Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

ARTICLE TYPE

An experimental comparison of metaheuristic frameworks for multi-objective optimization

Aurora Ramírez | Rafael Barbudo | José Raúl Romero*

Dpto. Informática y Análisis Numérico, Universidad de Córdoba, Córdoba 14071, Spain

Correspondence
*Corresponding author. Email: jrromero@uco.es

Summary

Multi-objective optimization problems frequently appear in many diverse research areas and application domains. Metaheuristics, as efficient techniques to solve them, need to be easily accessible to users with different expertise and programming skills. In this context, metaheuristic optimization frameworks are helpful, as they provide popular algorithms, customizable components and additional facilities to conduct experiments. Due to the broad range of available tools, this paper presents a systematic evaluation and experimental comparison of ten frameworks, covering from multi-purpose, consolidated tools to recent libraries specifically designed for multi-objective optimization. The evaluation is organized around seven characteristics: search components and techniques, configuration, execution, utilities, external support and community, software implementation and performance. An analysis of code metrics and a series of experiments serves to assess the last two features. Lesson learned and open issues are also discussed as part of the comparative study. The outcomes of the evaluation process reveal a contrasted support to recent advances in multi-objective optimization, with a lack of novel algorithms and variety of metaheuristics other than evolutionary algorithms. The experimental comparison also reports significant differences in terms of both execution time and memory usage under demanding configurations.

KEYWORDS:
metaheuristic optimization framework, multi-objective optimization, metaheuristics, evolutionary algorithms, swarm intelligence

1 | INTRODUCTION

A large number of real-world applications belonging to very different areas, e.g., engineering (Zavala, Nebro, Luna, & Coello Coello 2014) or healthcare (Tsai, Chiang, Ksentini, & Chen 2016), can take advantage of non-exact resolution methods like metaheuristics (Boussaïd, Lepagnot, & Siarry 2013) to efficiently solve their optimization problems. Most of these problems are multi-objective in nature (Stewart et al. 2008), meaning that two or more properties, often in conflict, need to be simultaneously optimized (Coello Coello, Lamont, & Van Veldhuizen 2007). Multi-objective optimization (MOO) can benefit from the general strengths of metaheuristics, such as their flexibility to cope with different types of decision variables or the ease to incorporate problem-specific knowledge (Jones, Mirrazavi, & Tamiz 2002). The presence of multiple objectives, which imposes a different point of view in solving the problem, has led to the appearance of multi-objective algorithms. Since 2013, the field is gaining strength again due to the interest in optimization problems with many objectives (B. Li, Li, Tang, & Yao 2015).

As a result of the growing complexity of both types of problems and algorithms, applying MOO techniques is a challenging task to domain experts, who are likely not to have the necessary programming skills to code such advanced techniques from scratch. Metaheuristic optimization frameworks (MOFs), which are a core element of the toolkit for researchers interested in metaheuristics, appear as a practical alternative. These frameworks provide widely-adopted techniques from which new algorithms are developed and experimentally validated. Conceived to alleviate coding efforts, the algorithms implemented by these MOFs are also highly customizable to allow domain experts to adapt them to address user-defined problems (Voß & Woodruff 2002). The need for generic components in the context of software tools for evolutionary computation was already discussed by Gagné and Parizeau (2006). Regarding the solution representation, MOFs for evolutionary computation should allow both the use of an existing encoding and the definition of new formulations. Similarly, fitness evaluation should support both minimization and maximization problems, as well as the definition of multiple objectives. Focusing on the search process, users should be allowed to choose among a set of available genetic operators, configure their parameters, and apply different evolutionary models. Regarding the configuration, MOFs should provide support to configure the algorithm components, as well as the output results and reports.

The wide range of options makes it difficult to choose a particular MOF better than other, since it often depends on multiple factors. On the one hand, the ease of use and availability of a graphical user interface (GUI) might be essential to those users with little experience in the field of metaheuristics. On the other hand, domain experts and researchers might be more conditioned by aspects related to the variability of techniques, having a development environment that makes integration easier, or performance issues if extensive experiments are conducted. Since most of the existing MOFs offer a common set of basic utilities, the availability of more advanced techniques or novel algorithms could be additional discriminant factors.

When many alternatives are available, comparative studies (Parejo, Ruiz-Cortés, Lozano, & Fernández 2012; Silva, de Souza, Souza, & de França Filho 2018) naturally emerge to provide a source of information that makes it easier to discriminate between different options. These studies do not only compile the list of available tools, but also analyze their differences objectively. Parejo et al. (2012) presented a comprehensive survey that considered a great number of relevant, multi-level characteristics (with their corresponding weights) for ten general-purpose MOFs, mostly addressing single-objective optimization problems. Their analysis of ten MOFs, some of which are still maintained and updated, was focused on six characteristics including design issues, documentation and advanced computed capabilities. Multi-objective metaheuristics was included in the analysis, but only referred to a list of algorithms available. Their study showed that not all MOFs implement MOO while those with some support included only a few multi-objective evolutionary algorithms (MOEAs) proposed in the early 2000s. However, it should be noted that frameworks specialized in MOO were excluded from the selection. More recently, Silva et al. (2018) analyzed a greater number of MOFs, with emphasis on their support to hybrid metaheuristics and multi-agent systems. Regarding MOO, the review indicated whether these frameworks are able to face multi-objective problems, as well as the list of available algorithms. Due to their general scope and timing, these studies did not cover the wide spectrum of MOFs for MOO available nowadays. Furthermore, some have recently appeared in the last years, or have experienced major upgrades. Besides, none of these studies present experiments to analyze their differences from a more empirical perspective.

This paper aims at answering the following research question: To what extent is MOO currently being supported by MOFs? To this end, we present a comparative study of ten MOFs that provide some kind of support for the resolution of multi-objective problems. Then, seven characteristics are evaluated on the basis of a set of features of interest, such as availability of algorithms or configuration facilities, thus providing useful guidelines to different practitioners. The proposed comparison goes beyond the collection and analysis of functionalities, and quality aspects like execution time or memory consumption are experimentally validated. The performance of different implementations of MOEAs and particle swarm optimization (PSO) methods is evaluated in order to force their scalability with respect to the population size, the number of objectives and the number of generations. The results reveal that specialized tools better capture new trends in the MOO field, despite the fact that mature MOFs provide more general utilities and external support. The experimental outcomes show that, under standard experimental settings, MOFs provide similar performance in terms of execution time and memory consumption, and only some differences are observed for certain algorithms and benchmarks. However, these differences become more evident and pronounced as increasingly demanding configurations are considered.

The rest of the paper is structured as follows. Section 2 presents the comparison methodology, including the definition of the characteristics under evaluation. Then, we present an extensive comparison organized in the next four sections: availability of search components and techniques (Section 3), configuration and execution capabilities (Section 4), general utilities and external support (Section 5) and software analysis (Section 6). Lessons learned and open issues are discussed in Section 7. Finally, conclusions are drawn in Section 8.

2 | COMPARISON METHODOLOGY

This section presents a brief introduction to the characteristics under analysis, and explains the methodology followed to perform the analysis and evaluate the selected frameworks.

TABLE 1 The set of characteristics and their features.

Characteristic	Feature	Outcomes
C1: search components and techniques	C1F1: types of metaheuristics	List of available search techniques
	C1F2: families of algorithms	List of algorithms per family
	C1F3: encodings and operators	List of operators per encoding
C2: configuration	C2F1: inputs	List of input types and data formats
	C2F2: batch processing	List of possible ways to run experiments
	C2F3: outputs	List of output types and data formats
C3: execution	C3F1: multi-thread execution	List of possible ways to apply parallelism
	C3F2: distributed execution	List of distributed computing models
	C3F3: stop and restart mode	Support to serialization and checkpointing
	C3F4: fault recovery	Support to parameter tuning and exception handling
	C3F5: execution and control logs	Support to show intermediate results and logs
C4: utilities	C4F1: graphical user interface	List of functionalities associated to the GUI
	C4F2: benchmarks	List of available test problems
	C4F3: quality indicators	List of available quality indicators
C5: documentation and community support	C5F1: software license	Type of license
	C5F2: available documentation	Types of external documentation
	C5F3: software update	Number of releases since January 2015
	C5F4: development facilities	List of auxiliary tools
	C5F5: community	List of communication channels
C6: software implementation	C6F1: implementation and execution	Programming language and execution platform
	C6F2: external libraries	Types of third-party libraries used
	C6F3: software metrics	List of metrics associated to code quality
C7: performance at runtime	C7F1: execution time	Measurement of execution time
	C7F2: memory consumption	Measurement of memory usage

2.1 | Overview of characteristics

The evaluation model consists of a hierarchical categorization of characteristics and features. This kind of model is a common practice when evaluating software tools (Parejo et al. 2012). The proposed characteristics capture the evaluation goals from different complementary views. They cover not only static properties, but also dynamic properties, which are essential to assess the performance in real contexts. Furthermore, the scope of these characteristics varies from general requirements, such as configuration and execution capabilities, to more specific functionalities and utilities that usually make a difference and offer added value with respect to the other proposals (Meng, Liu, Yang, Cai, & Liu 2010).

Table 1 lists the characteristics and breaks them down into their respective features, including their outcomes. Characteristics are defined as follows:

- *C1: search components and techniques*. It refers to the collection of building blocks that can be combined to solve multi-objective problems.
- *C2: configuration*. It evaluates the possibility to create experiments, their parametrization and reporting capabilities.
- *C3: execution*. It covers aspects related to how experiments are run and controlled.
- *C4: utilities*. It encompasses available utilities, divided into GUI, benchmarks and quality indicators.
- *C5: documentation and community support*. It is focused on the available documentation and the technologies for software distribution and interaction with the development team and other users.
- *C6: software implementation*. It analyzes development decisions like the programming language or dependencies to external libraries, as well as source code metrics.

- *C7: performance at runtime*. It evaluates execution time and memory consumption to provide information regarding performance and scalability.

2.2 | Evaluation process and supporting tools

The evaluation process started with the definition of a list of features of interest (see Section 2.1), which was iteratively refined. Data were collected from both documentation and source code, using different strategies for their evaluation.[1] For C1 to C6, the process is described below:

1. Determine a preliminary set of options—e.g. algorithms, benchmarks or indicators—according to the MOO literature (Coello Coello et al. 2007; Zhou et al. 2011).
2. Create a checklist for these options by agreeing a common nomenclature, and define the conditions that any MOF should meet for its positive evaluation.
3. Collect evidences of the level of compliance of the available documentation. If the information is not clear or missing, then the source code should be inspected and executed.
4. Refine the list of options to add any recent development. For inclusion, the algorithm, operator, benchmark or indicator should be accompanied by a reference. Otherwise, it should appear in at least two MOFs in order to be considered of general interest and not an in-house development.

Note that two particular features require special treatment: C3F4 (fault recovery) and C6F3 (software metrics). As part of the fault recovery evaluation, we prepared short experiments with incomplete or erroneous configurations in order to observe how the MOF responds to missing or wrong parameter values, respectively. Such experiments also served to confirm the use of default values to fix an user's mistake. The following situations were considered:

1. Missing parameters. The framework is expected to alert the user to the lack of a mandatory search component or parameter. The missing elements under evaluation are: a) population size; b) stopping criterion; c) optimization problem; d) algorithm; and e) crossover operator (extensible to mutation operator).
2. Invalid values. The framework should report that the specific value of a parameter is not valid. The following situations are considered: a) the population size is a negative number; b) the maximum number of evaluations is a negative number; c) the optimization problem does not exist (a wrong name is used); the algorithm does not exist (a wrong name is used); and d) the crossover probability is greater than 1 (extensible to the mutation operator).
3. Default values. All previous scenarios are evaluated. It is observed whether the MOF gives feedback about the assigned value.

As for C6F3, software metrics related to the maintainability and usability (Maxim & Kessentini 2016) are the most relevant. Notice that reliability, portability and efficiency are already covered by other features, whereas other dimensions of the software quality model considered by the ISO Std. 25010, such as security, are less applicable to a MOF, since they are not critical systems. Maintainability involves aspects of modularity, which can be mapped to code size or its organization (number and type of artifacts like classes, functions, etc.), and testability, for which coverage is a well-established indicator. Usability is linked to understandability and learnability, for which complexity and documentation metrics are appropriate. Therefore, we create a list of 14 metrics divided into four groups: code, complexity, testing and documentation. In terms of support tools, we found that two suites, namely SonarQube[2] and the Eclipse plugin *Metrics*,[3] served our purposes. Notice that depending on the programming language of the MOF under evaluation, some metrics were not available (see the technical report for all the details about the experimental environment).

Two experiments were planned to evaluate characteristic C7 in order to gather several execution time and memory measurements under different conditions. A first experiment aims to assess the performance of each framework for a variety of algorithms and benchmarks. The second experiment seeks to analyze how well a MOF scales with respect to the population size, the maximum number of generations and the number of objectives. Notice that the choice of algorithms, operators and benchmarks should be founded on the outcomes of C1F2, C1F3 and C4F2, respectively. No algorithm has been implemented in order not to introduce bias. Only some benchmarks and genetic operators were carefully

[1] In case of a partial fulfillment of a feature, it is not reported here. Instead, a full analysis of such cases are provided in the technical report (see Additional material in page 26).

[2] `http://www.sonarqube.org/` (Accessed May 22th, 2020)

[3] `http://metrics.sourceforge.net/` (Accessed May 22th, 2020)

TABLE 2 Basic information of the ten frameworks under analysis.

