

## Formula Optimizer: fast way to formulate and solve multi-objective combinatorial optimization problems

Thiago Gomes Nepomuceno, Tiago Carneiro Pessoa<sup>1</sup>, Thalyson Gomes Nepomuceno<sup>2</sup>

<sup>1</sup>ParGO Research Group (Paralellism, Graphs and Optimization)

Mestrado e Doutorado em Ciência da Computação, Universidade Federal do Ceará, Brazil

<sup>2</sup>Mestrado Acadêmico em Ciência da Computação

Universidade Estadual do Ceará, Brazil

thi.nepo@gmail.com, carneiro@lia.ufc.br, thalyson.uece@gmail.com

### ABSTRACT

This work presents the Formula Optimizer, a new software designed to formulate and solve multi-objective combinatorial optimization problems with no programming expertise and with low performance loss compared to the state-of-art solvers. In this paper, we describe how this software was designed, its features and examples of usage. We conducted a set of experiments comparing our software with a state-of-art framework to solve the bi-objective Travelling Salesman Problem. Results show that Formula is easier to use, faster to formulate a problem and that Formula has no big performance loss while solving the problem inside the software itself.

**KEYWORDS.** Optimization Software. Multi-objective Optimization. Metaheuristics.

**Optimization, Artificial Intelligence.**

### 1. Introduction

An Optimization Problem is defined as the task of finding one or more solutions that minimize (or maximize) one or more objectives that satisfy a set of constraints. When we solve problems with one objective, usually the optimization method results only in one solution. However, when we solve multi-objective problems rarely we have only one solution, because multi-objective problems are a set of conflicting objectives and all objectives need be optimized together and at the same time. In this way, solving a multi-objective problem is to find a set of solutions with different trade-offs, called Pareto Optimal Solutions, not dominated solutions or Pareto front.

Many frameworks for solving multi-objective problems by using meta-heuristics have appeared in the past years [Durillo e Nebro, 2011; Lukasiwycz et al., 2011; Streichert e Ulmer, 2005; Schoenauer et al., 2002]. These frameworks usually bring a set of state-of-the-art optimizers, a wide set of benchmark problems as well as a set of well-known quality indicators. There are many different ways to take advantage of these frameworks, such as, understanding the source code from a specific algorithm before implementing a new one, modifying the code according to some specific need, formulate a new problem and use the methods already implemented in the framework to solve it.

Researches interested in the formulation of the problem will, in general, use the framework only to help them to solve their problems. Basically they will repeat the following process: formulate a problem, implement a new *Problem* class that represents their problems into the framework, choose a method to solve the problem and evaluate the results using metrics and charts.

The process of implementing the *Problem* class may be challenging for researches with low programming skills, because they need to read and understand the documentation of the chosen framework. This process, on the other hand, may be too much repetitive to more advanced users. In order to help solving this problem, we present a new software called Formula Optimizer.

Formula is a software that helps researchers to focus on their formulations without the need of programming skills. In this paper, we present many advantages of using it by showing the

process to solve the bi-objective Travelling Salesman Problem. Then, we compare the Formula's performance with jMetal [Durillo e Nebro, 2011] framework, one of the most complete framework for solving multi-objective combinatorial optimization problems. According to the results, Formula is easier to use, faster to formulate a new problem and Formula has no big performance loss while solving the problem inside the software itself.

This work is organized as follows: Section 2 presents some related works and preliminary concepts necessary to a good understanding of this paper. Section 3 presents the Formula software and, in Section 4, we present a comparison of formulating a multi-objective problem with Formula *versus* formulating with jMetal and in Section 5 a computational evaluation of the proposed software is performed. Section 6 presents a discussion about the results obtained. Finally, we present in Section 7 the final considerations and we discuss about some features that are expected in the future.

## 2. Related Works

Currently, there are many frameworks for solving multi-objective combinatorial optimization problems available on the literature. Essentially these frameworks are a set of implementations of multi-objectives evolutionary algorithms, mutations, crossovers, selections, set of example problems and a class to perform analysis of results. Some state-of-art frameworks are discussed further.

Opt4J [Lukasiewicz et al., 2011] is a Java-based framework that has the implementation of evolutionary algorithms, differential evolution, particle swarm optimization and simulated annealing. This framework also includes a set of benchmark problems, such as ZDT, DTLZ, WFG and the knapsack problem. Evolving Objects [Schoenauer et al., 2002] is a C++ evolutionary computation library that allows the users to write stochastic optimization algorithms faster. It has a implementation of solution representation, algorithm paradigms, selection, crossover and mutation.

