

Efficient Metaheuristics in Higher-Level Programming Languages

Fellipe Souza Pessanha

Instituto de Computação, Universidade Federal Fluminense - UFF
PSR - Energy Consulting and Analytics
fellipe900@gmail.com, fellipesouza@psr-inc.com

Pedro Maciel Xavier

Elmore Family School of Electrical and Computer Engineering - Purdue University
pmaciex@purdue.edu

Augusto Magalhães Pinto de Mendonça

Instituto de Computação, Universidade Federal Fluminense - UFF
augusto01@gmail.com

Igor Machado Coelho

Instituto de Computação, Universidade Federal Fluminense – UFF
Programa de Pós-Graduação em Ciências Computacionais e Modelagem Matemática (CompMat),
Universidade do Estado do Rio de Janeiro – UERJ
imcoelho@ic.uff.br, igor.machado@ime.uerj.br

ABSTRACT

This paper studies the impact of modern programming languages in the design of metaheuristic algorithms. Well-known metaheuristics have been proposed and studied over the last decades, including population-based and evolutionary algorithms like Genetic and Memetic Algorithms, trajectory-based Simulated Annealing, Tabu Search, Variable Neighborhood Search, among others. Despite efforts from the community in proposing generic metaheuristic frameworks in low-level languages like C/C++, the growth of developers in higher-level programming languages — such as Julia and Python — demands existing efficient C/C++ tools to be adapted for the languages, so that new users can also benefit from them. This work proposes a Julia version of the OptFrame, a C/C++ combinatorial optimization framework for metaheuristics, comparing performance with other higher-level languages, like Python, considering well-known challenging combinatorial optimization problems. Results indicate that Julia is able to provide similar C/C++ performance, outpacing Python, while being easier to code and explore newer efficient metaheuristic techniques.

KEYWORDS. Julia. OptFrame. Metaheuristics.

Paper topics: MH – Metaheuristics

1. Introduction

Metaheuristics are fundamental techniques for dealing with challenging optimization problems, as general heuristic frameworks that allow searching for a global optimum while escaping lower quality local optima [Glover and Laguna, 2019]. Although such heuristic techniques do not guarantee optimality, they are usually able to provide lower runtimes than exact algorithms for several problems in industry, like networks, logistics, transportation, biomedical engineering, and scheduling [Gendreau and Potvin, 2019]. The design of efficient metaheuristics is still a challenge, due to emerging trends and novelties in the field like hybrid metaheuristics [Talbi, 2013], and also considering the technical challenges of efficient computational implementations in lower-level programming languages [Talbi, 2009].

In this sense, several authors have proposed computational frameworks for the design of metaheuristics, focusing on trajectory or population-based metaheuristics, single or multi-objective problems, or even integration with reinforcement learning strategies [Silva et al., 2019]. Although several frameworks exist for popular languages in optimization community, like C++¹, Julia² and Python³, few frameworks target multiple languages simultaneously, thus limiting the ability to adapt projects to specific language dependencies.

Coelho et al. [2010] introduced OptFrame, a multi-language and white-box object-oriented framework, initially implemented in C++ and then extended to Java⁴ and also Python⁵, designed to facilitate the development of efficient heuristic algorithms for combinatorial optimization problems. Over the following decade, this framework evolved significantly, as demonstrated by Coelho et al. [2020] in their 10-year anniversary study.

Building upon this established multi-language foundation of OptFrame, this work aims to extend these benefits to the growing community of developers working in higher-level programming languages, by exploring an efficient C++ dynamic library built on OptFrame that can be reused to other languages. By implementing an OptFrame package in Julia, a high-level and just-in-time compiled language, we seek to address the growing demand for efficient metaheuristic tools in modern programming environments while maintaining performance comparable to C++. We also perform computational experiments with the existing C++ and Python versions of OptFrame for two challenging optimization problems, which have not been done before in the literature, to the best of our knowledge.

This paper is organized in six sections. In Section 2, we review relevant metaheuristic frameworks across different programming languages, with particular attention to their design choices, capabilities, and performance characteristics. Section 3 discusses the challenges of efficiently managing memory in different platforms and programming languages, specially focusing on memory interactions of C++ and other languages. In Section 4, we provide some experiments, detailing the proposed algorithms and the classic optimization problems considered. Finally, Section 5 concludes the work and gives some future perspectives for improvements.