Name	Reference	Version	Year	Website
DEAP	(Fortin, De Rainville, Gardner, Parizeau, & Gagné 2012)	1.3.0	2019	`https://github.com/DEAP/deap`
ECJ	(Scott & Luke 2019)	27	2019	`https://cs.gmu.edu/~eclab/projects/ecj/`
EvA	(Kronfeld, Planatscher, & Zell 2010)	2.2.0	2015	`http://www.ra.cs.uni-tuebingen.de/software/eva2/`
HeuristicLab	(Wagner et al. 2014)	3.3.16	2019	`http://dev.heuristiclab.com/`
JCLEC-base	(Ventura, Romero, Zafra, Delgado, & Hervás 2008)	4.0	2014	`http://jclec.sourceforge.net/`
+ JCLEC-MO	(Ramírez, Romero, García-Martínez, & Ventura 2019)	1.0	2018	`http://www.uco.es/kdis/jclec-mo`
jMetal	(Nebro, Durillo, & Vergne 2015)	5.9	2019	`http://jmetal.github.io/jMetal/`
MOEA Framework	(Hadka 2019)	2.13	2019	`http://moeaframework.org/`
Opt4J	(Lukasiewycz, Glaß, Reimann, & Teich 2011)	3.1.4	2015	`http://opt4j.sourceforge.net/`
ParadisEO-MOEO	(Liefooghe, Jourdan, & Talbi 2011)	2.0.1	2012	`http://paradiseo.gforge.inria.fr/`
Platypus	(Hadka 2020)	1.0.4	2020	`https://github.com/Project-Platypus/Platypus`

implemented when no common options were available. Since benchmarks and operators are small pieces of software, they could be implemented following the programming style of the corresponding MOF. The list of algorithms, operators and benchmarks is presented in Section 6.2.

Experiments were run on a Debian 8 computer with 8 cores Intel Core i7-2600 CPU at 3.40 GHz and 16 GB RAM. Syrupy[4] was used to log CPU time and RAM memory consumption at runtime. We experimentally adjusted log intervals depending on the framework and configuration in order to always obtain 30 measurements. Runs were executed several times so as not to introduce bias into the measurement. For the first experiment, we repeated the execution five times, and the process was done using one core only. For the second experiment, six executions were run in parallel (two cores) per each pair MOF/configuration.

2.3 | Selected frameworks

In order to obtain a representative sample of software tools, we did not impose any restriction regarding the way they provide support to MOO. Therefore, any tool or library including the implementation of at least two multi-objective metaheuristic algorithms was initially selectable. Although we considered diverse programming languages, we focus on MOFs designed under the precepts of the object-oriented paradigm, as suggested by Parejo et al. (2012). It has shown to be a successful paradigm frequently adopted in the area, since it promotes the definition of independent and reusable components, e.g. algorithms and operators, that allows satisfying the genericity criteria expected for these kinds of developments (Gagné & Parizeau 2006). Finally, notice that source code and documentation must be publicly available in order to carry out measurement and experimentation on the frameworks.

Table 2 shows the list of the ten selected frameworks, as well as the analyzed version – the latest stable version available at the time of writing – and website. Two groups can be distinguished with respect to its degree of specialization. DEAP, ECJ, EvA, HeuristicLab and Opt4J are multi-purpose frameworks, whereas jMetal, MOEA Framework and Platypus are mostly focused on MOO. Additionally, ParadisEO-MOEO and JCLEC-MO represent intermediate solutions, since they both present an independent core package and complementary modules or wrappers for specialized techniques.

3 | SEARCH COMPONENTS AND TECHNIQUES (C1)

This section analyzes three features defined to evaluate the elements taking part in multi-objective search techniques: *types of metaheuristics* (C1F1), *families of algorithms* (C1F2) and *encodings and operators* (C1F3). Apart from consulting the available documentation, source code was inspected

[4]`https://github.com/jeetsukumaran/Syrupy` (Accessed May 22th, 2020)

TABLE 3 Coverage of C1F1: types of metaheuristics.

	DEAP	ECJ	EvA	HeuristicLab	JCLEC-MO	jMetal	MOEA Framework	Opt4J	ParadisEO-MOEO	Platypus
C1F1a: single-solution-based metaheuristics										
Hill climbing			✓							
Simulated annealing			✓							
Pareto-based local search									✓	
C1F1b: population-based metaheuristics										
Cellular algorithm						✓				
Differential evolution			✓			✓	✓	✓		
Evolution strategy			✓		✓	✓	✓			
Evolutionary programming			✓		✓					
Genetic algorithm	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Genetic programming	✓	✓	✓	✓	✓		✓			
Grammatical evolution		✓					✓			
Particle swarm optimization					✓	✓	✓	✓		✓
Scatter search						✓				

and run for some simple examples in order to verify any possible combination of metaheuristic paradigm and MOO algorithms that could not be clearly stated in the documentation.

3.1 | Types of metaheuristics (C1F1)

Metaheuristic paradigms have been classified into two groups according to the literature (Boussaïd et al. 2013). Consequently, C1F1a covers single-solution-based metaheuristics, and C1F2b refers to population-based techniques. Table 3 details those paradigms that are currently supported by each MOF. Two possibilities were considered to decide whether a multi-objective metaheuristic is really available or not. Firstly, a MOF defines an abstract implementation of the metaheuristic paradigm, i.e. the search iterative procedure, that is instantiated with specific elements of multi-objective problem solvers, e.g. Pareto evaluation. Secondly, the MOF includes an algorithm that is specific to such metaheuristic paradigm, e.g. the Pareto Archive Evolution Strategy confirms the availability of evolution strategies. Other metaheuristics included in the MOF are discarded because their implementation is coupled to single-objective problems.

As shown in Table 3, single-solution-based metaheuristics rarely appear in the context of MOO, EvA and ParadisEO-MOEO being the unique frameworks that implement some of them. Focusing on population-based techniques, evolutionary methods—particularly genetic algorithms— are the most frequently available. On the contrary, the application of swarm intelligence is restricted to PSO and it is only supported by five frameworks.

Focusing on whether the metaheuristic paradigm is instantiable or not, each approach is followed by half of the selected MOFs, and two MOFs—ParadisEO-MOEO and Opt4j—have examples of both. DEAP, ECJ, EvA and JCLEC-MO opt for a concrete class implementing the metaheuristic paradigm, which is then combined with other classes that include the selection and replacement methods of a particular multi-objective algorithm. This 'plug-and-play' approach provides flexibility to combine MOO ideas, e.g. non-dominated sorting, with any kind of evolutionary algorithm. However, it requires more configuration to be set by the user. In contrast, HeuristicLab, jMetal, MOEA Framework and Platypus build the algorithm within a unique, single class. In HeuristicLab, jMetal and MOEA Framework, this class mostly configures the default search components involved in the iterative process, thus customized variants could be created too. For all these four MOFs, there is also an abstract class that defines either the common behavior of the corresponding metaheuristic or the steps of a general heuristic algorithm.

3.2 | Families of algorithms (C1F2)

This section analyzes the catalog of multi-objective algorithms available in each MOF. Since the publication of the first MOEA, several *families* of algorithms have appeared. After two generations strongly focused on the Pareto dominance (Coello Coello 2003), novel techniques have emerged as multi-objective problems turn into many-objective ones (B. Li et al. 2015; Zhou et al. 2011). Based on the literature, the following features are established:

- *C1F2a: first generation*. Algorithms founded on the Pareto optimality that also include some niching or fitness sharing techniques.
- *C1F2b: second generation*. Algorithms that introduce an elitism mechanism, e.g. an external population or specific selection procedures.
- *C1F2c: relaxed dominance*. Algorithms that modify the principle of Pareto dominance.
- *C1F2d: indicator-based*. Algorithms that use a performance metric to guide the search towards the front.
- *C1F2e: decomposition*. Algorithms that define weight vectors to explore several search directions at the same time.
- *C1F2f: reference set*. Algorithms that consider a set of reference points to focus on specific regions of the search space.
- *C1F2g: preference-based*. Algorithms that incorporate information given by decision makers.

Table 4 shows the relation between algorithms—classified by their respective family—and frameworks, also including the median year of publication of the listed algorithms. This information reveals that general purpose MOFs provide few algorithms, mostly proposed in the early 2000s. In contrast, more specialized frameworks like JCLEC-MO, jMetal, MOEA framework and Platypus implement recent algorithms, and cover more families. NSGA-II (Deb, Pratap, Agarwal, & Meyarivan 2002), a second-generation method, is the unique algorithm available in all frameworks. Other frequent algorithms are SPEA2 (Zitzler, Laumanns, & Thiele 2001), from the same family as NSGA-II, OMOPSO (Sierra & Coello Coello 2005) (relaxed dominance), IBEA (Zitzler & Künzli 2004) (indicator-based), MOEA/D (Zhang & Li 2007) (decomposition) and NSGA-III (Deb & Jain 2014) (reference points), which are implemented in at least four frameworks. Many-objective algorithms—marked with an asterisk in Table 4—span across multiple families, but are barely offered. JCLEC-MO and jMetal provide five implementations each, followed by MOEA Framework that includes three algorithms. NSGA-III (Deb & Jain 2014) is the most commonly found algorithm, probably due to the popularity reached by its predecessor, and available in the three aforementioned frameworks, plus DEAP, ECJ and Platypus.

3.3 | Encodings and operators (C1F3)

Given that evolutionary computation predominates in the analysis of metaheuristics, this section analyzes the set of the available genetic operators, grouped by the required encoding.[5] The proposed classification considers binary, integer, permutation, double and tree encoding, as they are the most frequently used (Parejo et al. 2012) and have been found in at least two of the selected MOFs. After reviewing different sources, a common nomenclature was derived and check lists were created. Table 5 summarizes the findings by counting the number of operators provided by each MOF.

Focusing on crossover operators (feature C1F3a), the number of operators for binary, integer and permutation encodings looks similar for all frameworks. HeuristicLab seems to be an exception to this pattern, which indicates that this framework gives special importance to the variety of optimization problems to be solved. Those MOFs supporting GP also include operators to manage tree structures, but in an inferior number. We hypothesize that GP is not so extended and its target community has their own frameworks.[6]

Slightly different conclusions can be drawn with respect to the mutation operators (feature C1F3b). JCLEC-MO and HeuristicLab offer an extensive catalog of mutation operators, with special interest on tree and permutation encodings, respectively. Mutation operators for GP are also abundant in ECJ and, to a lesser extent, ParadisEO-MOEO. Overall, the fact that multi-objective algorithms are frequently tested with continuous optimization functions seems to be the reason behind the predominance of operators for double encoding. Binary encoding is also present in all MOFs, as it is often used to formulate popular benchmarks like the knapsack problem (KP).

[5] Extended information about other specialized operators and references are available in the technical report (see additional material in page 26).
[6] A list of GP-oriented frameworks is available at `http://geneticprogramming.com/software/` (Accessed May 22th, 2020).

TABLE 4 Coverage of C1F2: families of algorithms.

Family	DEAP	ECJ	EvA	HeuristicLab	JCLEC-MO	jMetal	MOEA Framework	Opt4J	ParadisEO-MOEO	Platypus
C1F2a: first generation	0	0	2	0	0	0	1	0	2	0
VEGA (1985)							✓			
MOGA (1995)			✓						✓	
NSGA (1995)			✓						✓	
C1F2b: second generation	2	2	6	2	4	10	6	2	2	3
SPEA (1999)			✓							
PAES (2000)					✓	✓	✓			
PESA (2000)			✓							
PESA2 (2001)			✓			✓	✓			
SPEA2 (2001)	✓	✓	✓		✓	✓	✓	✓	✓	✓
NSGA-II (2002)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
GDE3 (2005)						✓	✓			✓
CMAES-MO (2007)			✓	✓			✓			
MOCHC (2007)					✓	✓				
AbYSS (2008)						✓				
CellDE (2008)						✓				
MOCell (2009)						✓				
FAME (2019)						✓				
C1F2c: relaxed dominance	0	0	0	0	4	2	3	1	0	3
ϵ-MOEA (2002)					✓		✓			✓
OMOPSO (2005)					✓	✓	✓	✓		✓
SMPSO (2009)					✓	✓	✓			✓
GrEA (2013) *					✓					
C1F2d: indicator based	0	0	0	0	3	5	2	1	1	1
IBEA (2004)					✓	✓	✓		✓	✓
SMS-EMOA (2007)					✓	✓	✓	✓		
HypE (2011) *					✓					
MOMBI (2013) *						✓				
MOMBI-II (2015) *						✓				
D-NSGA-II (2018) *						✓				
C1F2e: decomposition	0	0	0	0	1	4	3	0	0	1
MSOPS (2003)							✓			
MOEA/D (2007)					✓	✓	✓			✓
dMOPSO (2011)						✓				
DBEA (2015) *							✓			
MOEADD (2015) *						✓				
CDG (2018)						✓				
C1F2f: reference set	1	1	0	0	2	2	2	0	0	1
R-NSGA-II (2006)						✓				
NSGA-III (2014) *	✓	✓			✓	✓	✓			✓
RVEA (2016) *					✓		✓			
C1F2g: preference based	0	0	0	0	1	3	0	0	0	0
WASF-GA (2014)						✓				
ESPEA (2015)						✓				
GWASF-GA (2015)						✓				
PAR (2016) *					✓					
Number of algorithms	**3**	**3**	**8**	**2**	**15**	**26**	**17**	**4**	**5**	**9**
Median publication year	**2002**	**2002**	**2001**	**2005**	**2007**	**2009**	**2005**	**2004**	**2002**	**2005**

* Originally proposed as a many-objective algorithm.