Eva [Streichert e Ulmer, 2005] is also a Java-based framework. This framework has a Graphical User Interface (GUI) that allows the user to interact with the framework functionalities. The jMetal [Durillo e Nebro, 2011] is an object-oriented Java-based framework for multi-objective optimization with meta-heuristics. It is one of the state-of-the-art frameworks to solve multi-objective combinatorial optimization problems by using meta-heuristics and one of the most complete frameworks. It has a huge set of algorithms, benchmark problems and operators (selection, crossover and mutation).

All the examples cited above allow the users to formulate and solve multiobjective combinatorial optimization problems fast, compared with the work and time necessary to code an algorithm, solution representation, crossover, selection, mutation and the problem itself from scratch. However all these frameworks requires programming effort, thus, programming expertise. Based on it, we propose a new way to formulate and solve multi-objective combinatorial optimization problems, which needs no programming effort and, if compared to the state-of-art solvers, has low performance loss.

## 3. Formula Optimizer

Formula Optimizer is a software that has the mission of allowing researchers to formulate and solve multi-objective problems without programming expertise and with minor performance loss. The software is divided into seven modules as showed in Figure 1.

The gray modules contain features that are usually present in a common framework:

- Import Problem: import a Java file that contains a problem implemented using another framework (like jMetal);
- Run Algorithm: run a set of parameters of the algorithm (a configuration);
- Run Experiments: run a set of configurations, e.g., to make a set of experiments to a scientific paper;

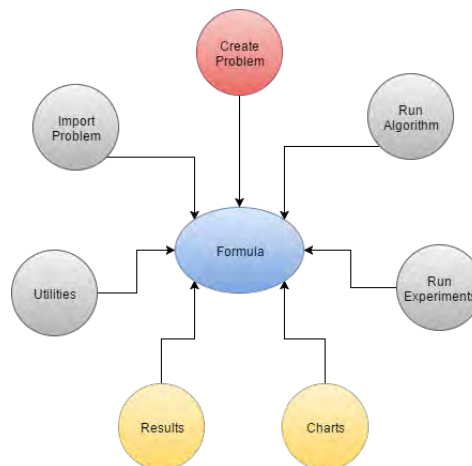


Figure 1: Modules of the Formula Optimizer

- **Utilities:** contains utilities tools. For example, a tool that given a set of files that contains different Pareto fronts, returns a Pareto front of the solutions present in all files.

The yellow modules are designed to create charts and tables of results that are usually present in many Genetic Algorithm (GA) works, like population charts, metrics charts and tables comparing results from different algorithms.

The core module is the *Create Problem*. This module allows the researcher to formulate a problem and solve it without any programming expertise and without leaving the software. As showed in Figure 2, in order to create a new problem using the GUI, the user should follow the following steps: create all need variables, choose how should be the mathematical formula of his formulation, choose the solution vector representation (Integer, Real, Binary, etc.) and choose a lower bound and upper bound value of each variable that the solution vector can assume (to Integer and Real solutions).

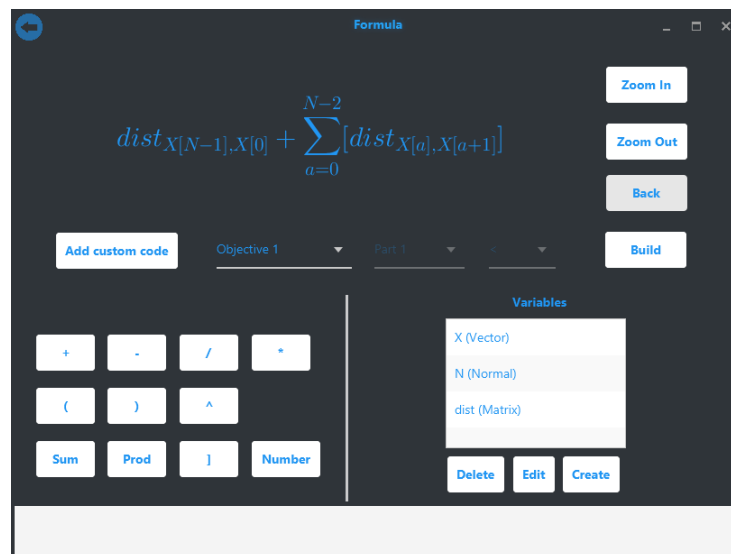


Figure 2: GUI of the *Create Problem* module.