2. Brief Metaheuristic Frameworks Review

In this section, we review relevant metaheuristic frameworks across different programming languages, with particular attention to their design choices, capabilities, and performance characteristics. Several frameworks have emerged to address the challenge of implementing efficient metaheuristics while reducing development effort.

¹<https://isocpp.org>

²<https://julialang.org/>

³<https://www.python.org>

⁴<https://sourceforge.net/projects/joptframe>

⁵<https://pypi.org/project/optframe>

[Ramírez et al., 2023] conducted a comprehensive evaluation of metaheuristic frameworks, finding that specialized tools better capture recent trends in optimization while mature multi-purpose frameworks provide more general utilities. Their analysis revealed that performance differences become more pronounced under demanding computational scenarios.

2.1. C++ Frameworks

ParadiSEO, an influential C++ framework with a 22-year history [Dreo et al., 2021], employs four main design patterns (Functors, Strategy, Generic Type, and Factory) organized into four core modules. It distinguishes itself by combining modularity, performance, and automation, significantly outperforming Python implementations in benchmarks. The framework supports SMP and MPI, enabling linear scalability across multiple cores, making it valuable for both research and professional applications.

On Python, DEAP is one of the most famous optimization frameworks, focusing on evolutionary computation metaheuristics [Fortin et al., 2012]. On Julia, we could not find general metaheuristic frameworks, at the time of the writing, which was an incentive to create one, based on OptFrame.

OptFrame, introduced by Coelho et al. [2010], is a white-box multi-language framework designed to facilitate efficient heuristic algorithms for combinatorial optimization problems. Its architecture minimizes the gap between theoretical concepts and implementation through clear conceptual modeling. The framework's evolution was documented by Coelho et al. [2020], showing how modern C++ features like coroutines and closures improved flexibility while reducing computational overhead from 405%-1714% to just 8%-34% compared to baseline implementations. An important feature of OptFrame is its ability to deal with both trajectory metaheuristics, like Simulated Annealing [Kirkpatrick et al., 1983], Iterated Local Search (ILS) [Lourenço et al., 2003], Tabu Search [Glover, 1997], Variable Neighborhood Search (VNS) [Mladenović and Hansen, 1997], among others, and also population-based metaheuristics like Genetic Algorithms (GA) [Holland, 1984], Memetic Algorithms (MA) [Moscato and Cotta, 2003] and, more recently, Biased Genetic Random Key Genetic Algorithms (BRKGA) [Resende and Ribeiro, 2016a]. It also supports single and multiobjective optimization problems, through Pareto optimization [Lust and Teghem, 2010].

2.2. Concepts and Components on OptFrame

OptFrame allows the generation of metaheuristics (or even hybrid metaheuristics) as a composition of *Components*, and we briefly explore some basic ones. The *XSolution* concept represents a *solution space*, which is entirely defined by some data structure passed by the user. For example, considering the classic Traveling Salesman Problem, where one tries to find a route that minimizes the traveling distances among cities, on C/C++ an array of integers can define such permutation as a valid *XSolution*. On the other hand, the *XEvaluation* concept represents the *objective space*, typically some *number* that gives a value to a solution (at least for single-objective problems), and that can be programmed on C/C++ using core numeric types like `int` or `double`.

One can also supply components like *Constructive* methods for some general *XProblem*, as a function of type $Constructive : XProblem \rightarrow XSolution$. An evaluation function is able to map solutions to other objective space elements, by defining the function $evaluate : XSolution \rightarrow XEvaluation$. At the core of the framework is the concept of *neighborhood structures*, which are composed of *move* operations (or a space of moves called *XMove*), where each move is capable of generating new solutions through some user-defined method $apply : (XMove, XSolution) \rightarrow XSolution$. The neighborhood structures of type *NS* are able to randomly generate *moves* according to some given solution, by defining the function $randomMove : XSolution \rightarrow XMove$. The neighborhoods of type *NSSeq* are able to generate iterators that generate *all possible moves* for a given input solution. Finally, each move *m* has a cost $\bar{m}$ for a given solution *s*, defined as

$\bar{m}(s) = \text{evaluate}(\text{apply}(m, s)) - \text{evaluate}(s)$. By using move components, neighborhood structures, evaluators and constructive methods, one can automatically use dozens of popular metaheuristics already implemented in OptFrame. Some other components in OptFrame are metaheuristic-specific, like mutation and crossover operators for GA and MA, greedy-randomized constructors for GRASP [Resende and Ribeiro, 2016b], and many others not covered in this brief review of OptFrame.