4 | CONFIGURATION AND EXECUTION

MOFs usually differ in how experiments should be created and executed. Both aspects are studied in this section, including details on the use of data formats in both inputs and outputs, the parametrization capability and the support to parallel and distributed computing.

TABLE 5 Coverage of C1F3: encodings and operators.

	DEAP	ECJ	EvA	HeuristicLab	JCLEC-MO	jMetal	MOEA Framework	Opt4J	ParadisEO-MOEO	Platypus
C1F3a: crossover operators										
Binary	3	3	3	3	4	3	4	3	3	1
Integer	3	5	3	8	3	2	1	3	2	0
Permutation	2	0	1	10	2	1	1	2	1	1
Double	5	6	10	10	6	3	8	6	6	4
Tree	2	2	1	1	2	0	1	0	1	0
Total	**15**	**16**	**18**	**32**	**17**	**9**	**15**	**14**	**13**	**6**
C1F3b: mutation operators										
Binary	1	2	3	2	3	1	1	1	3	1
Integer	1	1	1	2	3	1	1	1	0	0
Permutation	0	0	2	7	2	1	2	3	2	2
Double	2	3	1	5	5	4	2	2	2	4
Tree	2	7	2	1	7	0	1	0	5	0
Total	**6**	**13**	**9**	**17**	**20**	**7**	**7**	**7**	**12**	**7**

4.1 | Configuration features (C2)

MOFs have to provide mechanisms to easily configure algorithm parameters and prepare a batch of experiments. In this process, the use of standard data formats for both inputs and outputs could make their processing easier. The following features are defined in order to show the level of compliance of all these aspects:

- *C2F1: inputs*. This feature evaluates the type of input (C2F1a) and the data format (C2F2b) used in configuration files, if any.
- *C2F2: batch processing*. This feature considers whether a single task can be repeated (C2F2a), the possibility of executing a sequence of tasks (C2F2b) and running multiple tasks in parallel (C2F2c). A single and independent algorithm execution is referred as a task.
- *C2F3: outputs*. This feature is used to determine the type of output (C2F3a), the format of the generated data (C2F3b), the parametrization capability (C2F3c) and the possibility of setting diverse output devices (C2F3d).

Table 6 shows the outcomes for the three features. Focusing on C2F1, half of the MOFs provide at least two different input mechanisms. Configuration files are a common alternative in general-purpose MOFs, for which three different data formats have been found: KPV, XML or YAML.

With the exception of Platypus, all the MOFs allow the user to set the random seed before running the algorithm, looking for replicable executions (C2F2a). A series of independent executions can be concatenated in seven MOFs (C2F2b), but only four of them allow for parallelization (C2F2c). The observed lack of support to both options might limit somehow the usability of these frameworks to carry out extensive experiments without programming skills.

On the contrary, there is a great flexibility regarding the generation of outcomes (C2F3). Firstly, eight out of the ten MOFs are able to save results in files, and most of them also provide additional mechanisms to visualize them (C2F3a). For instance, jMetal incorporates a mechanism to report intermediate results in a graphical way, though it cannot be considered as a fully functional GUI. Similarly, DEAP and Platypus allow including some code to visualize a Pareto front via matplotlib, an external Python library.

TABLE 6 Coverage of C2: configuration.

	DEAP	ECJ	EvA	HeuristicLab	JCLEC-MO	jMetal	MOEA Framework	Opt4J	ParadisEO-MOEO	Platypus
C2F1: inputs										
C2F1a: type of input	C	F, CLI	C, F, GUI	C, GUI	F	C	C	F, GUI	C, F	C
C2F1b: input data format		KPV	YAML		XML			XML	KPV	
C2F2: batch processing										
C2F2a: task replication	✓	✓	✓	✓	✓	✓	✓	✓	✓	
C2F2b: sequential tasks	✓		✓	✓	✓	✓	✓			✓
C2F2c: parallel tasks			✓	✓		✓				✓
C2F3: outputs										
C2F3a: type of output	CLI	CLI, F, GUI	CLI, F,GUI	CLI, F, GUI	C, F	F	CLI, F, GUI	F, GUI	C, F	CLI
C2F3b: output data format		TSV	TSV	XLSX	CSV	CSV,TSV	TSV	TSV	TSV	JSON
C2F3c: parametrization										
Report frequency			✓		✓	✓	✓	✓	✓	✓
Report path		✓			✓	✓		✓	✓	
C2F3d: degree of flexibility										
Customizable		✓	✓	✓	✓		✓	✓		
Programmable						✓	✓		✓	✓

C: code, CLI: command line, F: file

As for C2F3b, TSV appears as the most frequently used data format. Markup languages like XML and XLSX are considered too. An important aspect concerning C2F3c is that the user can select, at least for one output mode, how often the results should be reported and set the path where they will be saved. JCLEC-MO, jMetal, Opt4J and ParadisEO-MOEO are the most flexible MOFs in this regard. Finally, most frameworks allow enabling and disabling the generation of results at the user's discretion (C2F3d), while four MOFs would still require some development in order to change the default behavior.

4.2 | Execution capabilities (C3)

In this section, MOFs are analyzed with respect to their availability of computational models exploiting parallelism and distribution mechanisms, as well as their mechanisms to monitor the execution and recover from errors. Since this type of information was not always available or clearly documented, we performed and executed some running examples (see Section 2). Next, the specific features under evaluation are described:

- *C3F1: multi-thread execution*. This feature indicates whether multithreading is supported. Notice that we differentiate between parallel evaluation of solutions (C3F1a) and the use of threads in other phases (C3F1b).
- *C3F2: distributed execution*. This feature focuses on the availability of the master-slave (C3F2a) and the island (C3F2b) models, which are the distributed models currently implemented by the selected MOFs.
- *C3F3: stop and restart mode*. This feature indicates whether an object state serialization mechanism (C3F3a) is implemented by the MOF. The second feature (C3F3b) looks for the existence of a working procedure to save and load checkpoints, meaning that serialized objects are effectively used.
- *C3F4: fault recovery*. This feature refers to the capability of a given MOF to control errors and recover from them. Related to parameter checking (C3F4a), several wrong configurations were tested to find out whether the MOF is able to identify missing values, prevent from

setting wrong values, and apply default values (see Section 2). A test case succeeds if the configuration error is identified and some feedback is reported by the framework. The percentage of successful cases is reported. In addition, a second feature refers to exception handling (C3F4b).

- *C3F5: execution and control logs*. This feature refers to the existence of any kind of log system, which should enable the visualization of either control messages or intermediate results.

Table 7 compiles the outcomes for all the aforementioned features. As expected, multithreading is mainly applied to accelerate the evaluation of solutions (C3F1a), since this phase usually demands the highest computational effort. Three MOFs use parallelism (C3F1b) to initialize the population (ECJ), decode solutions (Opt4J) or transform them (ParadisEO-MOEO). As for distributed models, they are not so commonly supported (C3F2), ECJ and MOEA Framework being the unique MOFs that implement both models. Focusing on C3F3, the implementation of a serialization mechanism (C3F3a) is a common practice in all but two MOFs, namely ParadisEO-MOEO and Platypus. However, just 60% of all frameworks are able to restore the checkpoints later (C3F3).

The executions performed to reveal how each framework controls the configuration of parameters (C3F4a) provide interesting insights. Firstly, the ability of ECJ to warn about both missing and wrong values is noteworthy. DEAP and JCLEC-MO frequently prevent from wrong values, whereas jMetal put more emphasis on detecting missing parameters. For the rest of frameworks, feedback about parameter configuration is reported in approximately half of the tested situations. This fact suggests that not all types of parameters are equally checked. In particular, we found that all MOFs impede the missing or wrong specification of the problem and algorithm. When the population size, the stopping criterion or the genetic operator are omitted or wrongly configured, DEAP, EvA, JCLEC-MO, Opt4J and ParadisEO-MOEO stop their execution due to raised exceptions. In contrast, HeuristicLab and Platypus continue to run despite these anomalies. Our analysis shows that the definition of default values is not a common practice among MOFs. With the exception of EvA, if needed, MOFs change parameter values to default values in less than 50% of the cases. When such an decision is made, only ECJ and MOEA Framework let the user know the value that has been set. As for C3F4b, all MOFs catch native exceptions in different moments of the execution, and most of them define their own exceptions too.

Finally, the coverage of C3F5 reveals that the majority of MOFs (60%) provide information about the progress of the launched experiment. ECJ, EvA and jMetal generate log messages with information about the experiment set-up process. Among them, EvA provides the more extensive information by returning the complete parameter configuration in a separate window. Messages informing that an algorithm has started and finished are generated by all the aforementioned frameworks, plus HeuristicLab, JCLEC-MO and Platypus. The total execution time is returned by EvA, HeuristicLab, and JCLEC-MO. MOEA Framework and Opt4J can also report the elapsed execution time by generation, but it should be configured as part of the class responsible for generating the search statistics. Although DEAP defines a "logbook", its current configuration does not provide any information about aspects related to the execution flow. Nonetheless, the user could define, implement and include his/her own measures. Similarly, any execution control in ParadisEO-MOEO has to be developed.

5 | UTILITIES AND EXTERNAL SUPPORT

This section discusses the availability of additional utilities like benchmarks and quality indicators. Other aspects related to the documentation and community support are also discussed.

5.1 | Additional utilities (C4)

The following features serve to evaluate those aspects related to the support utilities provided by the frameworks under analysis:

- *C4F1: graphical user interface*. GUI-based environments are usually found more practicable, especially by those users who are inexperienced in the application of metaheuristics. This feature analyzes some characteristics related to the use of a GUI, such as the possibility of designing experiments (C4F1a); the flexibility regarding parametrization (C4F1b); execution control options (C4F1c); and the possibility of generating charts to visualize results (C4F1d).

- *C4F2: benchmarks*. This feature provides the number of test problems currently implemented by each MOF, divided into continuous (C4F2a) and combinatorial (C4F2b) problems. A test suite containing multiple functions, such as DTLZ, is considered as one single benchmark.

TABLE 7 Coverage of C3: execution.

	DEAP	ECJ	EvA	HeuristicLab	JCLEC-MO	jMetal	MOEA Framework	Opt4J	ParadisEO-MOEO	Platypus
C3F1: multi-thread execution										
C3F1a: parallel evaluation	✓	✓		✓	✓	✓	✓	✓	✓	
C3F1b: parallelism in other phases		✓						✓	✓	
C3F2: distributed execution										
C3F2a: master-slave model		✓		✓			✓			
C3F2b: island model	✓	✓	✓				✓			
C3F3: stop and restart mode										
C3F3a: object state serialization	✓	✓	✓	✓	✓	✓	✓	✓		
C3F3b: save and load checkpoints	✓	✓		✓			✓	✓	✓	
C3F4: fault recovery										
C3F4a: control of parameter values										
Missing parameters	80%	100%	40%	40%	80%	40%	60%	40%	60%	60%
Wrong numerical values	40%	100%	60%	40%	40%	100%	60%	60%	60%	40%
Default values	0%	10%	50%	40%	10%	30%	40%	30%	10%	20%
C3F4b: exception handling										
Native exceptions	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Specific exceptions		✓		✓		✓	✓		✓	✓
C3F5: execution and control logs		✓	✓	✓	✓	✓				✓

- *C4F3: quality indicators*. The availability of evaluation metrics to compare the returned set of solutions in the objective space—a.k.a the Pareto front—offers the user the possibility of comparing algorithms. This feature reports the number of quality indicators,[7] classified according to the number of fronts required to compute them: one (C4F3a), two (C4F3b) or three (C4F3c).

The resulting evaluation of these features is compiled in Table 8. Note that the existence of a GUI (C4F1) does not always imply that the corresponding MOF provides full support to create, configure and execute experiments. Five MOFs have a GUI, but only EvA and HeuristicLab allow the user to define experiments, modify their parameters and visualize outcomes. HeuristicLab provides support to control the execution as well, being the most complete GUI among the MOFs under analysis. The rest of frameworks present some limitations. For instance, EvA and MOEA Framework allow stopping the execution of an algorithm at one point but without any possibility to resume it later. MOEA framework also imposes some restrictions with respect to the parameters that can be configured. ECJ allows using its GUI to run executions, but some results like the Pareto front cannot be visualized.

The analysis of benchmarks (C4F2) shows some differences between general-purpose and MOO-specific frameworks (see Table 8). MOEA Framework and jMetal implement the largest collection of test functions. With respect to ParadisEO-MOEO, it should be noted that it provides an implementation of the DTLZ, ZDT and WFG test suites, but they are external contributions to the project. As expected, the number of continuous problems is clearly higher than the number of combinatorial problems. The KP, LOTZ and the traveling salesman problem (TSP) are common examples of the latter. Curiously, both KP and TSP can be found as single-objective optimization problems in some frameworks, e.g., EvA, MOEA Framework and Opt4J, but their multi-objective variants are not provided yet.

[7] The full list of benchmarks and indicators is provided as part of the technical report (see additional material in page 26).

TABLE 8 Coverage of C4: utilities.