After creating the formulation, the user should click on the *Build* button and Formula automatically creates all needed files and the problem is ready to be solved. This process also

creates some Java files, each of this files contains a Java class that is completely compatible with the correspondent optimization framework (until the release of this paper, only the jMetal framework is supported). So, at this point, the user may continue to use the features of Formula or continue his research on his preferred framework. This feature is important not only to give freedom of choice to the user, but allows the user to integrate the solutions resulted from the algorithm into their own software easily, sometimes just solving the problem is not the final goal.

Some formulations cannot (or are not easy to) be formulated with only basic mathematical symbols. To solve this problem we added a *Add custom code* feature where the user can add Java code to be executed right after his mathematical formulation to each objective or constraint, add complex Java types (like Lists and Map, for example) or even add any codes directly on the constructor of his problem. This way, any problem that use one of the solutions representations present on the Formula can be formulated and solved into the software. Until the release of this paper, the Formula supports: Integer, Double, Binary and Permutation solutions. This means that each variable of the solution vector can be of the integer type, real type, binary type or the entire solution vector will be a permutation that contains the numbers from 0 to  $N - 1$ , where  $N$  is the size of the solution vector.

After formulating and building the problem, the user may want to solve it using one of the optimization algorithms. Formula allows the user to choose among different meta-heuristics like the NSGA-II [Deb et al., 2000], MoCell [Nebro et al., 2009], SPEA2 [Zitzler et al., 2001], MOPSO [Coello e Lechuga, 2002] and many more will be added in the future.

To solve a problem, the user needs to define a configuration, that is the object that holds all parameters needed to execute the meta-heuristic. Each configurations will have a name, the maximum number of evaluations (stop criteria of the algorithm), population size, crossover ratio, mutation ratio, algorithm to be executed, problem to be solved, the instance files, the method to generate the initial population, selection operator, crossover operator, mutation operator, the pattern of the output file and how many times this configuration should be executed. The GUI that create a new configuration is showed in the Figure 3.

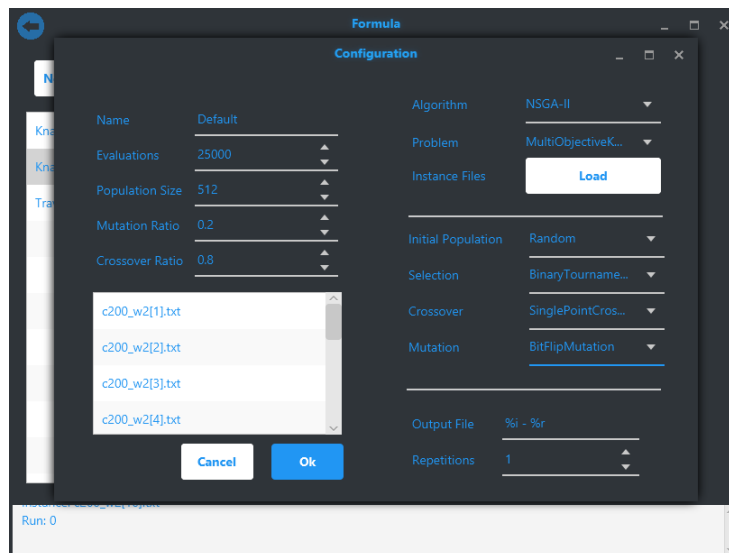


Figure 3: Create a new Configuration

We will describe how solve the problem using the *Run Experiment* module (the *Run Algorithm* module is a simpler version of it). The GUI of the *Run Experiments* module is showed in the Figure 4, in this GUI the user can create different configurations and, after that, choose which configurations will be executed in this experiment. When the user clicks on *Run* button, Formula

Optimizer will execute all the configurations sequentially and the progress bars will give a feedback to the user that can follow in which point of the experiments the execution already is. After Formula finish the execution to each instance, the results will already be available on the project folder, that holds all files needed to each project on the Formula Optimizer, like the problems, the solutions and general configuration file.