2.3. Other High-Level Language Frameworks

In the Julia ecosystem, de Dios and Mezura-Montes [2022] introduced Metaheuristics.jl, providing a uniform API for optimization algorithms with both single and multi-objective capabilities. This package uniquely supports equality and inequality constraints alongside multi-objective problem solving, addressing a notable gap in the Julia ecosystem.

The performance trade-offs between high-level languages and C++ implementations were analyzed by Gmys et al. [2020], who found that Julia was approximately 2.5× slower than C/OpenMP in sequential tests but required 57% fewer lines of code. For parallel implementations with 64 threads, this gap widened to 6.5×, highlighting Julia's immature multi-threading support. Despite these limitations, Julia's productivity advantages remain significant for prototyping and sequential execution.

3. Challenges of Efficient Memory Management: Julia and Python

In this section, we briefly discuss how memory is managed in high-level languages and how it affects the creation of efficient metaheuristics in these languages. This review also provides context for our Julia implementation of OptFrame by highlighting Julia's unique position in the programming language spectrum.

3.1. Julia Language

As documented in Eschle et al. [2023], Julia performs remarkably well compared to C++, with performance ratios ranging from 0.73 to 1.67, while Python ranges from 1.12 to 107. In a real-world analysis using particle physics data, Julia outperformed C++ by 11% and was approximately 1000 times faster than pure Python. JuMP, described by Dunning et al. [2017], allows users to express optimization problems in a high-level, algebraic syntax while achieving performance competitive with commercial systems. Unlike other modeling languages that use operator overloading, JuMP leverages Julia's macro system to translate expressions directly into efficient data structures, avoiding computational overhead typical of high-level languages.

3.2. Cross Platform Memory Management

Managing memory in a cross-platform environment comes with its difficulties, which may be aggravated by languages that choose different strategies of managing memory.

OptFrame is written in C/C++, which handles memory by letting the user allocate and free memory to fit the program's needs in the Resource Acquisition Is Initialization (RAII) idiom [W3Computing, 2023]. The language gives low-performance impact tools to the user, such as smart pointers and mutexes, allowing the user to build extremely fast programs at the cost of some development overhead time.

On the other hand, higher-level languages often employ the Garbage Collection strategy to free the user of the burden of manually managing memory. In this system, the program keeps a table of used memory, and whenever a memory range is no longer reachable, it is automatically freed. This strategy implies a heavier performance impact, since the algorithms run evaluate which memory ranges are accessible have non-negligible cost.

Bad memory management can lead to dangling references, where a pointer's content is dropped before the lifetime of the pointer itself, leading to undefined behavior, since you can not

know what the pointer is holding. This issue demand a great deal of caution, especially in developing software that spans platforms with different memory management strategies [W3Computing, 2023].

3.3. Challenges of Garbage Collection

The OptFrame C++ interface does not have a garbage collector, so it works by sharing *smart pointers*⁶ bidirectionally between the languages that represent objects or callback functions. In those callbacks, for instance, one may create an object representing a neighborhood move operator, which may be freed from memory — be it collected by garbage collection, or optimized away by the compiler, or being dropped by going out of scope — while the pointer to that object may not have the same fate. A pointer to that object will contain a *dangling reference*, which will cause undefined behavior whenever we try to access it at a later point in time.

To prevent that from happening, an object must be kept alive in memory for an arbitrary amount of time, which may pose a challenge in garbage collected languages, since this objective is not usual and may lead to a highly feared problem in programming: memory leaks.