	DEAP	ECJ	EvA	HeuristicLab	JCLEC-MO	jMetal	MOEA Framework	Opt4J	ParadisEO-MOEO	Platypus
C4F1: graphical user interface										
C4F1a: design of experiments			✓	✓						
C4F1b: parametrization			✓	✓				✓		
C4F1c: execution control		✓		✓				✓		
C4F1d: visualization of results			✓	✓			✓	✓		
C4F2: benchmarks										
C4F2a: continuous problems	7	6	1	6	2	17	16	3	1	4
C4F2b: combinatorial problems	0	0	0	0	2	1	3	3	2	0
C4F3: quality indicators										
C4F3a: unary indicators	2	1	2	2	5	3	2	1	1	2
C4F3b: binary indicators	1	0	3	2	12	7	8	0	3	3
C4F3b: ternary indicators	0	0	0	0	1	0	0	0	0	0

Focusing on C4F3, JCLEC-MO stands out as the MOF providing the most extensive catalog of quality indicators in all categories. All frameworks provide methods to compute the hypervolume, which is probably the most frequently used quality indicator. Generational distance, inverted generational distance and ϵ_+ are the most popular binary indicators. JCLEC-MO is the only MOF implementing a ternary indicator, the so-called Relative Progress. This indicator, particularly well-suited to measure search convergence, quantifies the improvement between two generations in terms of generational distance. Overall, it is worth noting that MOFs provide a limited collection of quality indicators compared to those defined in the literature (M. Li & Yao 2019), especially in terms of diversity of the measured properties (convergence, spread, uniformity and cardinality). Besides, some of the indicators more frequently available, such as the generational distance, might lead to erroneous conclusions if they are not correctly interpreted. It would be desirable to include some guidance for users about their proper use.

5.2 | Documentation and community support (C5)

The availability of external documentation and additional resources is essential to popularize any tool or software library. Five features have been analyzed for characteristic C5:

- *C5F1: software license*. This feature details under which type of license the MOF is released. For those frameworks composed of several modules with different licenses, they all are reported.
- *C5F2: documentation*. The analysis of the documentation is based on the availability of tutorials (C5F2a), an API specification (C5F2b), reference manuals (C5F2c), code samples (C5F2d) and research papers (C5F2e).
- *C5F3: software update*. This feature informs about how often the MOF has been updated by showing the number of releases since January 2015.
- *C5F4: development facilities*. This feature indicates whether MOFs use public repositories (C5F4a) and compilation or distribution mechanisms (C5F4b).
- *C5F5: community*. Contact email (C5F5a), forums or mailing lists (C5F5b) and issue trackers (C5F5c) are the communication channels currently offered by MOFs.

TABLE 9 Coverage of C5: external support and community.

	DEAP	**ECJ**	**EvA**	**HeuristicLab**	**JCLEC-MO**	**jMetal**	**MOEA Framework**	**Opt4J**	**ParadisEO-MOEO**	**Platypus**
C5F1: software license	LGPL	Acad. Free	LGPL	GPL	GPL	MIT	LGPL	LGPL	CeCill/LGPL	GPL
C5F2: documentation										
C5F2a: tutorials	✓	✓	✓	✓	✓		✓	✓	✓	
C5F2b: API	✓	✓	✓		✓	✓	✓	✓	✓	
C5F2c: reference manuals		✓		✓	✓					
C5F2d: code samples	✓		✓	✓	✓		✓			✓
C5F2e: research papers	✓	✓	✓	✓	✓	✓		✓	✓	
C5F3: software update	3	5	1	5	1	9	10	2	0	4
C5F4: development facilities										
C5F4a: public repositories	G	G	G	G, S	G	G	G	G, M	G	G
C5F4b: comp. / distrib.	Pi	Mk	Mv	Ms		Mv	At, Mv	Gr, Mv	Mk	Ac
C5F5: community										
C5F5a: contact email		✓		✓			✓	✓	✓	
C5F5b: forum / mailing list			✓	✓				✓	✓	
C5F5c: issue tracker	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

G: Git, M: Mercurial, S: SVN; At: Ant, Ac: Anaconda, Gr: Gradle, Mk: Makefile, Ms: MS Build, Mv: Maven, Pi: pip

Table 9 compiles the information referred to the features of C5. Most of the MOFs (70%) are distributed under a GNU license, but other open source licenses like MIT and *Academic Free* are found too. Focusing on the documentation (C5F2), tutorials, research papers and API specifications appear as the most common information assets. ECJ, with two decades of development behind it, and HeuristicLab, launched in 2002, are the MOFs with the largest amount of resources available at their website. More recent MOFs have a significant lack of precise information concerning design aspects, i.e., reference manuals. Sometimes, these aspects are only partially revealed inside other documents. For instance, the architecture of jMetal is briefly described within the online user manual,[8] containing some incomplete sections yet. MOEA Framework has a "quick start" guide, but the full manual user is not available for free. Similarly, complete code examples are rarely available for download, as they are usually distributed with the core package or textually described in tutorials. ECJ (Luke 2017) (16 cites[9]), EvA (Kronfeld et al. 2010) (47) and Opt4j (Lukasiewycz et al. 2011) (136) have been presented in research forums like conferences and workshops. Journal papers or book chapters describing the architecture and core functionalities of DEAP (Fortin et al. 2012) (581), HeuristicLab (Wagner et al. 2014) (51), JCLEC-MO (Ramírez et al. 2019) (1), jMetal (Durillo & Nebro 2011) (644) and ParadisEO-MOEO (Liefooghe et al. 2011) (41) can be found too.

A regular update of a MOF is required for maintainability reasons, competitiveness with other alternatives, and novelty with respect to the advances in research within the area of knowledge. MOEA Framework and jMetal are the MOFs that have launched more releases in the last five years (C5F3). Scheduling at least one release per year seems to be the most common release plan, a practice followed by ECJ, HeuristicLab, jMetal and MOEA Framework. JCLEC-MO (2018) and Platypus (2018) are the MOFs whose first release appeared more recently.

Source code repositories constitute a key resource to track the evolution of software tools and coordinate the coding process, especially when the development policy looks for opening the project and involving the community and practitioners. It is also the case of MOFs, whose source code is hosted in at least one public code repository (C5F4a). Git stands out as the preferred version control system in nine out of the ten projects. Mercurial (Opt4j) and SVN (HeuristicLab) are other alternatives. Similarly, MOFs tend to provide support to the compilation process (C5F4b), Maven being the most popular among Java frameworks. Four out of the six Java MOFs (EvA, jMetal, MOEA Framework and Opt4J) submit their releases to

[8]`https://github.com/jMetal/jMetalDocumentation` (Accessed May 22th, 2020)
[9]Source: Scopus (May 22th, 2020)

the Maven dependency repository. Among the MOFs written in Python, we found that DEAP can be installed via pip tool, and Platypus is available on Anaconda.

Focusing on response to the community (C5F5), the development teams behind consolidated frameworks often let users to contact them via email or forums. HeuristicLab, Opt4J and ParadisEO-MOEO provide both communication channels. However, DEAP, JCLEC-MO and Platypus do not indicate any contact channel or mailing list. With the wider adoption of version control systems, issue trackers have become a common way to report bugs and pull requests. Since all MOFs are hosted on GitHub, the number of issues and pull requests[10] in this platform allows us to compare their development activity. DEAP (128 open issues/209 closed issues/26 pull requests), MOEA Framework (101/125/3), jMetal (93/128/29) and Platypus (70/57/3) are the frameworks that register more issues and/or requests. jMetal (3302), DEAP (2133) and MOEA Framework (886), together with ECJ (1083) and ParadisEO (5333) are the projects with highest number of commits. Only HeuristicLab (last commit in 2019), EvA (2016) and Opt4J (2018) do not have activity in GitHub during 2020. JCLEC-MO source code has been recently added to this platform in 2020, and no activity can be tracked at the moment. It should be noted that HeuristicLab only uses GitHub to host stable versions, and it has its own issue tracker to control changes.[11]

6 | SOFTWARE ANALYSIS

This section takes an in-depth look at the internal structure and implementation of the MOFs under analysis, as well as their behavior at runtime in terms of execution time and memory consumption.

6.1 | Analysis of the software implementation (C6)

C6 seeks to provide a complementary view of MOFs, focusing on more technical aspects related to their implementation. It implies looking into their source code and trying to find out what technical decisions associated with design and coding could influence the choice of one MOF over another. The analysis of the software implementation is founded on the following features:

- *C6F1: code implementation and execution*. This feature details the programming language (C6F1a) and the execution platform (C6F1b).
- *C6F2: external libraries*. This feature focuses on third-party libraries, classified according to their purpose: configuration (C6F2a), graphics (C6F2b), mathematical processing (C6F2c), programming support (C6F2d), reporting (C6F2e), statistical analysis (C6F2f) and testing (C6F2g).
- *C6F3: software metrics*. The set of metrics to be evaluated is comprised of code metrics (C6F3a), complexity metrics (C6F3b), testing metrics (C6F3c) and documentation metrics (C6F3d).

Table 10 compiles the results for C6F1 and C6F2. On the one hand, it can be noted that Java and Python—two platform-independent languages—appear more often. They are also the best positioned object-oriented programming languages in the TIOBE index.[12] C++ and C#, two other top programming languages according to the TIOBE ranking, are also considered in the metaheuristic field. However, users of HeuristicLab should know that running it on Linux—a common operating system of experimental environments—is only partially supported by means of the *Mono* tool, and the parallelization module of ParadisEO is not fully portable to Windows environments. It is worth mentioning that there are versions of jMetal in Python and C++, but they still provide partial functionality. On the other hand, the analysis of C6F2 reveals that the use of external libraries is a common practice to perform complex mathematical calculations and facilitate the programming process. MOFs also rely on external libraries to process inputs and outputs, particularly when some kind of file management is required. Those MOFs having GUI and distributing testing packages also tend to use third-party libraries to create graphics and define test cases, respectively. jFreeChart (graphics), Apache Commons (configuration, mathematical processing and programming support) and jUnit (testing) are examples of commonly used libraries.[13] Observe that the use of third-party libraries imposes certain dependencies on MOFs. Given that MOFs are often distributed as open source software (see C5F1), their dependencies are available under similar licenses. The use of build tools like Maven for Java or Anaconda for Python (see C5F4b) makes the configuration and update of dependencies transparent to the user, as an attempt to simplify the direct import and handling of the external libraries.

[10] Up to May 28th, 2020.
[11] `https://dev.heuristiclab.com/trac.fcgi/wiki/ChangeLog` (Accessed May 22th, 2020)
[12] May 2020 ranking of programming languages: `https://www.tiobe.com/tiobe-index/` (Accessed May 22th, 2020)
[13] See the additional material in page 26 for the full list of libraries per MOF.

TABLE 10 Coverage of C6F1: code implementation and execution, and C6F2: external libraries.

	DEAP	ECJ	EvA	HeuristicLab	JCLEC-MO	jMetal	MOEA Framework	Opt4J	ParadisEO-MOEO	Platypus
C6F1: code implementation and execution										
C6F1a: programming lang.	Python	Java	Java	C#	Java	Java	Java	Java	C++	Python
C6F1b: execution platform	L, M, W	L, M, W	L, M, W	M, W	L, M, W	L, M, W	L, M, W	L, M, W	L, M	L, M, W
C6F2: external libraries										
C6F2a: configuration			✓		✓		✓			
C6F2b: graphics		✓		✓		✓	✓		✓	✓
C6F2c: mathematical proc.	✓	✓	✓	✓		✓	✓	✓		
C6F2d: programming sup.	✓	✓		✓	✓	✓	✓	✓		✓
C6F2e: reporting		✓		✓	✓	✓				
C6F2f: statistical anal.					✓					
C6F2g: testing		✓	✓		✓	✓	✓	✓		

L: Linux, M: MacOS, W: Windows

Focusing on the MOF implementation, Table 11 shows the values for the different metrics that have been directly extracted from the source code (C6F3a). The symbol "-" indicates that the corresponding value is not available. Regarding its size in terms of lines of code (LOC) without comments, HeuristicLab stands out as the largest MOF, with one order of magnitude difference. Nevertheless, its most recent releases extend the scope of this framework beyond metaheuristic optimization, with strong emphasis on data analysis functionalities. Python libraries, i.e. DEAP and Platypus, are considerably lighter compared to the rest of MOFs, though it might be attributed to the compactness of Python compared to other languages like Java. Size is not the only factor that impacts the maintainability of a MOF. Code clones, i.e. pieces of code that repeatedly appear across the program, are known to be difficult to detect and fix (Chatterji, Carver, Kraft, & Harder 2013). The lowest percentage of duplication has been obtained for Opt4J, the smaller among Java MOFs, but it has a greater LOC value than Python libraries. MOEA Framework and HeuristicLab also present good ratios compared to their size.

Differences also arise regarding code organization, which is reflected in the number of directories or packages. For instance, jMetal and ECJ distribute their code into a greater number of packages than EvA or ParadisEO-MOEO, even when they declare more classes. Another development decision to be highlighted is the variety in the use of abstraction and interface definition. HeuristicLab is characterized by a more interface-oriented design. Among Java frameworks, EvA and jMetal make an intensive use of interfaces, whilst JCLEC-MO presents the highest percentage of abstract classes (14%). These differences might be imperceptible for users that only configure or instantiate MOF functionalities. However, the relation of interfaces and abstract classes is relevant for those programmers that want to extend the framework. Notice that the fact that C# and Java do not allow multiple inheritance may lead to an increase in the number of interfaces. The ratio between functions (methods) and classes gives some idea of how modular MOFs are. EvA (12.16) and DEAP (10.19) present the highest average of functions per class, in sharp contrast to Opt4j (4.02) and Platypus (3.97).