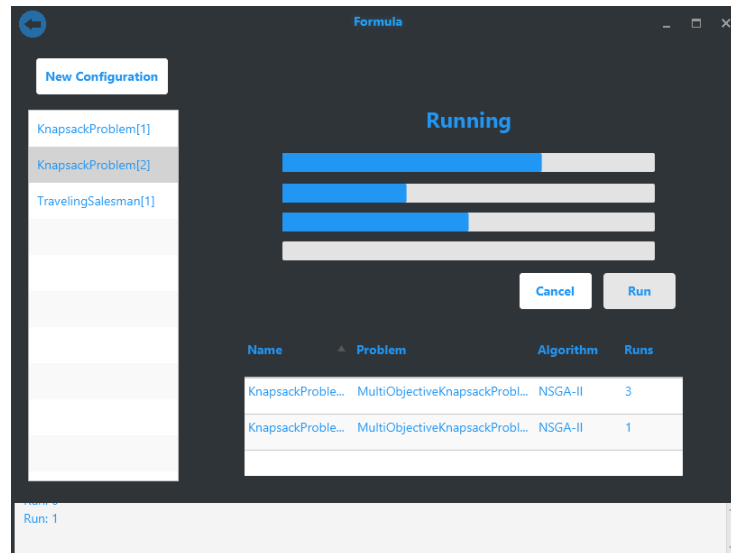


Figure 4: GUI of the Run Experiments module

After formulating and solving the problem, the researcher evaluates the results using charts and quality metrics, which can be done using the *Charts* and *Results* module. In the Figure 5 the *Charts* GUI can be seen. For example, to generate a population chart the user need to provide the title, label of the  $x$  axis, label of the  $y$  axis, the populations files and which objective will be in each axis (case the problem have more than two objectives).

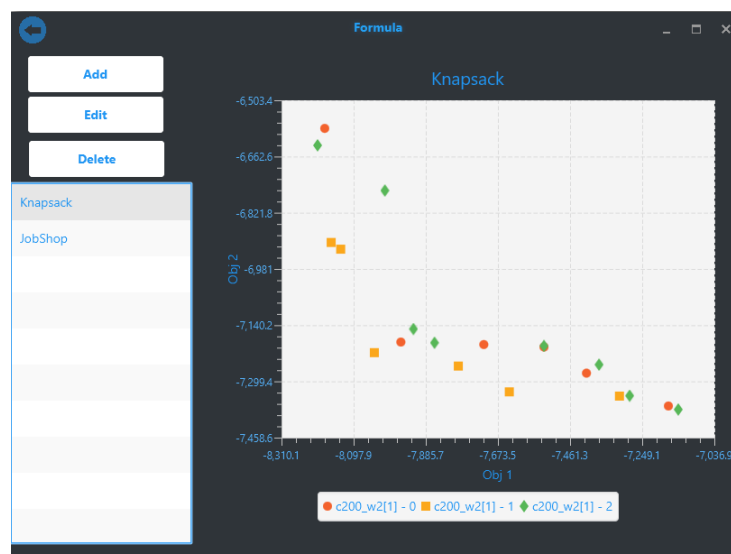


Figure 5: GUI of the Charts module

Thus, Formula provides to the user a set of tools that allows to formulate a problem, carry a complete experimentation with different algorithms and parameters and evaluate the results using charts and tables of result.

### 3.1. Aspects of implementation

The main part of Formula implementation is the *Create Problem* module, that allow the user to formulate a problem using only a GUI. The most important part is translate the formulation to a Java code, then use this Java code to solve the problem using the Formula or a external framework.

The translation is done using a template that will be filled with some informations. The basic informations are: solution type, upper and lower bound and size of the chromosome. After it, Formula will create a code to each objective and constraint. A *Builder* class will receive the formulation of the objective or constraint and create a code to each one. Each variable of the formulation will be represented in the code and each sum or product will have a associated variable storing the resulting value. The result of this process is a Java class that represent the formulation of the problem and can be understood by any programmer.

This code will be used by the Formula in the *Run Algorithm* and *Run Experiments* modules when the user is using a method to solve the created problem. A second option is export it as a Java file and use it in a external framework.

So, in a high level, the implementation consist in fill basic information about the problem and translate the information about the formulation of each objective and constraint, creating a correspondent code to each one, after this process we will have a Java class that represents the desired problem.