A memory leak is what happens whenever there is an unreachable object in memory, which is the purpose of garbage algorithms [Cohen, 1981]. It keeps track of which objects are still reachable by the language interpreter and disposes of the ones that are not. What is not taken in consideration in this context is that other entities, such as other programs, may try to access that memory, and thus that job falls into the user who is keeping that interface alive, who should know when to free that memory.

The challenges of cross-language memory management become critical when dealing with cyclic references across FFIs. Yamazaki et al. [2023] demonstrate that conventional garbage collectors fails to reclaim cyclic garbage spanning language boundaries. Their "Refgraph GC" algorithm coordinates collection between languages while modifying only one runtime, which informed our OptFrame-Julia implementation approach. We implemented a reference tracking system that maintains a compressed graph of shared objects, allowing Julia's collector to properly identify when OptFrame objects can be safely reclaimed without creating dangling references between C++'s RAII-based management and Julia's garbage-collected environment.

3.4. Garbage Collection Manipulation

Explicit management of garbage collection becomes essential when working with cross-language interfaces like OptFrame. Python provides a comprehensive API through its `gc` module that allows developers to disable automatic collection via `gc.disable()`, manually trigger collection with `gc.collect()`, and access unreachable objects through `gc.garbage` when debugging is enabled [Python Software Foundation, 2025]. This fine-grained control helps prevent dangling references by allowing precise coordination of when objects can be safely reclaimed, particularly important when shared pointers cross language boundaries. Julia's garbage collection system offers complementary capabilities through its non-moving, parallel, generational collector. As described by [JuliaLang, 2025], Julia implements a tiered allocation approach with different strategies for objects of different sizes, and provides runtime parameters like `--gcthreads` for tuning collection behavior.

In order to properly solve the integration problem of C++ with Julia, we define a global space of pointers named *arena* in Julia, such that any memory allocated by Julia is held in this arena, until C++ requests its deletion by executing some given callback function. This is necessary because Julia does not allow users to manually manage memory outside of the scope of its garbage collector, to prevent errors from the user. In Python, a similar strategy can be implemented without

⁶Automatically managed pointers in C++ are typically `std::unique_ptr` and `std::shared_ptr`, introduced on C++11 standard.

an explicit arena. Since objects in Python are reference counted, dangling objects created in Python can have their reference-count number manually incremented, until a callback function from C++ decrements this number and allows memory to be free'd. This system is, in principle, more efficient than relying in an arena like in Julia, but as long as the C++ code is able to handle most of the memory, the arena is not likely to generate a lot of burden to the system, typically not consuming lots of objects simultaneously during the execution of a metaheuristic.

4. Experiments: Python vs C++ vs Julia

For the computational experiments, we implemented a Julia version of the OptFrame C++ project, publicly available online⁷. We have used the latest OptFrame v6 in C++ as the basis for building our Julia version, and also compared to OptFrame Python v6, both released in 2025. As a target problem for the multi-language framework, we have chosen two classic optimization problem with well-known metaheuristic strategies in the literature: the classic Traveling Salesman Problem (TSP) with Euclidean distances and the 0-1 Knapsack Problem (KP). We now describe the heuristic components of the proposed test strategies.

4.1. Neighborhood Structures and Constructive Methods for the TSP and KP

We have implemented two classic neighborhood structures for the TSP: the Swap and the 2-Opt. The Swap move consists in selecting two elements from the solution and swapping their positions. On the other hand, the 2-Opt consists in removing two arcs from the solution and reconnecting them, in a way that a new route is formed. For the KP, we created a bit flip move operator that removes an item, if it belongs to the knapsack, or adds it, if it does not belong to the knapsack. The capacity of the knapsack is always respected during the move operation, otherwise the move is not applied. Regarding constructive methods, we created a completely randomized initial solution generator for both problems.

4.2. Microbenchmarking for heuristic components

As the biggest differences would be in the small implementation details, we chose to use as benchmarking tool OptFrame's own microbenchmarking tool – Check Modules – as an important tool for benchmarking. Inspired by [Coelho et al., 2020], the module evaluates each individual module in OptFrame's engine, generating individual tests based on its function in the code. This is ideal for our test case because it makes it possible to measure time spent in the most critical parts of code: the interaction between different execution times.