Two metrics are calculated to analyze the complexity of MOFs (C6F3b). On the one hand, the cyclomatic complexity, as originally defined by T.J. McCabe, is based on the number of paths through the code. On the other hand, the so-called cognitive complexity values the difficulty to understand the code flow by using a mathematical model based on how intuitive loops, conditionals or recursion are for programmers.[14] Both metrics are unbounded, but their values serve in practice to establish a comparative framework whose outcomes reveal that larger MOFs, such as HeuristicLab and EvA, present a greater cyclomatic complexity. In our analysis, MOFs with less cyclomatic complexity are usually less difficult

[14]Details about how this metric is computed can be found at SonarQube documentation website: https://docs.sonarqube.org/latest/user-guide/ metric-definitions/ (Accessed May 22th, 2020).

TABLE 11 Coverage of C6F3: software metrics.

	DEAP	ECJ	EvA	HeuristicLab	JCLEC-MO	jMetal	MOEA Framework	Opt4J	ParadisEO-MOEO	Platypus
C6F3a: code metrics										
Lines of code (KLOC)	3.8	54.0	84.0	798.2	30.5	45.1	34.5	26.2	57.2	5.7
Duplicated lines (%)	1.6	8.0	10.4	8.8	12.4	17.7	3.6	1.2	8.5	5.1
Directories or packages	2	109	63	-	63	154	96	50	107	5
Classes	36	637	736	8,125	414	629	508	552	1,289	166
Abstract classes	-	36	33	547	58	28	35	41	343*	-
Interfaces	-	30	96	930	41	62	24	66		-
Functions	367	3,496	8,947	76,329	2,827	3,488	2,956	2,220	5,600	659
C6F3b: complexity metrics										
Cyclomatic complexity	971	11,167	18,152	178,893	5,435	7,614	6,658	4,582	-	1,457
Cognitive complexity	1,154	13,162	16,820	15,5964	5,279	6,908	5,789	3,990	-	1,585
C6F3c: testing metrics										
Lines to cover (KLOC)	3.4	33.3	50.3	448.1	19.3	34.6	33.1	13.9	43.5	5.0
Condition coverage (%)	-	9.8	0.1	-	23.2	19.8	63.6	1.1	54.6	-
Line coverage (%)	6.4	10.2	0.3	15.1	23.9	17.3	70.4	1.4	33.3	16.9
Coverage (%)	6.4	10.0	0.3	15.1	23.8	17.9	68.4	1.3	41.2	16.9
C6F3d: documentation metrics										
Density of comments (%)	47.3	30.1	20.9	18.3	35.4	16.6	32.4	33.1	-	15.5

*Number of code files in which at least one virtual method is declared.

to understand. The only exceptions are DEAP, ECJ and Platypus, though the Python libraries are clearly less complex. Even though the cognitive complexity metric was designed to be language-agnostic, the high-level, lightweight syntax of Python can contribute to reduce the need of nesting or create long blocks of code.

As for testing metrics (C6F3c), reaching high coverage is desired to mitigate the presence of bugs, but the effort required is proportional to the number of conditions and LOC. Despite the presence of randomized behaviors within some components of the MOFs, what might be more difficult to test, it is important to guarantee that the algorithms behave as expected in order to trust their results. Table 11 details the number of lines that can be covered by test cases, which varies between the 53% (Opt4J) and the 96% (MOEA Framework) of total LOC. Two different coverage metrics are calculated for the functional code, namely condition coverage and line coverage. The former reports the percentage of conditions that are covered, whereas the latter indicates the percentage of code lines executed by unit test cases. Only MOEA Framework and ParadisEO-MOEO achieve more than 50% of coverage for at least one metric. In general, both types of coverage seem to be equally important for development teams. As for the overall coverage, which consider all types of statements, MOEA Framework stands out as the MOF with the highest percentage of tested code, followed by ParadisEO-MOEO. In light of these numbers, it seems that development effort has been more focused on implementing new functionalities. However, we observe that those frameworks that have released new versions in the last years, i.e. ECJ, JCLEC-MO, jMetal and specially Platypus, devote more effort to testing code.

Finally, the density of comments gives a precise idea of the extend to which MOFs have been internally documented (C6F3d). This aspect is especially relevant for researchers who need to understand the internal details of the algorithm implementations. The percentage of comments, which also includes commented-out code (i.e. blocks of code under comments), varies between 15.5% (Platypus) and 47.3% (DEAP). Notice that these MOFs are the smallest in terms of LOC, which clearly indicates that the decision of including more or less comments is not only a matter of the number of classes and functions. As happens with testing practices, a good documentation becomes particularly relevant as the MOF increases its size. In general, we speculate that MOFs still should devote even more effort to documenting their code, since the average number of commented lines is below 30%.

6.2 | Analysis of performance at runtime (C7)

Understanding how frameworks behave under diverse execution conditions and how they manage computing resources provide an interesting insight into their use and performance. Here, a study of execution time and memory consumption is carried out as a way to complement the static information already given. Next, the most relevant results of both experiments are presented and discussed below.

6.2.1 | Experiment #1: Comparison of algorithms and benchmarks

TABLE 12 Experiment #1: Parameter values.

Parameter	Value	Parameter	Value
	General parameters		
Population/swarm size	100	Crossover (binary)	1-Point
No. of generations	100	Crossover (double)	SBX
Crossover probability	0.9	Mutation (binary)	Uniform Bit flip
Non-uniform mutation prob.	0.1	Mutation (double)	Uniform Polynomial
Uniform mutation prob.	1/genotype-length		
SPEA2		IBEA	
Population size (P)	50	Parent selector	Binary tournament
Archive size (A)	50	Fitness indicator	Hypervolume
Parent selector	Binary tournament	Scaling factor (κ)	0.05
k-neighbours	$\sqrt[2]{\mathrm{P}+\mathrm{A}}$	Reference point (ρ)	2
MOEA/D		OMOPSO	
Neighbourhood size (τ)	10	Archive size	100
Max. No. of replacements (nr)	2	Non uniform mut. prob.	0.1
Evaluation function	Tchebycheff	Uniform mut. prob.	1/particle-length
Weights	Uniformly generated	ϵ-dominance (ϵ values)	0.0075

The goal of this experiment is to study the behavior of MOFs by simulating how domain experts would usually work with them. A diverse collection of algorithms and benchmarks is required to draw some general conclusions without introducing subjectivity or bias to the analysis. Looking at the outcomes of C1F2 (see Section 3.2), we found that only two algorithms are available in the majority of MOFs: NSGA-II, implemented by all MOFs; and SPEA2, only missing in HeuristicLab. Using these two algorithms would not allow us to cover a variety of families, because they both belong to the second generation. We also observe that some MOFs —particularly those with a more specialized scope— share some other algorithms in their catalog that could serve our purposes. For these reasons, we take the decision of comparing MOFs in two groups. Group #1 is comprised of DEAP, ECJ, EvA, HeuristicLab, Opt4J and ParadisEO-MOEO, for which we will study NSGA-II and, when possible, SPEA2. JCLEC-MO, jMetal, MOEA Framework and Platypus constitute Group #2, for which algorithms from three additional families—indicator-based (IBEA), decomposition (MOEA/D) and relaxed dominance (OMOPSO)—are available for comparison.

Each selected algorithm is combined with several benchmarks. As a result, we increase the number of samples by changing the type of problem, i.e. continuous and combinatorial, and the number of objectives. We presume that domain experts would be more interested in problems with two to six objectives, leaving highly-dimensional problems for the second experiment. The selected continuous benchmarks are DTLZ and ZDT, and LOTZ and KP as combinatorial problems (see Table 13 for a short description and references of their formulations). A greater number of functions from DTLZ and ZDT are considered for Group #1 to compensate the lack of algorithms. DTLZ and ZDT are available in the majority of MOFs (see C4F2 in Section 5.1). The implementation of DTLZ was required for ECJ and EvA. ZDT was also developed for EvA. Combinatorial problems are less common, and only MOEA Framework include both LOTZ and KP, while JCLEC-MO implements KP. Even so, we consider these problems because of their popularity. Furthermore, implementing them in the rest of MOFs is straightforward.

TABLE 13 Experiment #1: Selected benchmarks.

Benchmark	Description	Reference
DTLZ1	Minimize a variable number of functions. The problem has a linear Pareto front (PF).	(Deb, Thiele, Laumanns, & Zitzler 2005)
DTLZ2	Minimize a variable number of functions. The PF is the first quadrant of a sphere.	(Deb et al. 2005)
ZDT1	Minimize two functions. The PF is convex.	(Zitzler, Deb, & Thiele 2000)
ZDT4	Minimize two functions. The problem is multimodal, i.e. it has local PFs.	(Zitzler et al. 2000)
ZDT6	Minimize two functions. PF with non-uniformly distributed solutions.	(Zitzler et al. 2000)
LOTZ	Maximize the number of leading '1' and and trailing '0' in a bit array.	(Zitzler, Laumanns, & Bleuler 2004)
KP	Find the subset of items that maximizes the profits of k knapsacks.	(Zitzler & Thiele 1999)

As for the configuration of the algorithms, Table 12 shows the list of parameters and their values, including genetic operators. Since the purpose of the experiment does not concern how well algorithms solve an optimization problem, standard values for general parameters like the population size and operator probabilities are considered. Crossover and mutation operators are chosen depending on the required encoding among the most commonly available (see C1F3 in Section 3.3). Only a few developments were needed: SBX for HeuristicLab and JCLEC-MO, one point crossover for Platypus, and UMP for EvA, HeuristicLab and ParadisEO-MOEO. As for the specific parameters, default values provided by the original authors are applied.

Figures 1 and 2 show the results of Experiment #1 for Group #1 and Group #2, respectively. Values represent the average execution time expressed in seconds, and errors bars indicate the standard deviation. The average ranking position of each MOFs after sorting each combination of algorithm and problem in decreasing time is shown in the legend. Focusing on Group #1, ECJ provides the best overall performance, obtaining 1.625 in the ranking. With the exception of four configurations, it always completes the search process in less than 1s. Opt4J, at the second position of the ranking, presents quite stable execution times, but some increase is observed when solving problems with more than two objectives. With a similar ranking, ParadisEO-MOEO has a more variable behavior depending on the algorithm. On average, it is faster than ECJ when only SPEA2 results are compared, and barely increases the execution time when solving the KP. Similar to Opt4J, EvA shows a quite stable behavior but always needs more than 1s to conclude. The implementation of NSGA-II in HeuristicLab does not experience great changes across configurations, with times similar to those obtained by EvA. However, it should be noted that HeuristicLab is running on top of Mono, a tool that transforms the code to make it compatible with Linux. Thus, its performance might be somehow affected compared to the native implementation for Windows. Finally, DEAP is the framework reporting higher execution times, even reaching values close to 1 minute for some executions of SPEA2.

As for Group #2, differences among Java frameworks are not so evident (see Figure 2). jMetal and MOEA framework obtain the same average ranking, with all executions finished in less than 3s. jMetal is faster than MOEA Framework for MOEA/D and OMOPSO, while MOEA Framework is the fastest MOF for NSGA-II and IBEA. In light of the results, we speculate that these MOFs are more affected by the choice of the algorithm than the problem under resolution. Next in the ranking, JCLEC-MO also shows differences depending on the selected algorithm. Particularly, the execution time increases more than 1s when IBEA is chosen. Lastly, Platypus is the framework reporting the worse performance, requiring several seconds or even minutes to conclude all executions. As opposed to the three Java frameworks, the implementation of NSGA-II is significantly slower. Besides, it seems to be specially affected by the optimization problem and the number of objectives.

We have observed that the type of algorithm and benchmark have different influence on MOFs. With the exception of Platypus, the implementation of NSGA-II is highly efficient in all MOFs compared to the rest of algorithms selected. Focusing on the benchmark, the type of problem has more impact than the number of objectives in ECJ (KP) and DEAP (KP and ZDT4). In contrast, Platypus responds worse when the number of objectives increases.