## 4. Evaluation

In this Section, we present the evaluation of the proposed software, in terms of problem formulation and computational performance. This Section is organized as follows: first, we describe the evaluation methodology, then, we compare the effort evolved into formulating a problem by using Formula Optimizer vs. jMetal. The following section presents the computational evaluation, in order to show that the user can solve problems inside Formula, without considerable performance loss. Finally, we present the results and an analysis of the results achieved.

### 4.1. Evaluation Methodology

Formula Software was created based on the premise that it is possible to formulate multi-objective problems easily than the state-of-art solvers, with no programming expertise, and solve them with low performance loss. To show that Formula archive its goals, we will perform two evaluations: we will show that it is easier to formulate multi-objective problems by using Formula, then, we will compare the performance of Formula vs. jMetal.

On the first experiments, we will show the necessary steps to formulate the multi-objective Travelling Salesman Problem (TSP) [Shi e Li, 2009], the problem of finding a shortest Hamiltonian cycle though a given number of cities in such a way that each city is visited only once, is one of the most disseminated and studied combinatorial optimization problem, having plenty of real-world applications [Cook, 2012], justifying our choice. Its multi-objective version will be discussed further.

The second experiment will be a computational evaluation comparing the performance between Formula vs. jMetal. This experiment will only evaluate the Runtime of the algorithms, because both approaches use the same algorithms to solve the same instances. So, metrics like Hypervolume [Zitzler, 1999] and Spread [Azarm e Wu, 2001] do not make sense. We will perform two different versions of tests, the first one is without the *Algorithm Progress Bar*, which means that the user will can't see if the execution is close to end or not, and in the second version we allow the use of the *Algorithm Progress Bar*.

The instances used in this study comes from the instance generator proposed by Cirasella et al. [2001], which creates instances of the Asymmetric Travelling Salesmen Problem using properties found in real-world situations. Two classes of instances, defining a representative set of properties, were selected: *crane*, that represents stacker crane operations, and *disk*, modeling the movements of the reading head of a hard disk. We created instances with 100, 500 and 1000 cities using each generator. In order to generate the multi-objective instance, one matrix cost  $M_{N \times N}$  was generated per objective. Each cost  $c_{ij}$  represents the cost of going from the city  $i$  to the city  $j$ .



#### 4.1.1. Testing environment

The testing environment has the following characteristics:

1. Operating system: Windows 10, 64 bits;
2. Processor: Intel Core i5-3210M @2.50GHz with two cores and 8 GB RAM;
3. Java(TM) SDK (build 1.8.0\_65).

#### 4.2. Formulating the Multi-objective Travelling Salesman Problem

In the Multi-objective Travelling Salesman [Shi e Li, 2009] are given a set of  $n$  cities ( $i = 1, \dots, n$ ) where for each pair of cities  $(i, j)$ , there are associated costs  $M_{ij}^1$  and  $M_{ij}^2$ , representing the two kinds of costs involved when moving from city  $i$  to city  $j$ . A solution for the Multi-objective TSP is a permutation  $x_1, x_2, \dots, x_n$  of the cities that minimizes the two cost functions. Formally, we have the following objective functions:

$$\min f^1(x) = \left( \sum_{i=1}^{n-1} M_{x_i x_{i+1}}^1 \right) + M_{x_n x_1}^1, \quad \min f^2(x) = \left( \sum_{i=1}^{n-1} M_{x_i x_{i+1}}^2 \right) + M_{x_n x_1}^2. \quad (1)$$

##### 4.2.1. Using the jMetal for formulating the multi-objective TSP

In order to formulate the multi-objective TSP by using jMetal, the first step is to create a new *Problem* class that inherits from the core *Problem* class of the jMetal. After that, the programmer will create all variables that will be necessary for his problem inside this class. For the multi-objective TSP, the programmer will need at least two different distance matrix, one to each objective. The next step is to create a constructor and attribute a correct value to all variables present on the core jMetal *Problem* class. To be aware of each variable, the programmer can use the documentation or see the examples present on the framework itself.

The programmer must create a way to receive a instance file and read the instance, attributing the correct value to each problem variable. On the multi-objective TSP, it is necessary to read the number of cities and the two distance matrices. The two last steps are override the *evaluate* and the *evaluateConstraint* functions, that calculate the value of the solution fitness and calculate how many (and how much) constraints the solution violated respectively. So, in the *evaluate* function, the programmer needs to code the formulation of the problem and set the value to each objective, and in the *evaluateConstraint* function, the programmer wants to know if the solution breaks any problem constraint and, if it does, how many constraint were violated and how much it was violated. For the multi-objective TSP, there are no constraints, because the *Permutation* solution type already deal with it.