4.3. Tested Heuristic Components in Check Modules

In OptFrame's architecture, each step in the metaheuristics algorithm is represented by a module, such as Constructive, Evaluate, and Apply (as described in Section 2.2). In measuring those different Modules, we can have a statistical idea of how different language implementations behave on a micro scale, while being certain the modules are sufficiently robust, and check how they'll behave in different scenarios. Table 1 presents the microbenchmark results between Python, Julia and C++, with average times for each benchmark metric and standard deviation. Since these values are very small, we have removed outliers by calculating interquartile ranges and removing those above and below 1.5, with a total of 1% elements removed, indicating stability in the metrics measured. To keep an uniform testing space, we use the same dynamic library API for building our C++ examples.

From Table 1, we can see that all language-specific components implemented for the 0-1 knapsack problem take longer in Python, and then Julia follows close to C++ performance. Considering constructive method, which is more language-specific resource-intensive process, we note a larger difference between the languages: Python is 22.5 times slower than C++, while Julia

⁷OptFrame Julia: <https://github.com/fellipessanha/optframe.jl/>

Table 1: Microbenchmark results from Python, Julia and C++ for the classic 0-1 Knapsack Problem

Benchmark	Python		Julia		C++	
	avg.(ms)	sd.	avg.(ms)	sd.	avg.(ms)	sd.
solution clone	0.0037	0.0012	0.0004	0.0005	0.0001	0.0001
constructive	0.0235	0.0074	0.0034	0.0007	0.0010	0.0001
evaluate	0.0017	0.0007	0.0001	0.0001	0.0001	0.0001
move apply	0.0031	0.0022	0.0003	0.0005	0.0001	0.0001
move cost	0.0081	0.0025	0.0005	0.0005	0.0003	0.0005

is 3.4 times slower than C++. For the evaluation function, Julia achieves the same performance as C++, which is remarkable. In the following section, the article digs deeper in the comparison between Julia and C++ using established strategies of solving the Traveling Salesman Problem as our baseline.

4.4. Parameter Tuning for Metaheuristic Optimization

We have tested two classic metaheuristics in this phase: the Simulated Annealing (SA) and the Iterated Local Search, using Variable Neighborhood Descent as local search (ILS-VND). Both algorithms are known to perform well on a broad range of combinatorial problems, so we test them on the classic Traveling Salesman Problem. At this point, we only consider C++ and Julia, since Python had much lower efficiency compared to these two languages in previous experiments.

Complex metaheuristic algorithms typically require careful parameter selection to achieve optimal performance across different problem instances. We conducted extensive parameter tuning experiments utilizing IRACE Version 4.2.0 [López-Ibáñez et al., 2016] on computing equipment consisting of an i7-13700 processor with 32GB RAM running Ubuntu 22. The tuning process focused on optimizing algorithm performance simultaneously across eight classical TSP instances of varying complexity: berlin52, kroD100, lin318 and TRP-1001.

SA parameter tuning was performed with a computational budget of 300 algorithm executions. Table 2 presents the explored parameter space and resulting optimal configuration. The parameters used were: Neighborhood, determining which move operators to employ (0=swap-only, 1=2-opt-only, 2=both); Alpha (α), controlling the cooling rate of the temperature; $IterT$, determining the number of iterations at each temperature level; and T_0 , setting the temperature value (early acceptance probabilities of worse solutions).

Table 2: Simulated Annealing parameter configurations tested using IRACE

Parameter	Tested Values					
Neighborhood	0	1	2	-	-	-
Alpha (α)	0.94	0.95	0.96	0.97	0.98	0.99
IterT	500	1000	2000	3000	4000	5000
T_0	25000	50000	75000	100000	125000	150000

Statistical analysis of final configurations identified: Neighborhood=2 (both swap and 2-opt operators), $\alpha = 0.99$, $IterT = 4000$, and $T_0 = 100000$ as optimal. This setup provided a good balance between exploration and exploitation phases of the search process. Values beyond $\alpha = 0.99$ were excluded over high computational cost and were unnecessary for effective framework comparison.