Along the experiment, RAM memory usage has been monitored at regular intervals. Since the tested configurations are quite standard and require only a few seconds to finish, high memory requirements are not expected. Nonetheless, notable differences arise when comparing frameworks developed in different programming languages. In general, MOFs written in Java have a minimum memory consumption of 12,000KB per run, whereas this minimum falls to around 5,000KB for those libraries developed in Python. HeuristicLab (C#) and ParadisEO-MOEO (C++) require a minimum on 6,700KB and 2,700KB on average, respectively. In fact, ParadisEO-MOEO is the MOF that offers a better memory management, as

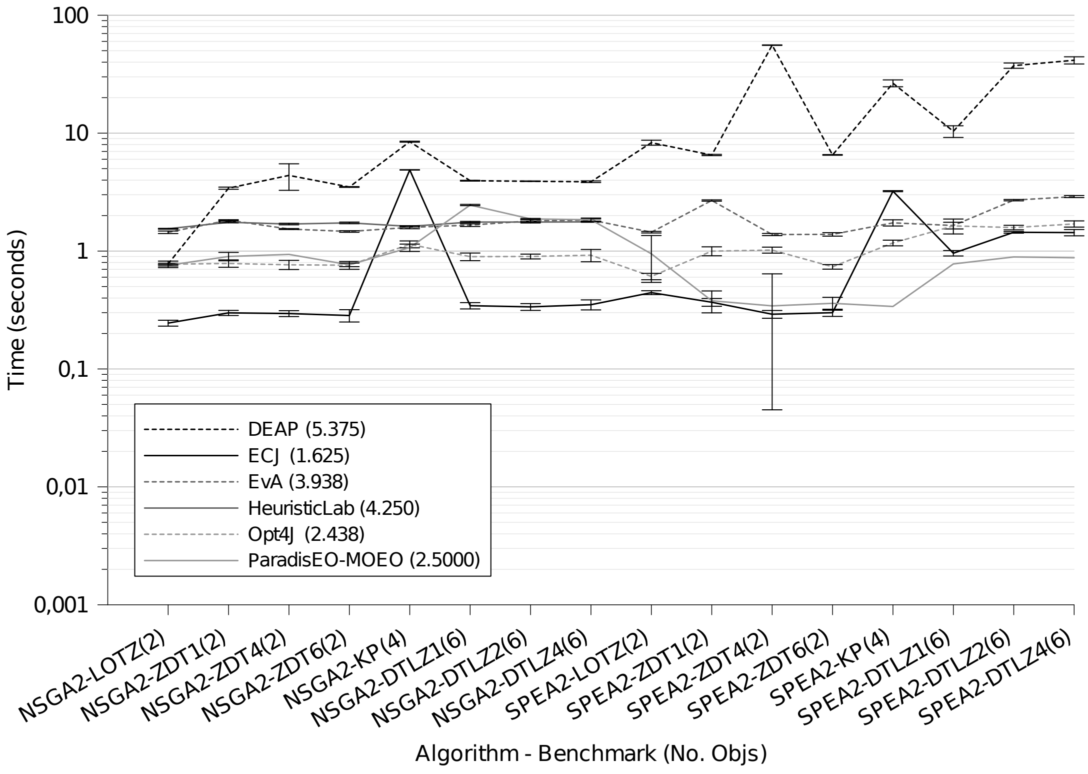

FIGURE 1 Experiment #1: Execution time for Group #1. Time axis in logarithmic scale.

the maximum value is quite stable and never exceeds 4,000KB. Other MOFs showing a similar behavior are the two Python libraries, i.e. DEAP (with a maximum of 20,764KB on average) and Platypus (24,086KB), EvA (56,459KB) and HeuristicLab (79,504KB). Opt4J also behaves very stably, but it usually requires a large amount of memory (average of 126,084KB). As for the rest of Java MOFs, the maximum memory usage is more dispersed, some reaching peaks of more than 100,000KB in JCLEC-MO (average of 76,130KB), jMetal (88,326KB) and MOEA Framework (89,005KB), and even more than 500,000KB in ECJ (149,265KB on average) for the two configurations solving the KP.

6.2.2 | Experiment #2: Scalability study

This experiment is intended to analyze how MOFs manage runtime resources when more complex configurations are demanded. Therefore, this experiment is more oriented to researchers who are interested in comparing the performance of algorithms under different parameter settings or in solving highly-dimensional problems. Scalability is tested in terms of three parameters: population size (100, 500, 1000), number of generations (100, 500, 1000, 5000) and number of objectives (2, 5, 10, 25, 50), resulting in 60 combinations in total. As a way to reduce the influence of any other aspect, an algorithm without parameters and available in all MOFs, NSGA-II, and a benchmark with configurable objectives, DTLZ1, were selected. Genetic operators and their parameters are the same than those used in the previous experiment (see Table 12).

With respect to the execution time, Table 14 summarizes the minimum and maximum average time (in seconds), together with the standard deviation, for each population size. Usually, these times correspond to the configuration with 100 generations and two objectives, and 5000 generations and 50 objectives, respectively. The last column in Table 14 shows the average ranking position per MOF considering all configurations, whose breakdown of values is detailed in the technical report (see additional material in page 26).

When studying each MOF separately, we can observed that ECJ is the Java framework that provides the best response to demanding configurations, getting the top position in the ranking. HeuristicLab shows a variable behavior, e.g. it does not provide the best performance when evolving 100 individuals, but it responds better than many other MOF for the remaining population sizes. This fact is reflected in the third position of the ranking (3.38). Both Opt4J (5.20) and EvA (7.13) show similar minimum times, although it is noted that EvA suffers severely as the number of generations and objectives increases. Likewise, ParadisEO-MOEO experiences the same problems as EvA, showing the worst performance, except for the Python libraries in almost all executions. MOEA Framework (2.53) is the fastest Java library after ECJ, followed by jMetal (3.98) and JCLEC-MO

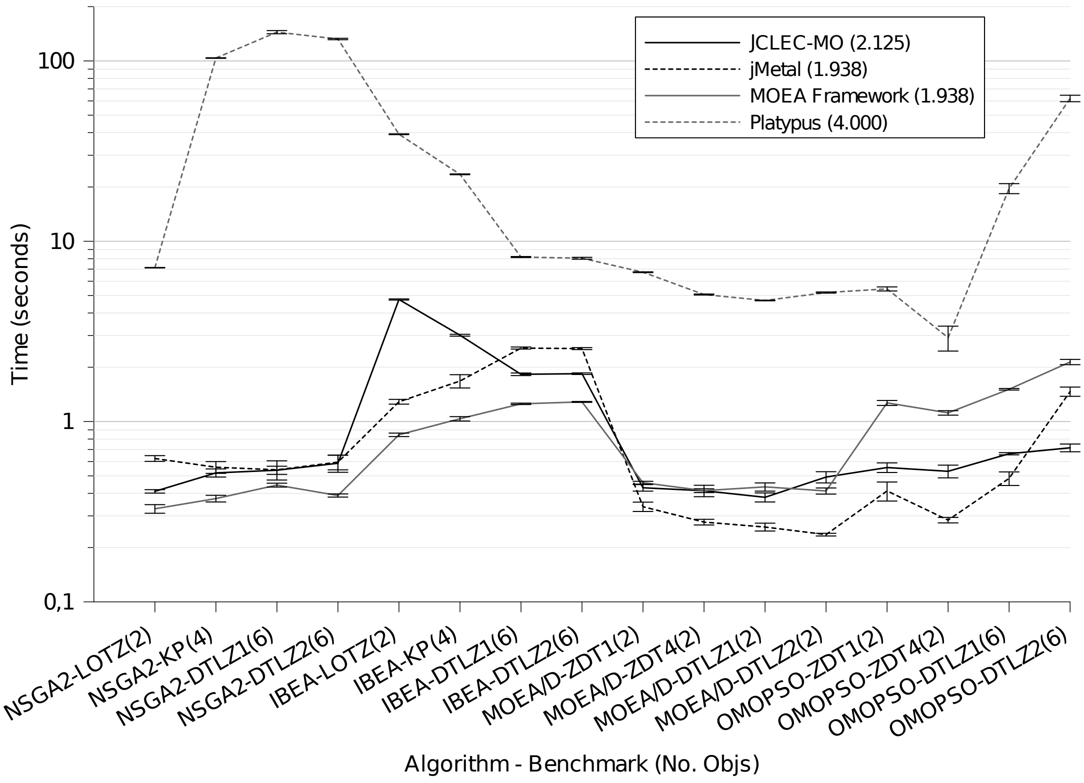

FIGURE 2 Experiment #1: Execution time for Group #2. Time axis in logarithmic scale.

TABLE 14 Minimum and maximum execution time in Experiment #2 (expressed in seconds).

Framework	Population size = 100		Population size = 500		Population size = 1000		Avg.
	Minimum	Maximum	Minimum	Maximum	Minimum	Maximum	ranking
DEAP	3.55 ± 0.12	588.13 ± 8.28	75.36 ± 0.84	$11,708.99 \pm 100.31$	298.55 ± 5.00	$48,052.73 \pm 1023.74$	**8.53**
ECJ	0.34 ± 0.04	14.37 ± 0.92	0.79 ± 0.05	170.61 ± 2.44	2.73 ± 0.09	589.33 ± 3.58	**1.13**
EvA	1.79 ± 0.02	226.77 ± 2.25	5.74 ± 0.62	$3,830.01 \pm 94.69$	16.38 ± 0.49	$16,030.96 \pm 512.88$	**7.13**
HeuristicLab	1.84 ± 0.13	106.45 ± 0.91	2.65 ± 0.02	311.71 ± 38.41	4.52 ± 0.05	680.33 ± 2.98	**3.38**
JCLEC-MO	0.64 ± 0.07	30.55 ± 0.58	3.25 ± 0.10	475.77 ± 4.83	11.86 ± 0.07	$1,611.35 \pm 91.90$	**4.78**
jMetal	0.58 ± 0.06	30.74 ± 1.21	2.64 ± 0.07	576.81 ± 3.86	9.01 ± 0.17	$2,237.61 \pm 30.34$	**3.98**
MOEA Fram.	0.37 ± 0.01	19.26 ± 0.21	1.43 ± 0.08	339.96 ± 1.59	4.27 ± 0.14	$1,284.00 \pm 7.17$	**2.53**
Opt4J	0.93 ± 0.08	39.18 ± 0.68	4.12 ± 0.09	533.96 ± 6.95	13.98 ± 0.31	$1,743.74 \pm 12.82$	**5.20**
Par.-MOEO	0.91 ± 0.01	208.66 ± 1.22	20.61 ± 0.27	$19,352.82 \pm 190.56$	82.06 ± 1.42	$69,172.09 \pm 716.24$	**8.38**
Platypus	5.23 ± 0.03	$1,437.75 \pm 38.90$	84.25 ± 1.09	$24,692.45 \pm 923.18$	314.05 ± 2.75	$91,748.88 \pm 1583.78$	**9.83**

(4.78). Python libraries always return minimum execution times higher than MOFs implemented in other languages. In this case, DEAP seems to guarantee better scalability than Platypus according to their respective maximum values. Looking at their average rankings, both libraries are far from the positions obtained by Java frameworks.

To better contextualize the differences among the MOFs and visualize how well they scale, Figures 3 and 4 show the average time for all configurations with population sizes 100 and 1000, respectively. As can be observed in Figure 3, the execution time of both Python libraries (DEAP and Platypus) has an exponential growth with respect to the number of objectives, especially when the number of generations is greater than 500. ParadisEO-MOEO also experiences some troubles to deal with demanding configurations, and it usually appears as the second slowest MOF when 50 objectives are configured (see Figure 4). Similarly, HeuristicLab (C#) seems to be more influenced by the number of objectives, which causes its

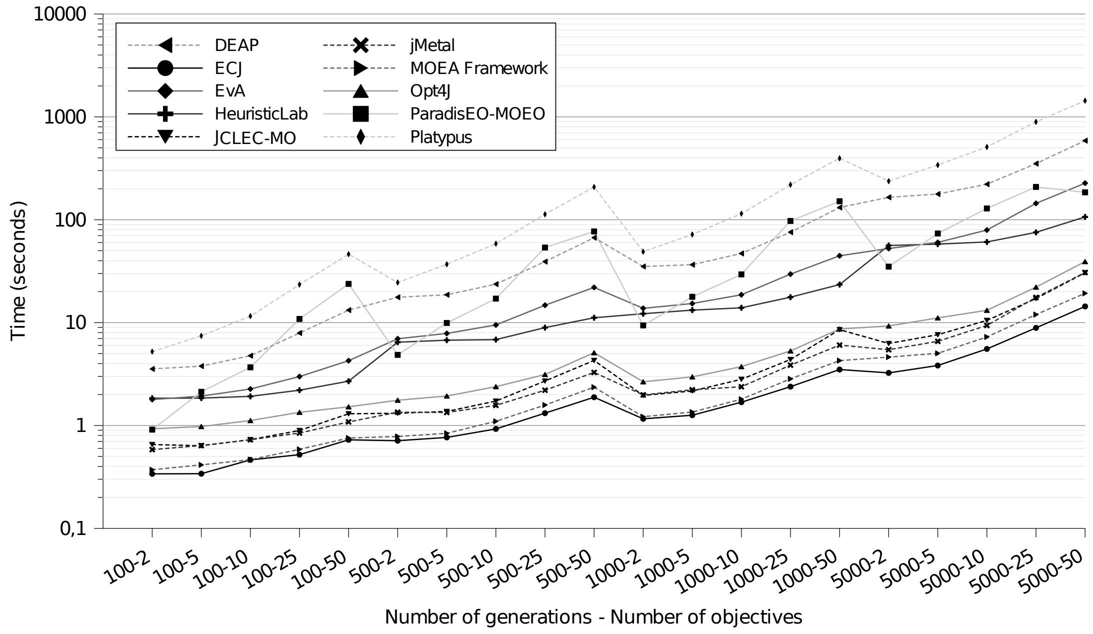

FIGURE 3 Execution time of Experiment #2 (population size = 100). Time axis in logarithmic scale.

deviation from Java MOFs. With the exception of EvA, all Java MOFs show good scalability and report acceptable execution times compared to those MOFs developed using other languages.