The next step is choosing the algorithm to solve the problem, where the programmer needs to create a algorithm object, create a problem object, pass to the algorithm object, choose all parameters (maximum evaluation, crossover, mutation, selection, etc.), execute the algorithm and store the result population.

##### 4.2.2. Using the Formula Software

The first step to formulate the multi-objective TSP by using Formula is clicking on the *Create problem* button, after that, the user will inform the name of the problem and the quantity of objectives and constraints, in this case, as we are formulating the TSP, we have two objectives and zero constraints. In this point, we have a problem, but we have no data about the problem, so, we will create the both objectives of the multi-objective TSP formulation.

To create the problem formulation, the user must create each variable that is necessary and use the GUI to inform the proper formulation. This process is similar to the use a common

calculator software. After informing the objectives, the user should inform the type of variable and the lower value and upper value to each variable in the solution vector.

Formula already creates two variables that are common to all problems:  $X$  and  $N$ . The  $X$  variable always represents the solution vector and the  $N$  variable always represents the size of the solution vector.

The last step to formulate the multi-objective TSP by using Formula, is informing how the instance file will present the values. To do that, the user will inform to Formula the order in which the variables will be presented in the instance file. After this short process, we have a problem ready to be solved using the Formula Software or exported to a preferred framework. The GUI resulted from this process can be seen in Figure 2.

## 5. Performance Evaluation

The first set of experiments is showed in Table 1. To each algorithm we use different quantity of evaluations (stop criteria) and different size of population to solve a *crane* instance with 1000 cities. The name in the Configuration column of the Table 1 correspond to *algorithm\_maxEvaluations\_sizeOfPopulation*. In the second set of experiments, showed in the Table 2, we used the configuration that use *maxEvaluations* = 500,000 and *sizeOfPopulation* = 128 to solve multiples instances and the name in the Configuration column of the Table 2 correspond to *algorithm\_generator\_numberOfCities*.

The statistical evaluation of the results used the two-sample t-test with a significance level equal to 5%. The null hypothesis for both approaches is that the Runtime distributions are independent random samples from normal distributions with equal means and equal but unknown variances. The alternative hypothesis is that the means are not equal.

Tables 1 and 2 presents the results for the Runtime metric to both algorithms, NSGA-II and MoCell, in both approaches, jMetal and Formula. Each table entry shows three values: the mean, the standard deviation and the p-value obtained with the t-test. For all tests the reference sample is the one achieved by jMetal. The t-test of each column always matches with the jMetal algorithm result. Each table cell indicates, using a symbol, if the hypothesis has been rejected (X) or not (✓).

Runtime			
Configuration	jMetal	Formula	Formula with progress bar
NSGA-II.100000.128	5793.93333 (2566.81902)	6270.73333 (915.02342) X[0.00001]	7082.43333 (2448.45489) X[0.00000]
NSGA-II.100000.256	7287.10000 (1508.10102)	7544.20000 (4149.22195) ✓[0.09432]	7781.76667 (2339.58829) X[0.00000]
NSGA-II.500000.128	28880.20000 (3320.36787)	28801.60000 (8587.17120) ✓[0.80256]	39970.63333 (13057.71944) X[0.00000]
NSGA-II.500000.256	32523.23333 (1894.12496)	33202.23333 (7171.56854) X[0.01085]	42860.66667 (7650.52970) X[0.00000]
MoCell.100000.128	6090.36667 (936.63812)	6738.80000 (2976.40266) X[0.00000]	8416.16667 (2570.23621) X[0.00000]
MoCell.100000.256	6076.73333 (936.14415)	6757.40000 (1358.63579) X[0.00000]	8367.23333 (2539.16037) X[0.00000]
MoCell.500000.128	29439.26667 (1927.50353)	32062.10000 (5352.29509) X[0.00000]	42793.53333 (12738.22364) X[0.00000]
MoCell.500000.256	29935.20000 (2799.43294)	31935.76667 (8885.78614) X[0.00000]	40680.43333 (5455.03981) X[0.00000]