4.5. Iterated Local Search Configuration

ILS parameter tuning employed IRACE [López-Ibáñez et al., 2016] with an identical budget of 300 executions. Table 3 presents the explored parameter space and resulting optimal configuration. The parameters used were: Neighborhood, determining the local search approach (0=Best Improvement with Swap, 1=Best Improvement with 2-opt, 2=VND); *maxIter*, controlling the maximum number of iterations before increasing perturbation level; and *maxPertLevel*, determining the maximum perturbation intensity applied to escape local optima.

Table 3: Iterated Local Search parameter configurations tested using IRACE

Parameter	Tested Values									
Neighborhood	0	1	2	-	-	-	-	-	-	-
maxIter	500	1000	2000	3000	5000	10000	-	-	-	-
maxPertLevel	1	2	3	5	6	7	8	9	10	-

The analysis of the configuration identified: Neighborhood=2 (Best Improvement with VND), *maxIter*=5000, and *maxPertLevel*=2 as optimal.

4.6. Performance Comparison

We compare the tuned versions of SA and ILS-VND, for C++ and Julia. On Table 4, we compare a C++ version of SA with the ILS-VND. Note that we provide best and average values found for 10 executions of each method, and we also register the average time spent and the time to find the best solution (TTB). From this table, we observe that SA was able find better quality solutions than ILS-VND, for the given parameters. Advanced move recalculation techniques can also be used in OptFrame C++ for avoiding full recalculation of moves, thus highly accelerating move-intensive methods like ILS-VND, however these would require more efforts to replicate on Julia, so we stick to a more basic version of ILS-VND. We believe these can have influenced the result towards SA, since in recent publications in literature the ILS has shown better results than SA for routing problems on general. We highlight that the idea of this experiment was not to search for the most efficient methods to solve the TSP, since this would require much efforts as a quite classical problem, this only illustrates the capability of the framework in solving the problem with different metaheuristics.

Table 4: Results on 10 executions C++ for algorithms SA and ILS-VND on the TSP with N cities with timelimit of $N/5$ seconds

Instance	SA				ILS-VND			
	solution quality		time (secs)		solution quality		time (secs)	
	best	avg.	TTB	avg.	best	avg.	TTB	avg.
berlin52	8095	8283.67	3.19	9.66	9083	9370	8.52	10.40
krod100	25288	26385.3	1.59	9.06	33943	36986.7	15.42	20.00
lin318	72960	78186.3	17.67	34.19	149942	162118	0.01	63.86
TRP-1001	30980	31479.7	52.79	52.84	211227	216849	0.01	203.32

Finally, on Table 5, we present a comparison of the ILS-VND in both C++ and Julia languages, using the developed OptFrame Julia. We have chosen ILS-VND to do this comparison since it is a more complex method than Simulated Annealing, integrating several features of the framework, like neighborhood structures, iterators, perturbation and a VND local search method,

composed of two neighborhoods in a Best Improvement search strategy. Although C++ was able to find better quality solution than Julia for the smaller two instances, Julia was able to find better quality solution for the larger ones. This may indicate some slight differences in the implementation of each component of C++ and Julia metaheuristics, so as differences in the used random generation techniques, which were not standardized in both methods for this experiment. If it was standardized, both should have presented exactly the same result, and this is a promising line of investigation for future experiments with both C++ and Julia frameworks.

Table 5: Results for ILS-VND on C++ and Julia the TSP with N cities with timelimit of $N/5$ seconds

Instance	C++				Julia			
	solution quality		time (secs)		solution quality		time (secs)	
	best	avg.	TTB	avg.	best	avg.	TTB	avg.
berlin52	9083	9370	8.52	10.40	10170	10987.5	0.52	0.83
krod100	33943	36986.7	15.42	20.00	46727	52429.9	2.88	4.96
lin318	211227	216849	0.01	203.32	175247	184461	51.70	63.60
TRP-1001	211227	216849	0.01	203.32	81496	88006.50	199.36	200.20