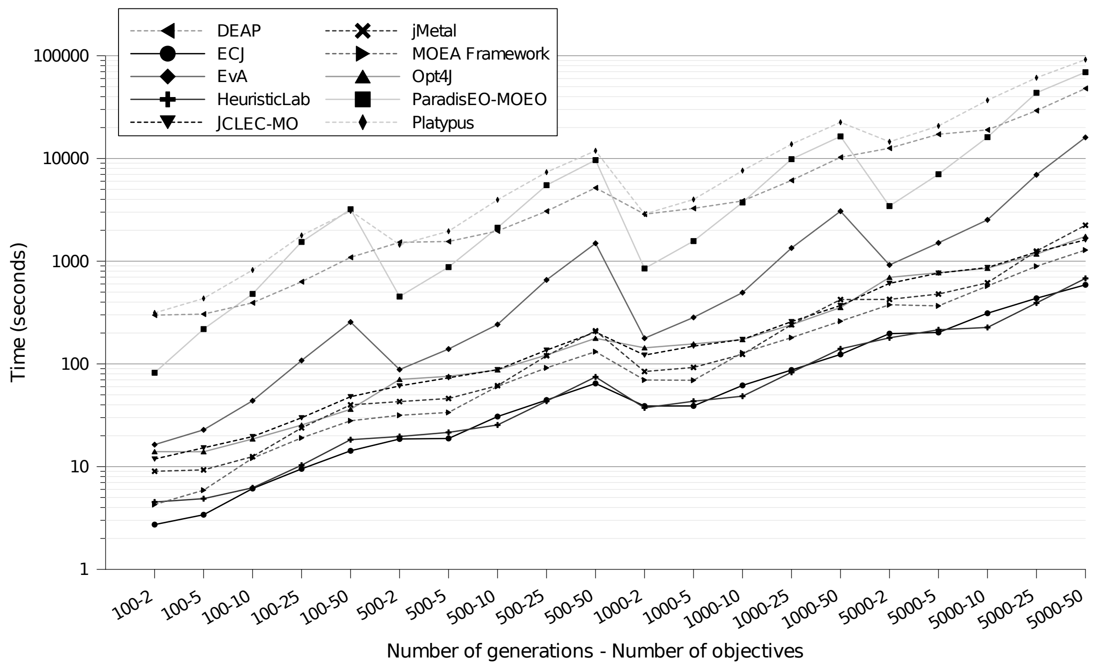

FIGURE 4 Execution time of Experiment #2 (population size = 1000). Time axis in logarithmic scale.

Regarding memory requirements, Figures 5 and 6 depict boxplots for the minimum and maximum memory consumption considering all configurations. Definitely, the programming language becomes a key factor here. Notice that ParadisEO-MOEO (C++) maintains highly stable values, always below 5,000KB. Both Python libraries, i.e. DEAP and Platypus, behave very similarly and they are not so far from the maximum memory values obtained by ParadisEO-MOEO. HeuristicLab (C#) provides a good memory management, but it still remains the most demanding framework

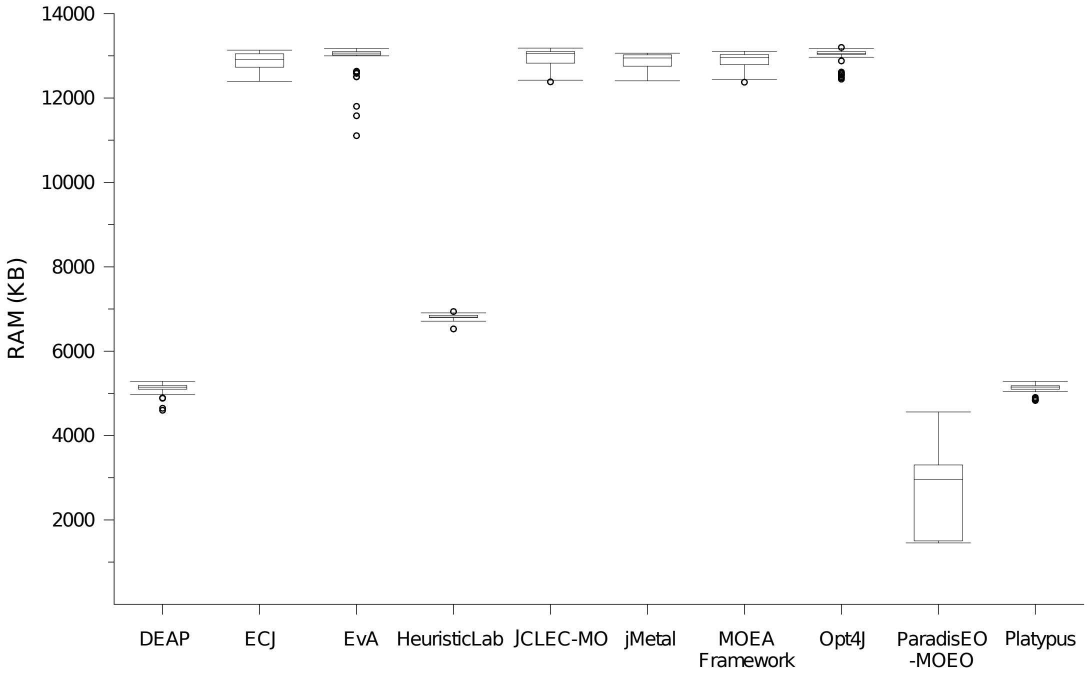

FIGURE 5 Minimum memory usage in Experiment #2.

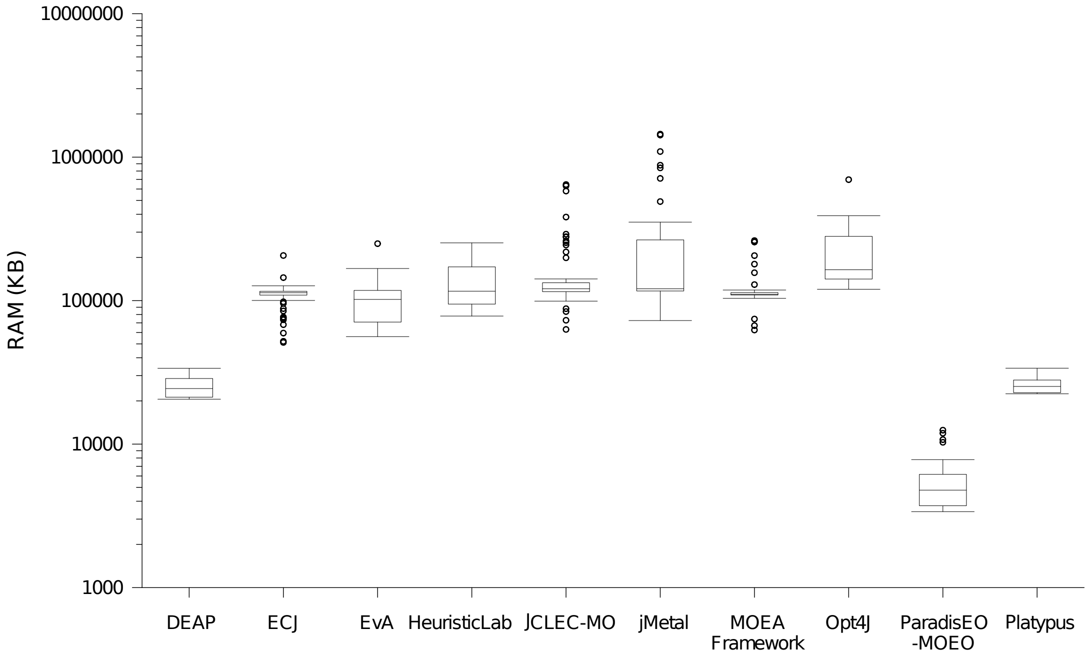

FIGURE 6 Maximum memory usage in Experiment #2. RAM axis in logarithmic scale.

for seven of the 60 configurations. Notice that MOFs implemented in Java require significantly more memory resources than the rest. On the one hand, ECJ, EvA and MOEA Framework seem to be the most competitive Java MOFs, never reporting global maximum values. On the other hand, JCLEC-MO and jMetal tend to experience "memory peaks" more often (see outliers in Figure 6), resulting in global maximum values for 1 and 13 configurations, respectively. Finally, this experiment reveals that Opt4J is the MOF demanding the largest amount of memory, reaching global maximum values of RAM usage for 65% of the configurations under evaluation.

TABLE 15 Summary of best frameworks for each feature.

Characteristic	Feature	Best framework
C1: search components and techniques	C1F1: types of metaheuristics	EvA
	C1F2: families of algorithms	jMetal
	C1F3: encodings and operators	HeuristicLab
C2: configuration	C2F1: inputs	EvA
	C2F2: batch processing	EvA, HeuristicLab, jMetal
	C2F3: outputs	MOEA Framework
C3: execution	C3F1: multi-thread execution	ECJ, Opt4J, ParadisEO-MOEO
	C3F2: distributed execution	ECJ, MOEA Framework
	C3F3: stop and restart mode	ECJ, HeuristicLab, MOEA Framework, Opt4J
	C3F4: fault recovery	ECJ, HeuristicLab, jMetal, MOEA Framework, ParadisEO-MOEO, Platypus
	C3F5: execution and control logs	ECJ, EvA, HeuristicLab, JCLEC-MO, jMetal, Platypus
C4: utilities	C4F1: graphical user interface	HeuristicLab
	C4F2: benchmarks	MOEA Framework
	C4F3: quality indicators	JCLEC-MO
C5: documentation and community support	C5F1: software license	See Table 9
	C5F2: available documentation	JCLEC-MO
	C5F3: software update	MOEA Framework
	C5F4: development facilities	Opt4J
	C5F5: community	HeuristicLab, Opt4J, ParadisEO-MOEO
C6: software implementation	C6F1: implementation and execution	DEAP, ECJ, EvA, JCLEC-MO, jMetal, MOEA Framework, Op4tJ, Platypus
	C6F2: external libraries	ECJ, JCLEC-MO, jMetal, MOEA Framework
	C6F3: software metrics	See Table 11
C7: performance at runtime	C7F1: execution time	ECJ
	C7F2: memory consumption	ParadisEO-MOEO

7 | LESSONS LEARNED AND OPEN ISSUES

In terms of the lessons learned during the MOF evaluation process, we have been able to compile some common essential and use aspects. First lesson is related to the *definition of genetic operators*. The compilation and classification of genetic operators resulted in a hard task due to the lack of a common naming policy and external references. Secondly, multi-purpose frameworks have usually not gone through a true *adaptation to MOO*. In fact, the documentation available for multi-purpose frameworks is strongly oriented towards single-objective optimization. As a general rule, domain experts cannot be certain about whether it is possible to adapt some functionalities to MOO, or how to proceed. Third lesson is related to the *configuration and execution alternatives* provided by the MOF. As an example, external documentation tends to be focused on one single type of input system, i.e., GUI, files or code. And they do not always guarantee that the same configuration options can be set for all types of input system. Also, the possibility of execution control—e.g. checkpoints or setting default values— can vary depending on whether the GUI is used or not. Finally, the forth lesson learned concerns some *developmental assumptions*. During the experimental design, a carefully inspection of the code revealed that not all frameworks provide exactly the same implementation of a given algorithm. Also, default configurations do not always correspond to the original algorithm. Solving these issues required us to revise the description of the algorithms in their original papers, and perform minor changes in their code or configuration accordingly.

From the previous sections, we can also learn that there is not “the best framework”, but still observe strengths and weaknesses for each one. Table 15 compiles the frameworks that best support each feature under analysis. More specifically, the framework(s) satisfying more items of the corresponding checklists are listed in the last column. The resulting table serves as a quick reference for the user to determine which MOF best suits his/her specific needs, depending on what aspects he/she wants to focus on the development.

The development of this analysis, together with its experimentation, has allowed us to identify gaps and open problems that are worth mentioning. Likewise, we have learned some lessons that we also find interesting to share later. Firstly, the open issues are listed next:

1. *Update of multi-purpose MOFs*. Resources for MOO in multi-purpose MOFs are still scarce in comparison with other specialized frameworks and libraries. This issue is especially critical with regard to the novelty of the supported algorithms. In fact, multi-purpose frameworks offer fundamentally first and second generation algorithms.

2. *Support to different metaheuristics*. The lack of variety with respect to metaheuristic paradigms is a major issue across all frameworks, but it becomes even more evident in popular frameworks with years of development, such as ECJ and HeuristicLab. In fact, these MOFs implement other metaheuristics for single-objective problems: ant colony optimization, GRASP and PSO are available in ECJ, while HeuristicLab includes PSO and several single-solution-based metaheuristics. Efforts should be directed at enabling the development of solutions based on swarm intelligence and single-solution-based metaheuristics, allowing the addition of new proposals as the state of the art advances.

3. *Parallelism and distribution*. The growing interest in solving problems with a high number of decision variables, objectives or constraints, makes necessary to consider advanced computing models. The appearance of metaheuristics specially designed to be run in parallel (Alba 2005), as well as new alternatives to massive processing like GPUs (Talbi & Hasle 2013) or Hadoop (Ferrucci, Kechadi, Salza, & Sarro 2013) still need to be considered in the near future.

4. *Lack of documentation*. Advanced features do not often appear in tutorials, while design decisions are seldom explained. Consequently, a manual inspection of code is often required to fully understand the capabilities of the framework.

5. *Test cases and error handling*. Our software analysis reveals low levels of code coverage in test cases, which are obsolete in some cases. In addition, not all MOFS intuitively respond to configuration errors, and only a few give proper feedback.

6. *GUI support*. Special attention should be paid on providing users with a fully functional graphical environment, including experiment management and visualization of results.

8 | CONCLUDING REMARKS

This paper has presented a thorough evaluation of metaheuristic optimization frameworks than can be used to solve MOO problems. More specifically, ten frameworks have been analyzed in terms of seven characteristics and twenty four features, focused on the available search techniques, configuration and execution capabilities, and external support. The analysis of the source code provides additional information about code maintainability and usability, what helps users to understand how frameworks can be extended. The assessment process has been completed with an extensive experimental study to analyze the performance of these tools in different usage scenarios.