Table 1: Multiples configurations experiment

Runtime			
Configuration	jMetal	Formula	Formula with progress bar
NSGA-II.crane.100	4912.23333 (1152.95246)	5495.56667 (2975.78114) X[0.00000]	5590.33333 (972.54134) X[0.00000]
NSGA-II.crane.500	12951.06667 (2168.27255)	12898.03333 (3453.70076) ✓[0.70295]	19233.16667 (6534.64813) X[0.00000]
NSGA-II.crane.1000	28880.20000 (3320.36787)	28801.60000 (8587.17120) ✓[0.80256]	39970.63333 (13057.71944) X[0.00000]
NSGA-II.disk.100	5520.30000 (2657.23358)	4823.33333 (1541.84132) X[0.00000]	5150.16667 (1318.76085) X[0.00065]
NSGA-II.disk.500	13550.10000 (2075.32617)	12234.13333 (1945.55737) X[0.00000]	19158.80000 (6788.04160) X[0.00000]
NSGA-II.disk.1000	29410.56667 (5385.49323)	28891.60000 (7413.46540) ✓[0.10072]	39993.96667 (14119.54734) X[0.00000]
MoCell.crane.100	3385.66667 (1385.81769)	4931.50000 (2846.40993) X[0.00000]	4897.33333 (2025.10954) X[0.00000]
MoCell.crane.500	13374.83333 (1815.43553)	15119.80000 (2716.19233) X[0.00000]	19044.23333 (6925.51697) X[0.00000]
MoCell.crane.1000	29439.26667 (1927.50353)	32062.10000 (5352.29509) X[0.00000]	42793.53333 (12738.22364) X[0.00000]
MoCell.disk.100	3356.80000 (2355.13966)	4144.56667 (1289.40504) X[0.00000]	4700.53333 (1863.93333) X[0.00000]
MoCell.disk.500	13375.73333 (2101.51418)	14081.06667 (2257.69924) X[0.00000]	19205.06667 (6877.19259) X[0.00000]
MoCell.disk.1000	29712.60000 (3726.78993)	31555.26667 (6277.92863) X[0.00000]	42640.20000 (11650.28389) X[0.00000]

Table 2: Multiples instances experiment.



## 6. Discussion

This ability to export a formulation is specially important to commercial applications, because they will solve the problem as a part of a bigger system. But, if the goal is do a complete experimentation about a new formulation, all the problem can be solved inside the Formula using a Graphical User Interface (GUI).

To solve the problem using the jMetal we need to read and understand the documentation, learn how to use a algorithm to solve the problem and how use the *experiment* module to carry the experiments (or the programmer can create a own Java class that do it). Using Formula, the user will use the *Run Experiments* GUI and create the proper configuration, using the GUI showed in the Figure 3, to their experiments. After created all the configurations needed to the experiment, the user should add them to be executed and click the *Run* button.

The Formula method is good to create experiments with up ten or twenty configurations. Even with the *Clone configuration* feature, where the user can clone a configuration and modify only a small part of the parameters, the process to create fifty different configurations become tough and the traditional method, creating a Java class or a script, should work better in this case. We will probably add a way to create scripts inside the Formula to solve this problem in the future.

In his current state, Formula allows the user to formulate a new problem using the GUI, import a problem coded in a external framework, solve the problem by using a custom configuration, make a complete experimentation with many different types of configurations and evaluate the results using charts and tables. Thus, the whole process to solve a multi-objective problem (formulate, solve and evaluate results) can be done inside the Formula, but the user have the freedom to, after the formulation step, continue his experiment on his preferred framework. This can be useful for users that have programing expertise, but they want to formulate their problems fast and, then, tune the code generated by Formula Optimizer, according to their needs.

How it was expected, in our tests, the jMetal is the fastest one, because the GUI present in the Formula have a associated cost in memory and CPU usage. Formula with the progress bar have the worst performance because the progress bar is update in every generation of the Genetic Algorithm. But as we can see in the Tables 1 and 2 the increase in the time using the Formula Optimizer, when compared with the jMetal framework, is low and represent a 6% increase in average without use the progress bar and 11% allowing the progress bar.

## 7. Conclusions

This work has presented a new way to formulate multi-objective combinatorial optimization problems by using a software called Formula Optimizer. The proposed software accomplishes its mission, that is, to allow users with no programming expertises to formulate their problems quickly and solve multi-objective combinatorial optimization problems with low performance loss, inside formula itself, if compared to the state-of-art solvers. Formula Optimizer showed also to be useful for users with programming skills, that want to formulate their problems fast and, then, tune the code generated by Formula Optimizer, according to their needs.