5. Conclusions

In this work, we present a Julia version of the C++ framework OptFrame, also comparing to a Python version of OptFrame. In preliminary experiments, we used a classic Knapsack Problem to benchmark heuristic components on Python, Julia and C++, where C++ got the best results, followed by Julia, around 3 times slower. In a complete metaheuristic experiment on the Traveling Salesman Problem, Julia was able to compete with C++ for the Iterated Local Search method, indicating that, in some situations, the easy of programming of Julia can facilitate the development of efficient metaheuristics, typically done in C++. In this sense, our results show a very promising future for metaheuristic development in Julia, with it being able to beat even native C++ code in the largest instances, although random number generation should be standardized between both languages in future experiments of the kind, in order to provide a definitive comparison between both languages, in the scope of metaheuristics. All OptFrame C++, Python and Julia optimization packages used in this work are open-source and already published online. Programming in Julia also has the benefit of being fairly easier, with friendlier and easier to fix errors, and demanding less expertise in obscure language features compared to C++, that may be a promising path for researchers and companies that are already familiar with higher-level programming languages.

Acknowledgements

This work was carried out with the support of the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brazil (CAPES) – Financing Code 001, the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) for grants PQ 2021 315059/2021-4 and PQ 2024 312165/2025-0, and the Fundação Carlos Chagas Filho de Amparo à Pesquisa do Estado do Rio de Janeiro (FAPERJ) for grant JCNE 2021 E-26/201.451/2022.

References

- Coelho, I. M., Ribas, S., Perché, M. H. P., Munhoz, P. L. A., Souza, M. J. F., and Ochi, L. S. (2010). Optframe: a computational framework for combinatorial optimization problems. In *Anais do XLII Simpósio Brasileiro de Pesquisa Operacional*, p. 1887–1898, Bento Gonçalves, RS, Brasil. Sociedade Brasileira de Pesquisa Operacional, SOBRAPO.

- Coelho, I. M., Coelho, V. N., Zudio, A., Araujo, R. P., Haddad, M. N., Munhoz, P. L. A., Maia, B., Ochi, L. S., and Souza, M. J. F. (2020). Microbenchmark studies in optframe: a 10-year anniversary. In *Proceedings of the LII Simpósio Brasileiro de Pesquisa Operacional*, Campinas. SOBRAPO. URL <https://doi.org/10.59254/sbpo-2020-122744>.
- Cohen, J. (1981). Garbage collection of linked data structures. *ACM Computing Surveys (CSUR)*, 13(3):341–367.
- de Dios, J.-A. M. and Mezura-Montes, E. (2022). Metaheuristics: A julia package for single- and multi-objective optimization. *Journal of Open Source Software*, 7(78):4723. URL <https://doi.org/10.21105/joss.04723>.
- Dreo, J., Liefoghe, A., Verel, S., Schoenauer, M., Merelo Guervós, J. J., Quemy, A., Bouvier, B., and Gmys, J. (2021). Paradiseo: From a modular framework for evolutionary computation to the automated design of metaheuristics: 22 years of paradiseo. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO '21 Companion*, p. 1522–1530, New York, NY, USA. Association for Computing Machinery. ISBN 978-1-4503-8351-6.
- Dunning, I., Huchette, J., and Lubin, M. (2017). Jump: A modeling language for mathematical optimization. *SIAM Review*, 59(2):295–320. ISSN 1095-7200. URL <http://dx.doi.org/10.1137/15M1020575>.
- Eschle, J., Gál, T., Giordano, M., Gras, P., Hegner, B., Heinrich, L., Hernandez Acosta, U., Kluth, S., Ling, J., Mato, P., Mikhasenko, M., Moreno Briceño, A., Pivarski, J., Samaras-Tsakiris, K., Schulz, O., Stewart, G. A., Strube, J., and Vassilev, V. (2023). Potential of the julia programming language for high energy physics computing. *Computing and Software for Big Science*, 7(1):10. URL <https://doi.org/10.1007/s41781-023-00104-x>.
- Fortin, F.-A., De Rainville, F.-M., Gardner, M.-A., Parizeau, M., and Gagné, C. (2012). DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13:2171–2175.
- Gendreau, M. and Potvin, J.-Y., editors (2019). *Handbook of Metaheuristics*, volume 3 of *International Series in Operations Research & Management Science*. Springer. ISBN 978-3-319-91085-7. URL <https://doi.org/10.1007/978-3-319-91086-4>.
- Glover, F. (1997). Tabu search and adaptive memory programming—advances, applications and challenges. *Interfaces in computer science and operations research: Advances in metaheuristics, optimization, and stochastic modeling technologies*, p. 1–75.
- Glover, F. and Laguna, M. (2019). *Tabu Search*, volume 272 of *International Series in Operations Research & Management Science*, p. 37–55. Springer, Cham. URL https://link.springer.com/chapter/10.1007/978-3-319-91086-4_2.
- Gmys, J., Carneiro, T., Melab, N., Talbi, E.-G., and Tuytens, D. (2020). A comparative study of high-productivity high-performance programming languages for parallel metaheuristics. *Swarm and Evolutionary Computation*, 57:100720. ISSN 2210-6502. URL <https://www.sciencedirect.com/science/article/pii/S2210650220303734>.
- Holland, J. H. (1984). Genetic algorithms and adaptation. *Adaptive control of ill-defined systems*, p. 317–333.

