crossover, and mutation) and using ranking and crowding in the replacement step can be considered as a variant of NSGA-II.

The parameters or components of NSGA-II that can be adjusted are included in this table:

Parameter name	Allowed values
algorithmResult	*externalArchive*, *population*
populationSize	100
populationSizeWithArchive	10,20,50,100,200,400
offspringPopulationSize	1,5,10,20,50,100,200,400
createInitialSolutions	*random*, *latinHypercubeSampling*, *scatterSearch*
variation	*crossoverAndMutationVariation*
crossover	*SBX*, *BLX_Alpha*
crossoverProbability	[0.0, 1.0]
crossoverRepairStrategy	*random*, *round*, *bounds*
sbxCrossoverDistributionIndex	[5.0, 400.0]
blxAlphaCrossoverAlphaValue	[0.0, 1.0]
mutation	*uniform*, *polynomial*
mutationProbability	[0.0, 1.0]
mutationRepairStrategy	*random*, *round*, *bounds*
polynomialMutationDistributionIndex	[5.0, 400.0]
uniformMutationPerturbation	[0.0, 1.0]
selection	*random*, *tournament*
selectionTournamentSize	[2, 10]

Our *autoNSGAII* can optionally adopt an external archive to store the non-dominated solutions found during the search process. The archive size of bounded and the crowding distance estimator is used to remove solutions with the archive is full. Then, in case of using no archive, the result of the algorithm is the population, which is configured with the *populationSize* parameter; otherwise, the output is the external archive, whose maximum size is *populationSize* parameter value, but then the population size can be tuned by taking values from the set (10, 20, 50, 100, 200, 400).

In the classical NSGA-II, the offspring population size is equal to the population size, but we can set its value from 1 (which leads to a steady-state selection scheme) to 400.

The initial population is typically filled with randomly created solutions, but we also allows to use a latin hypercube sampling scheme and a strategy similar to the one used in the scatter search algorithm.

The *autoNSGAII* has a *variation* component than can take a single value named *crossoverAndMutationVariation*. It is intended to represent the typical crossover and mutation operators of a genetic algorithm (additional values, e.g., *DifferentialiEvolutionVariation* are expected to be added in the future). The *crossover* operators included are *SBX* (simulated binary crossover) and *BLX_Alpha*, which are featured by a given probability and a *crossoverRepairStrategy*, which defines what to do when the crossover produces a variable value out of the allowed bounds (please, refer to Section 3.2 and Figure 3 in the paper). The *SBX* and *BLX_Alpha* require, if selected, a distribution index (a value in the range [5.0, 400]) and an alpha value (in the range [0.0, 1.0]), respectively. Similarly, there are two possible mutation operators to choose from, *polynomial* and *uniform*, requiring both a mutation probability and a repairing strategy; the polynomial mutation has, as the SBX crossover, a distribution index parameter (in the range [5.0, 400]) and the *uniform* mutation needs a perturbation value (in the range [0.0, 1.0]).

Finally, the *selection* operator be *random* or *tournament*; this last one can take a value between 2 (i.e., binary tournament) and 10.

As we intend to use irace as auto-tuning package, it requires a text file containing information about the parameters, the values they can take, an their relationships. We have created then a file called `parameters-NSGAII.txt` containing the required data:

```
algorithmResult                         "--algorithmResult "                    c  ␣
↪     (externalArchive,population)
populationSize                          "--populationSize "                     o  ␣
↪     (100)
populationSizeWithArchive               "--populationSizeWithArchive "          o  ␣
↪     (10,20,50,100,200)          | algorithmResult %in% c("externalArchive")
```

(continues on next page)