The whole process to solve a multi-objective problem (formulate, solve and evaluate results) can be done inside the Formula, however, the overhead involved in using the Formula Optimizer, when compared with the jMetal framework, is low and represents a 6% increase in average, without use the progress bar, and 11% allowing the progress bar, which is not a big overhead, if compared to the benefits of formulating the multi-objective combinatorial optimization problem fast and using a Graphical User Interface.

A first future research direction is to make Formula available to be tested by the scientific community. Besides that, genetic algorithms and meta-heuristics have been successful parallelized along the years, allowing the researches to solve bigger instances or just to solve instances quickly. Thus, an other future research direction is, based on the formulation given by the user, to generate code for diverse heterogeneous and parallel architectures, such as Advanced Vector Extensions (AVX), present on mainstream multi-core computers, and for Graphics-process unities, by using

APIs such as CUDA and OpenCL, which will allow Formula Optimizer to run codes on different heterogeneous systems, from mobile phones to computational clusters.

## References

- Azarm, S. e Wu, J. (2001). Metrics for Quality Assessment of a Multiobjective Design Optimization Solution Set. *Journal of Mechanical Design*, 123(1):18–25.
- Cirasella, J., Johnson, D., McGeoch, L., e Zhang, W. (2001). The asymmetric traveling salesman problem: Algorithms, instance generators, and tests. *Algorithm Engineering and Experimentation*, p. 32–59.
- Coello, C. A. C. e Lechuga, M. S. (2002). Mopso: a proposal for multiple objective particle swarm optimization. In *Evolutionary Computation, 2002. CEC '02. Proceedings of the 2002 Congress on*, volume 2, p. 1051–1056.
- Cook, W. (2012). *In pursuit of the traveling salesman: mathematics at the limits of computation*. Princeton University Press.
- Deb, K., Agrawal, S., Pratab, S., e Meyarivan, T. (2000). A fast elitist non- dominated sorting genetic algorithm for multi-objective optimization: NSGA-II. *Parallel Problem Solving from Nature VI Conference*.
- Durillo, J. J. e Nebro, A. J. (2011). JMetal: A Java framework for multi-objective optimization. *Advances in Engineering Software*, 42(10):760–771. ISSN 09659978.
- Lukasiewicz, M., Głaß, M., Reimann, F., e Teich, J. (2011). Opt4J: A Modular Framework for Meta-heuristic Optimization. URL <http://dl.acm.org/citation.cfm?id=2001808> <http://doi.acm.org/10.1145/2001576.2001808>.
- Nebro, A. J., Durillo, J. J., Luna, F., Dorronsoro, B., e Alba, E. (2009). MOCeLL: A Cellular Genetic Algorithm for Multiobjective Optimization. *International Journal of Intelligent Systems*, 24(7):726–746. ISSN 0884-8173. URL <http://dx.doi.org/10.1002/int.v24:7> <http://doi.wiley.com/10.1002/int.20358>.
- Schoenauer, M. K., Merele, J. J., Romero, G., e M. (2002). Evolving Objects: A General Purpose Evolutionary Computation Library. *Artificial Evolution*, 2310:829—888. URL <http://www.lri.fr/~marc/EO/EO-EA01.ps.gz>.
- Shi, L. e Li, Z. (2009). An improved pareto genetic algorithm for multi-objective tsp. In Wang, H., Low, K. S., Wei, K., e Sun, J., editors, *ICNC (4)*, p. 585–588. IEEE Computer Society. ISBN 978-0-7695-3736-8. URL <http://dblp.uni-trier.de/db/conf/icnc/icnc2009-4.html#ShiL09>.
- Streichert, F. e Ulmer, H. (2005). JavaEvA A Java based framework for Evolutionary Algorithms - Manual and Documentation -. Technical report, University of Tübingen, Tübingen.
- Zitzler, E. (1999). *Evolutionary Algorithms for Multiobjective Optimization: Methods and Applications*. PhD thesis, Swiss Federal Institute of Technology (ETH).
- Zitzler, E., Laumanns, M., e Thiele, L. (2001). SPEA2: Improving the Strength Pareto Evolutionary Algorithm. p. 95–100.