- JuliaLang (2025). Garbage collection in julia · the julia language. URL <https://docs.julialang.org/en/v1/devdocs/gc/>. Accessed: 2025-03-02.
- Kirkpatrick, S., Gelatt Jr, C. D., and Vecchi, M. P. (1983). Optimization by simulated annealing. *science*, 220(4598):671–680.
- López-Ibáñez, M., Dubois-Lacoste, J., Cáceres, L. P., Birattari, M., and Stützle, T. (2016). The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58.
- Lourenço, H. R., Martin, O. C., and Stützle, T. (2003). Iterated local search. In *Handbook of metaheuristics*, p. 320–353. Springer.
- Lust, T. and Teghem, J. (2010). Two-phase pareto local search for the biobjective traveling salesman problem. *Journal of Heuristics*, 16(3):475–510.
- Mladenović, N. and Hansen, P. (1997). Variable neighborhood search. *Computers & operations research*, 24(11):1097–1100.
- Moscato, P. and Cotta, C. (2003). A gentle introduction to memetic algorithms. In *Handbook of metaheuristics*, p. 105–144. Springer.
- Python Software Foundation (2025). gc — garbage collector interface. URL <https://docs.python.org/3/library/gc.html>. Accessed: 2025-03-02.
- Ramírez, A., Barbudo, R., and Romero, J. R. (2023). An experimental comparison of metaheuristic frameworks for multi-objective optimization. *Expert Systems*, 40(4):e12672. URL <https://onlinelibrary.wiley.com/doi/abs/10.1111/exsy.12672>.
- Resende, M. G. and Ribeiro, C. C. (2016a). Biased ranom-key genetic algorithms: an advanced tutorial. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion*, p. 483–514.
- Resende, M. G. and Ribeiro, C. C. (2016b). *Optimization by GRASP*. Springer.
- Silva, M. A. L., de Souza, S. R., Souza, M. J. F., and Bazzan, A. L. C. (2019). A reinforcement learning-based multi-agent framework applied for solving routing and scheduling problems. *Expert Systems with Applications*, 131:148–171.
- Talbi, E.-G. (2009). *Metaheuristics: From Design to Implementation*. John Wiley & Sons, Hoboken, NJ. ISBN 9780470278581. URL <https://www.wiley.com/en-us/Metaheuristics%3A+From+Design+to+Implementation+-p-9780470278581>.
- Talbi, E.-G., editor (2013). *Hybrid Metaheuristics*, volume 434 of *Studies in Computational Intelligence*. Springer, Berlin, Heidelberg. ISBN 978-3-642-30671-6. URL <https://link.springer.com/book/10.1007/978-3-642-30671-6>.
- W3Computing (2023). Resource acquisition is initialization (raii) in c++. URL <https://www.w3computing.com/articles/resource-acquisition-is-initialization-raii-in-cpp/>. Accessed: 2025-05-14.

Yamazaki, T., Nakamaru, T., Shioya, R., Ugawa, T., and Chiba, S. (2023). Collecting cyclic garbage across foreign function interfaces: Who takes the last piece of cake? *Proceedings of the ACM on Programming Languages*, 7(PLDI):Article 130. URL <https://doi.org/10.1145/3591244>.