(continued from previous page)

```
#
maximumNumberOfEvaluations                "--maximumNumberOfEvaluations "          c  ␣
↪     (25000)
createInitialSolutions                    "--createInitialSolutions "              c  ␣
↪     (random,latinHypercubeSampling,scatterSearch)
#
variation                                 "--variation "                           c  ␣
↪     (crossoverAndMutationVariation)
offspringPopulationSize                   "--offspringPopulationSize "             o  ␣
↪     (1,10,50,100)
crossover                                 "--crossover "                           c  ␣
↪     (SBX,BLX_ALPHA)
crossoverProbability                      "--crossoverProbability "                r  ␣
↪     (0.0, 1.0)                       | crossover %in% c("SBX","BLX_ALPHA")
crossoverRepairStrategy                   "--crossoverRepairStrategy "             c  ␣
↪     (random, round, bounds)          | crossover %in% c("SBX","BLX_ALPHA")
sbxDistributionIndex                      "--sbxDistributionIndex "                r  ␣
↪     (5.0, 400.0)                     | crossover %in% c("SBX")
blxAlphaCrossoverAlphaValue               "--blxAlphaCrossoverAlphaValue "         r  ␣
↪     (0.0, 1.0)                       | crossover %in% c("BLX_ALPHA")
mutation                                  "--mutation "                            c  ␣
↪     (uniform, polynomial)
mutationProbability                       "--mutationProbability "                 r  ␣
↪     (0.0, 1.0)                       | mutation %in% c("uniform","polynomial")
mutationRepairStrategy                    "--mutationRepairStrategy "              c  ␣
↪     (random, round, bounds)          | mutation %in% c("uniform","polynomial")
polynomialMutationDistributionIndex       "--polynomialMutationDistributionIndex " r  ␣
↪     (5.0, 400.0)                     | mutation %in% c("polynomial")
uniformMutationPerturbation               "--uniformMutationPerturbation "         r  ␣
↪     (0.0, 1.0)                       | mutation %in% c("uniform")
#
selection                                 "--selection "                           c  ␣
↪     (tournament, random)
selectionTournamentSize                   "--selectionTournamentSize "             i  ␣
↪     (2, 10)                          | selection %in% c("tournament")
#
```

To know about the syntax of irace configuration files, please refer to the irace documentation.

2.3 The `EvolutionaryAlgorithm` class

Once we have defined the parameters of NSGA-II that can be tuned, the next issue to deal with is to have an implementation of the algorithm that can be configured with any valid combination of parameter values. The implementation of NSGA-II provided by jMetal is based on inheritance from the `AbstractEvolutionaryAlgorithm` class, so adapting it for auto-configuration is not a simple task, so our decision has been to create a new Maven subproject, called `jmetal-auto` from scratch and include in it all the classes related to the auto-configuration of metaheuristics. This way we do not interfere in the existing code, but with the disadvantage that we are going to have duplications of some functionalities. In particular,

The following code snippet include the most relevant parts of the `EvolutionaryAlgorithm` class, which is the algorithm template we have defined for developing autoconfigurable metaheuristics. It is not an abstract but a regular class containing the basic components of an evolutionary algorithm, including the selection, variation and replacement steps.

```
package org.uma.jmetal.auto.algorithm;
...
public class EvolutionaryAlgorithm<S extends Solution<?>>{
  ...
  public EvolutionaryAlgorithm(
    String name,
    Evaluation<S> evaluation,
    InitialSolutionsCreation<S> initialPopulationCreation,
    Termination termination,
    MatingPoolSelection<S> selection,
    Variation<S> variation,
    Replacement<S> replacement,
    Archive<S> externalArchive) {
    ...
  }

  public void run() {
    population = createInitialPopulation.create();
    population = evaluation.evaluate(population);
    initProgress();
    while (!termination.isMet(attributes)) {
      List<S> matingPopulation = selection.select(population);
      List<S> offspringPopulation = variation.variate(population, matingPopulation);
      offspringPopulation = evaluation.evaluate(offspringPopulation);
      updateArchive(offspringPopulation);

      population = replacement.replace(population, offspringPopulation);
      updateProgress();
    }
  }

  private void updateArchive(List<S> population) {
    if (externalArchive != null) {
      for (S solution : population) {
        externalArchive.add(solution);
      }
    }
  }

  ...

  @Override
  public List<S> getResult() {
    if (externalArchive != null) {
      return externalArchive.getSolutionList();
    } else {
      return population;
    }
  }
}
```

To configure NSGA-II, we have developed a package `org.uma.jmetal.auto.component` which provides components that can be used with the `EvolutionaryAlgorithm` class. Each component has an interface and a number of implementations. It is worth mentioning that two of the components, `evaluation` and `termination`, will not typically be used in the auto-configuration of the algorithm, but the `termination` is particularly interesting because it allows to define different stopping conditions: by number of evaluations, by computing time, and when the user presses a key.