The extraction, analysis and discussion of all these characteristics do not only provide domain experts with practical guidance to start using MOFs, but also constitutes a reference source for researchers who want to choose one for these toolkits. Moreover, it can serve as a reflection for the teams behind their development, as it highlights the current state of similar tools and poses open issues that might be addressed in the future. As an ultimate goal, this paper encourages those users identified with any of these roles to get involved in the provision and enhancement of software tools for MOO.

The information extracted from characteristics C1 to C6 reveals some differences between multi-purpose MOFs and multi-objective-specific libraries. Probably due to their maturity, multi-purpose MOFs tend to provide more basic functionalities and reusable elements like genetic operators. On the contrary, specialized libraries tend to offer up-to-date algorithms and utilities like benchmarks and indicators. Some of the selected MOFs are devoting important efforts to provide the community with new updates, development facilities and external support. However, there are also some weak points, such as the lack of fully functional graphical environments and the limited support to parallelism and distribution. Our experimental study has revealed how MOFs respond to a variety of configurations, thus providing a hitherto unanalyzed perspective. We also detected some important differences between implementations of the same algorithm in different programming languages, as well as experimental evidences revealing that some frameworks experience troubles to control execution time and manage memory.

DATA AVAILABILITY STATEMENT

The data that supports the findings of this study are available in the supplementary material of this article.

ADDITIONAL MATERIAL

A technical report containing detailed information about the evaluation of each characteristic, including the decisions taken about the partial fulfillment of features and any other relevant, extensive information gathered along the analysis, is publicly available from the website `https://www.uco.es/kdis/mofs-multiobjective`

ACKNOWLEDGMENTS

This work was supported by the Spanish Ministry of Economy and Competitiveness [project TIN2017-83445-P], the Spanish Ministry of Education under the FPU program [grant FPU17/00799], the University of Córdoba [grant "Plan propio 2019 - mod. 2.4"], and FEDER funds.

Conflict of interest

The authors declare no potential conflict of interests.

References

Alba, E. (2005). *Parallel Metaheuristics: A New Class of Algorithms*. Wiley.

Boussaïd, I., Lepagnot, J., & Siarry, P. (2013). A survey on optimization metaheuristics. *Inf. Sci.*, *237*(0), 82–117. doi: 10.1016/j.ins.2013.02.041

Chatterji, D., Carver, J. C., Kraft, N. A., & Harder, J. (2013). Effects of cloned code on software maintainability: A replicated developer study. In *Proceedings of the 20th working conference on reverse engineering (wcre)* (pp. 112–121). doi: 10.1109/WCRE.2013.6671286

Coello Coello, C. A. (2003). Evolutionary Multiobjective Optimization: Current and Future Challenges. In *Advances in soft computing* (pp. 243–256). Springer London. doi: 10.1007/978-1-4471-3744-3_24

Coello Coello, C. A., Lamont, G. B., & Van Veldhuizen, D. A. (2007). *Evolutionary Algorithms for Solving Multi-Objective Problems* (2nd ed.). Springer. doi: 10.1007/978-0-387-36797-2

Deb, K., & Jain, H. (2014). An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point-Based Nondominated Sorting Approach, Part I: Solving Problems With Box Constraints. *IEEE Trans. Evol. Comput.*, *18*(4), 577–601. doi: 10.1109/TEVC.2013.2281535

Deb, K., Pratap, A., Agarwal, S., & Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans. Evol. Comput.*, *6*(2), 182-197. doi: 10.1109/4235.996017

Deb, K., Thiele, L., Laumanns, M., & Zitzler, E. (2005). Scalable Test Problems for Evolutionary Multiobjective Optimization. In *Evolutionary multiobjective optimization: Theoretical advances and applications* (pp. 105–145). Springer London. doi: 10.1007/1-84628-137-7_6

Durillo, J. J., & Nebro, A. J. (2011). jMetal: A Java framework for multi-objective optimization. *Adv. Eng. Softw.*, *42*(10), 760–771. doi: 10.1016/j.advengsoft.2011.05.014

Ferrucci, F., Kechadi, M. T., Salza, P., & Sarro, F. (2013). *A Framework for Genetic Algorithms Based on Hadoop* (Tech. Rep.). University of Salerno, University College Dublin and University College London: arXiv:1312.0086.

Fortin, F.-A., De Rainville, F.-M., Gardner, M.-A. G., Parizeau, M., & Gagné, C. (2012). Deap: Evolutionary algorithms made easy. *J. Mach. Learn. Res.*, *13*(1), 2171–2175.

Gagné, C., & Parizeau, M. (2006). Genericity in evolutionary computation software tools: principles and case-study. *Int. J. Artif. Intell. Tools*, *15*(2), 173–194. doi: 10.1142/S021821300600262X

Hadka, D. (2019, December). *MOEA Framework: A Free and Open Source Java Framework for Multiobjective Optimization.* Version 2.13. http://www.moeaframework.org (Accessed May 22th, 2020).

Hadka, D. (2020, April). *Platypus: A Free and Open Source Python Library for Multiobjective Optimization.* Version 1.0.4. https://github.com/Project-Platypus/Platypus (Accessed May 22th, 2020).

Jones, D., Mirrazavi, S., & Tamiz, M. (2002). Multi-objective meta-heuristics: An overview of the current state-of-the-art. *Eur. J. Oper. Res.*, *137*(1), 1–9. doi: 10.1016/S0377-2217(01)00123-0

Kronfeld, M., Planatscher, H., & Zell, A. (2010). The EvA2 Optimization Framework. In *4th int. learning and intelligent optimization conference (lion)* (pp. 247–250). doi: 10.1007/978-3-642-13800-3_27

Li, B., Li, J., Tang, K., & Yao, X. (2015, September). Many-Objective Evolutionary Algorithms: A Survey. *ACM Comput. Surv.*, *48*(1), 13:1–35. doi: 10.1145/2792984

Li, M., & Yao, X. (2019). Quality Evaluation of Solution Sets in Multiobjective Optimisation: A Survey. *ACM Comput. Surv.*, *52*(2). doi: 10.1145/3300148

Liefooghe, A., Jourdan, L., & Talbi, E.-G. (2011). A software framework based on a conceptual unified model for evolutionary multiobjective optimization: ParadisEO-MOEO. *Eur. J. Oper. Res.*, *209*(2), 104–112. doi: 10.1016/j.ejor.2010.07.023

Lukasiewycz, M., Glaß, M., Reimann, F., & Teich, J. (2011). Opt4J: A Modular Framework for Meta-heuristic Optimization. In *Proc. 13th ann. conf. on genetic and evolutionary computation (gecco'11)* (pp. 1723–1730). doi: 10.1145/2001576.2001808

Luke, S. (2017). ECJ Then and Now. In *Proc. companion publication of the 2017 annual conference on genetic and evolutionary computation* (pp. 1223–1230). doi: 10.1145/3067695.3082467

Maxim, B. R., & Kessentini, M. (2016). Chapter 2 - an introduction to modern software quality assurance. In I. Mistrik, R. Soley, N. Ali, J. Grundy, & B. Tekinerdogan (Eds.), *Software quality assurance* (pp. 19–46). Boston: Morgan Kaufmann. doi: 10.1016/B978-0-12-802301-3.00002-8

Meng, Z., Liu, X., Yang, G., Cai, L., & Liu, Z. (2010). A comprehensive evaluation methodology for domain specific software benchmarking. In *2010 ieee int. conf. on progress in informatics and computing (pic)* (Vol. 2, pp. 1047–1051). doi: 10.1109/PIC.2010.5688006

Nebro, A. J., Durillo, J. J., & Vergne, M. (2015). Redesigning the jMetal Multi-Objective Optimization Framework. In *Proc. companion publication of the 2015 annual conference on genetic and evolutionary computation* (pp. 1093–1100). doi: 10.1145/2739482.2768462

Parejo, J. A., Ruiz-Cortés, A., Lozano, S., & Fernández, P. (2012). Metaheuristic optimization frameworks: a survey and benchmarking. *Soft Comput.*, *16*(3), 527–561. doi: 10.1007/s00500-011-0754-8

Ramírez, A., Romero, J. R., García-Martínez, C., & Ventura, S. (2019). JCLEC-MO: A Java suite for solving many-objective optimization engineering problems. *Eng. Appl. Artif. Intell.*, *81*, 14–28. doi: 10.1016/j.engappai.2019.02.003

Scott, E. O., & Luke, S. (2019). ECJ at 20: Toward a General Metaheuristics Toolkit. In *Proc. genetic and evolutionary computation conference (gecco companion)* (p. 1391—1398). doi: 10.1145/3319619.3326865

Sierra, M. R., & Coello Coello, C. A. (2005). Improving PSO-Based Multi-objective Optimization Using Crowding, Mutation and ϵ-Dominance. In *Proc. evolutionary multi-criterion optimization* (Vol. 3410, pp. 505–519). doi: 10.1007/978-3-540-31880-4_35

Silva, M. A. L., de Souza, S. R., Souza, M. J. F., & de França Filho, M. F. (2018). Hybrid metaheuristics and multi-agent systems for solving optimization problems: A review of frameworks and a comparative analysis. *Appl. Soft Comput.*, *71*, 433–459. doi: 10.1016/j.asoc.2018.06.050

Stewart, T., Bandte, O., Braun, H., Chakraborti, N., Ehrgott, M., Göbelt, M., … Di Stefano, D. (2008). Real-World Applications of Multiobjective Optimization. In *Multiobjective optimization* (Vol. 5252, pp. 285–327). Springer Berlin Heidelberg. doi: 10.1007/978-3-540-88908-3_11

Talbi, E.-G., & Hasle, G. (2013). Metaheuristics on GPUs. *Journal of Parallel and Distributed Computing*, *73*(1), 1–3. doi: 10.1016/j.jpdc.2012.09.014

Tsai, C.-W., Chiang, M.-C., Ksentini, A., & Chen, M. (2016). Metaheuristic Algorithms for Healthcare: Open Issues and Challenges. *Computers & Electrical Engineering*, *53*, 421–434. doi: 10.1016/j.compeleceng.2016.03.005

Ventura, S., Romero, C., Zafra, A., Delgado, J. A., & Hervás, C. (2008). JCLEC: a Java framework for evolutionary computation. *Soft Comput.*, *12*(4), 381–392. doi: 10.1007/s00500-007-0172-0

Voß, S., & Woodruff, D. (2002). *Optimization Software Class Libraries* (1st ed., Vol. 18). Springer US. doi: 10.1007/b101931

Wagner, S., Kronberger, G., Beham, A., Kommenda, M., Scheibenpflug, A., Pitzer, E., … Affenzeller, M. (2014). Architecture and Design of the HeuristicLab Optimization Environment. In *Advanced methods and applications in computational intelligence* (Vol. 6, pp. 197–261). Springer. doi: 10.1007/978-3-319-01436-4_10

Zavala, G. R., Nebro, A. J., Luna, F., & Coello Coello, C. A. (2014). A survey of multi-objective metaheuristics applied to structural optimization. *Struct. Multidiscipl. Optim.*, *49*(4), 537–558. doi: 10.1007/s00158-013-0996-4

Zhang, Q., & Li, H. (2007). MOEA/D: A Multiobjective Evolutionary Algorithm Based on Decomposition. *IEEE Trans. Evol. Comput.*, *11*(6), 712–731. doi: 10.1109/TEVC.2007.892759

Zhou, A., Qu, B.-Y., Li, H., Zhao, S.-Z., Suganthan, P. N., & Zhang, Q. (2011). Multiobjective evolutionary algorithms: A survey of the state of the art. *Swarm Evol. Comput.*, *1*(1), 32–49. doi: 10.1016/j.swevo.2011.03.001

Zitzler, E., Deb, K., & Thiele, L. (2000). Comparison of Multiobjective Evolutionary Algorithms: Empirical Results. *Evol. Comput.*, *8*(2), 173–195. doi: 10.1162/106365600568202

Zitzler, E., & Künzli, S. (2004). Indicator-Based Selection in Multiobjective Search. In *Proc. parallel problem solving from nature (ppsn viii)* (Vol. 3242, pp. 832–842). doi: 10.1007/978-3-540-30217-9_84

Zitzler, E., Laumanns, M., & Bleuler, S. (2004). Metaheuristics for Multiobjective Optimisation. In *Metaheuristics for multiobjective optimisation* (pp. 3–37). Berlin, Heidelberg: Springer Berlin Heidelberg. doi: 10.1007/978-3-642-17144-4_1

Zitzler, E., Laumanns, M., & Thiele, L. (2001). SPEA2: Improving the Strength Pareto Evolutionary Algorithm. In *Proc. conf. on evolutionary methods for design, optimisation and control with applications to industrial problems* (p. 95-100).

Zitzler, E., & Thiele, L. (1999). Multiobjective evolutionary algorithms: a comparative case study and the strength Pareto approach. *IEEE Trans. on Evol. Comput.*, *3*(4), 257–271. doi: 10.1109/4235.797969

How to cite this article: A. Ramírez, R. Barbudo, and J.R. Romero (2023), An experimental comparison of metaheuristic frameworks for multi-objective optimization, *Expert Systems*, *Volume 40*, *Issue 4*, *May 2023*, *e12672*. https://doi.org/10.1111/exsy.12672