2.4 The AutoNSGAII class

An example of configuring and running NSGA-II with these EvolutionaryAlgorithm class is provided in org.uma.jmetal.auto.algorithm.nsgaii.NSGAII, where that class is instantiated with the components leading to an standard NSGA-II. However, our purpose is to have the ability of automatically configure NSGA-II, so we need something more flexible.

The approach we have adopted is to get a sequence of pairs <parameter, value> as input, which is parsed to properly get a version of NSGA-II. This task is performed by class org.uma.jmetal.auto.algorithm.nsgaii.AutoNSGAII. This way, to get an NSGA-II algorithm with standard settings the following string must be passed to class AutoNSGAII from the command line:

```
--problemName org.uma.jmetal.problem.multiobjective.zdt.ZDT1 "
+ "--referenceFrontFileName ZDT1.pf "
+ "--maximumNumberOfEvaluations 25000 "
+ "--algorithmResult population "
+ "--populationSize 100 "
+ "--offspringPopulationSize 100 "
+ "--createInitialSolutions random "
+ "--variation crossoverAndMutationVariation "
+ "--selection tournament "
+ "--selectionTournamentSize 2 "
+ "--rankingForSelection dominanceRanking "
+ "--densityEstimatorForSelection crowdingDistance "
+ "--crossover SBX "
+ "--crossoverProbability 0.9 "
+ "--crossoverRepairStrategy bounds "
+ "--sbxDistributionIndex 20.0 "
+ "--mutation polynomial "
+ "--mutationProbability 0.01 "
+ "--mutationRepairStrategy bounds "
+ "--polynomialMutationDistributionIndex 20.0 "
```

We include a class named org.uma.jmetal.auto.algorithm.nsgaii.NSGAWithParameters" showing how to use this parameter string with ``AutoNSGAII.

2.5 Auto-configuring NSGA-II with irace

To replicate the results presented in https://doi.org/10.1145/3319619.3326832 all the needed resources are include in the folder jmetal-auto/src/main/resources/irace. Just copy the contents of that folder to the machine where you are going to run the experiments. Take into account that irace will generate thousands of configurations, so using a multi-core machine is advisable (we use a Linux virtual machine with 24 cores). We have tested the software in both Linux and macOS. A requirement of irace is to have R installed.

The contents of irace folder is the following:

1. irace.tar.gz: file containing irace
2. parameters-NSGAII.txt: file describing the parameters that can be tuned, including their allowed values and their dependences. You are free to modify some parameter values if you know their meaning.
3. instances-list.txt: the problems to be solved and their reference Pareto fronts are included here. It currently contains the following:

```
org.uma.jmetal.problem.multiobjective.zdt.ZDT1 --referenceFrontFileName ZDT1.pf
org.uma.jmetal.problem.multiobjective.zdt.ZDT2 --referenceFrontFileName ZDT2.pf
org.uma.jmetal.problem.multiobjective.zdt.ZDT3 --referenceFrontFileName ZDT3.pf
org.uma.jmetal.problem.multiobjective.zdt.ZDT4 --referenceFrontFileName ZDT4.pf
org.uma.jmetal.problem.multiobjective.zdt.ZDT6 --referenceFrontFileName ZDT6.pf
```

We must note that **currently we can only auto-configure NSGA-II with benchmark problems** included in jMetal.

4. `scenario-NSGAII.txt`: default irace parameters (we usually keep this file unchanged)
5. `target-runner`. Bash script which is executed in every run of irace. It contains as `FIXED_PARAMS` the path of the `jmetal-auto-6.0-SNAPSHOT-jar-with-dependencies.jar` which is included in the resources directory (this file can be generated with Maven just running `mvn package` from the project root directory).
6. `run.sh`. Bash script to run irace. VERY IMPORTANT: the number of cores to be used by irace are indicated in the `IRACE_PARAMS` variable (the default value is 24).

To run irace simply run the following command:

```
./run.sh NSGAII
```

Then irace will create a directory called `execdir` where it will write a number of output files. Two of those files are of particular interest: `irace.stderr.out`, which should be empty if everything is ok, and `irace.sdtout.err`, which contains the configurations being tested and, when irace stops, the best configurations founds. These configurations can be used with the `NSGAWithParameters` program.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search